Pintos

by Ben Pfaff
Virginia Tech modifications by Godmar Back

Table of Contents

1 Introduction

Welcome to Pintos. Pintos is a simple operating system framework for the 80x86 architecture. It supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way. In the Pintos projects, you and your project team will strengthen its support in all three of these areas. You will also add a virtual memory implementation.

Pintos could, theoretically, run on a regular IBM-compatible PC. Unfortunately, it is impractical to supply every CS 3204 student a dedicated PC for use with Pintos. Therefore, we will run Pintos projects in a system simulator, that is, a program that simulates an 80x86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it. In class we will use the Bochs and qemu simulators. Pintos has also been tested with VMware GSX Server.

These projects are hard. We will do what we can to reduce the workload, such as providing a lot of support material, but there is plenty of hard work that needs to be done. We welcome your feedback. If you have suggestions on how we can reduce the unnecessary overhead of assignments, cutting them down to the important underlying issues, please let us know.

This chapter explains how to get started working with Pintos. You should read the entire chapter before you start work on any of the projects.

1.1 Getting Started

To get started, you'll have to log into a machine that Pintos can be built on. The CS 3204 "officially supported" Pintos development machines are the machines in McBryde 124. You can log on to those machines remotely using

```
ssh -X yourlogin@rlogin.cs.vt.edu
```

We will test your code on these machines, and the instructions given here assume this environment. While Pintos and its supporting tools are portable enough that it should build "out of the box" in other environments, we will not provide any support for doing so.

Once you've logged into one of these machines, either locally or remotely, start out by adding our binaries directory to your `PATH` environment. Under `csh` you can do so with this command:

```
set path = ( ~cs3204/bin $path )
```

It is a good idea to add this line to the '`.cshrc`' file in your home directory. Otherwise, you'll have to type it every time you log in. `bash` shell users would instead add

```
export PATH=~cs3204/bin:$PATH
```

to their '`.bashrc`' file.

1.1.1 Source Tree Overview

Now you can extract the source for Pintos into a directory named '`pintos/src`', by executing

```
tar xzf /home/courses/cs3204/pintos/pintos.tar.gz
```

Alternatively, fetch `http://courses.cs.vt.edu/~cs3204/spring2006/gback/pintos/pintos.tar.gz` and extract it in a similar way.

Let's take a look at what's inside. Here's the directory structure that you should see in '`pintos/src`':

'`threads/`'
: Source code for the base kernel, which you will modify starting in project 1.

'`userprog/`'
: Source code for the user program loader, which you will modify starting with project 2.

'`vm/`'
: An almost empty directory. You will implement virtual memory here in project 3.

'`filesys/`'
: Source code for a basic file system. You will use this file system starting with project 2, but you will not modify it until project 4.

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

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

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

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

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

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

'`misc/`'

'`utils/`'
: These files may come in handy if you decide to try working with Pintos away from the lab machines. Otherwise, you can ignore them.

1.1.2 Building Pintos

As the next step, build the source code supplied for the first project. First, `cd` into the '`threads`' directory. Then, issue the '`make`' command. This will create a '`build`' directory

under '`threads`', populate it with a '`Makefile`' and a few subdirectories, and then build the kernel inside. The entire build should take less than 5 seconds.

Following the build, the following are the interesting files in the '`build`' directory:

'`Makefile`'
: A copy of '`pintos/src/Makefile.build`'. It describes how to build the kernel. See [Adding Source Files], page 41, for more information.

'`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 `backtrace` (see Section E.4 [Backtraces], page 93) on it.

'`kernel.bin`'
: Memory image of the kernel. These are the exact bytes loaded into memory to run the Pintos kernel. To simplify loading, it is always padded out with zero bytes up to an exact multiple of 4 kB in size.

'`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.

'`os.dsk`' Disk image for the kernel, which is just '`loader.bin`' followed by '`kernel.bin`'. This file is used as a "virtual disk" by the simulator.

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

1.1.3 Running Pintos

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

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

This command creates a '`bochsrc.txt`' file, which is needed for running Bochs, and then invoke Bochs. Bochs opens a new window that represents the simulated machine's display, and a BIOS message briefly flashes. Then Pintos boots and runs the `alarm-multiple` test program, which outputs a few screenfuls of text. When it's done, you can close Bochs by clicking on the "Power" button in the window's top right corner, or rerun the whole process by clicking on the "Reset" button just to its left. The other buttons are not very useful for our purposes.

(If no window appeared at all, and you just got a terminal full of corrupt-looking text, then you're probably logged in remotely and X forwarding is not set up correctly. In this case, you can fix your X setup, or you can use the '`-v`' option to disable X output: `pintos -v -- run alarm-multiple`.)

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

The `pintos` program offers several options for configuring the simulator or the virtual hardware. If you specify any options, they must precede the commands passed to the Pintos kernel and be separated from them by '`--`', so that the whole command looks like `pintos` *option*`...` `--` *argument*`...`. Invoke `pintos` without any arguments to see a list of available options. Options can select a simulator to use: the default is Bochs, but on the Linux machines '`--qemu`' selects qemu. You can run the simulator with a debugger (see Section E.5 [gdb], page 95). You can set the amount of memory to give the VM. Finally, you can select how you want VM output to be displayed: use '`-v`' to turn off the VGA display, '`-t`' to use your terminal window as the VGA display instead of opening a new window (Bochs only), or '`-s`' to suppress the serial output to `stdout`.

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

1.1.4 Debugging versus Testing

When you're debugging code, it's useful to be able to run a program twice and have it do exactly the same thing. On second and later runs, you can make new observations without having to discard or verify your old observations. This property is called "reproducibility." The simulator we use by default, Bochs, can be set up for reproducibility, and that's the way that `pintos` invokes it by default.

Of course, a simulation can only be reproducible from one run to the next if its input is the same each time. For simulating an entire computer, as we do, this means that every part of the computer must be the same. For example, you must use the same command-line argument, the same disks, the same version of Bochs, and you must not hit any keys on the keyboard (because you could not be sure to hit them at exactly the same point each time) during the runs.

While reproducibility is useful for debugging, it is a problem for testing thread synchronization, an important part of most of the projects. In particular, when Bochs is set up for reproducibility, timer interrupts will come at perfectly reproducible points, and therefore so will thread switches. That means that running the same test several times doesn't give you any greater confidence in your code's correctness than does running it only once.

So, to make your code easier to test, we've added a feature, called "jitter," to Bochs, that makes timer interrupts come at random intervals, but in a perfectly predictable way. In particular, if you invoke `pintos` with the option '`-j` *seed*', timer interrupts will come at irregularly spaced intervals. Within a single *seed* value, execution will still be reproducible, but timer behavior will change as *seed* is varied. Thus, for the highest degree of confidence you should test your code with many seed values.

On the other hand, when Bochs runs in reproducible mode, timings are not realistic, meaning that a "one-second" delay may be much shorter or even much longer than one second. You can invoke `pintos` with a different option, '`-r`', to set up Bochs for realistic

timings, in which a one-second delay should take approximately one second of real time. Simulation in real-time mode is not reproducible, and options '`-j`' and '`-r`' are mutually exclusive.

On the Linux machines only, the qemu simulator is available as an alternative to Bochs (use '`--qemu`' when invoking `pintos`). The qemu simulator is much faster than Bochs, but it only supports real-time simulation and does not have a reproducible mode.

1.2 Grading

We will grade your assignments based on test results and design quality, each of which comprises 50% of your grade.

1.2.1 Testing

Your test result grade will be based on our tests. Each project has several tests, each of which has a name beginning with '`tests`'. To completely test your submission, invoke `make check` from the project '`build`' directory. This will build and run each test and print a "pass" or "fail" message for each one. When a test fails, `make check` also prints some details of the reason for failure. After running all the tests, `make check` also prints a summary of the test results.

For project 1, the tests will probably run faster in Bochs. For the rest of the projects, they will probably run faster in qemu.

You can also run individual tests one at a time. A given test *t* writes its output to '*t*`.output`', then a script scores the output as "pass" or "fail" and writes the verdict to '*t*`.result`'. To run and grade a single test, `make` the '`.result`' file explicitly from the '`build`' directory, e.g. `make tests/threads/alarm-multiple.result`. If `make` says that the test result is up-to-date, but you want to re-run it anyway, either run `make clean` or delete the '`.output`' file by hand.

By default, each test provides feedback only at completion, not during its run. If you prefer, you can observe the progress of each test by specifying '`VERBOSE=1`' on the `make` command line, as in `make check VERBOSE=1`. You can also provide arbitrary options to the `pintos` run by the tests with '`PINTOSOPTS='...'`', e.g. `make check PINTOSOPTS='--qemu'` to run the tests under qemu.

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

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

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

1.2.2 Design

We will judge your design based on the design document and the source code that you submit. We will read your entire design document and much of your source code.

Don't forget that the design document is 50% of your project grade. It is better to spend one or two hours writing a good design document than it is to spend that time getting the last 5% of the points for tests and then trying to rush through writing the design document in the last 15 minutes.

1.2.2.1 Design Document

We provide a design document template for each project. For each significant part of a project, the template asks questions in four areas:

Data Structures

The instructions for this section are always the same:

> Copy here the declaration of each new or changed `struct` or `struct` member, global or static variable, `typedef`, or enumeration. Identify the purpose of each in 25 words or less.

The first part is mechanical. Just copy new or modified declarations into the design document, to highlight for us the actual changes to data structures. Each declaration should include the comment that should accompany it in the source code (see below).

We also ask for a very brief description of the purpose of each new or changed data structure. The limit of 25 words or less is a guideline intended to save your time and avoid duplication with later areas.

Algorithms

This is where you tell us how your code works, through questions that probe your understanding of your code. We might not be able to easily figure it out from the code, because many creative solutions exist for most OS problems. Help us out a little.

Your answers should be at a level below the high level description of requirements given in the assignment. We have read the assignment too, so it is unnecessary to repeat or rephrase what is stated there. On the other hand, your answers should be at a level above the low level of the code itself. Don't give a line-by-line run-down of what your code does. Instead, use your answers to explain how your code works to implement the requirements.

Synchronization

An operating system kernel is a complex, multithreaded program, in which synchronizing multiple threads can be difficult. This section asks about how you chose to synchronize this particular type of activity.

Rationale

Whereas the other sections primarily ask "what" and "how," the rationale section concentrates on "why." This is where we ask you to justify some design decisions, by explaining why the choices you made are better than alternatives. You may be able to state these in terms of time and space complexity, which

can be made as rough or informal arguments (formal language or proofs are unnecessary).

An incomplete, evasive, or non-responsive design document or one that strays from the template without good reason may be penalized. Incorrect capitalization, punctuation, spelling, or grammar can also cost points. See Appendix D [Project Documentation], page 89, for a sample design document for a fictitious project.

1.2.2.2 Source Code

Your design will also be judged by looking at your source code. We will typically look at the differences between the original Pintos source tree and your submission, based on the output of a command like `diff -urpb pintos.orig pintos.submitted`. We will try to match up your description of the design with the code submitted. Important discrepancies between the description and the actual code will be penalized, as will be any bugs we find by spot checks.

The most important aspects of source code design are those that specifically relate to the operating system issues at stake in the project. For example, the organization of an inode is an important part of file system design, so in the file system project a poorly designed inode would lose points. Other issues are much less important. For example, multiple Pintos design problems call for a "priority queue," that is, a dynamic collection from which the minimum (or maximum) item can quickly be extracted. Fast priority queues can be implemented many ways, but we do not expect you to build a fancy data structure even if it might improve performance. Instead, you are welcome to use a linked list (and Pintos even provides one with convenient functions for sorting and finding minimums and maximums).

Pintos is written in a consistent style. Make your additions and modifications in existing Pintos source files blend in, not stick out. In new source files, adopt the existing Pintos style by preference, but make your code self-consistent at the very least. There should not be a patchwork of different styles that makes it obvious that three different people wrote the code. Use horizontal and vertical white space to make code readable. Add a brief comment on every structure, structure member, global or static variable, and function definition. Update existing comments as you modify code. Don't comment out or use the preprocessor to ignore blocks of code (instead, remove it entirely). Use assertions to document key invariants. Decompose code into functions for clarity. Code that is difficult to understand because it violates these or other "common sense" software engineering practices will be penalized.

In the end, remember your audience. Code is written primarily to be read by humans. It has to be acceptable to the compiler too, but the compiler doesn't care about how it looks or how well it is written.

1.3 License

Pintos was developed at Stanford University by Ben Pfaff and others. Pintos is distributed under a liberal license that allows free use, modification, and distribution. Students and others who work on Pintos own the code that they write and may use it for any purpose.

In the context of Virginia Tech's CS 3204 course, please respect the spirit and the letter of the honor code by refraining from reading any homework solutions available online or elsewhere. Reading the source code for other operating system kernels, such as Linux or

FreeBSD, is allowed, but do not copy code from them literally. Please cite the code that inspired your own in your design documentation.

Pintos comes with NO WARRANTY, not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

The '`LICENSE`' file at the top level of the Pintos source distribution has full details of the license and lack of warranty.

1.4 Acknowledgements

Pintos and this documentation were written by Ben Pfaff `blp@cs.stanford.edu`. Adjustments for Virginia Tech's CS 3204 were made by Godmar Back. Vijay Kumar wrote the section on how to set up CVS.

The original structure and form of Pintos was inspired by the Nachos instructional operating system from the University of California, Berkeley. A few of the source files were originally more-or-less literal translations of the Nachos C++ code into C. These files bear the original UCB license notice.

A few of the Pintos source files are derived from code used in the Massachusetts Institute of Technology's 6.828 advanced operating systems course. These files bear the original MIT license notice.

The Pintos projects and documentation originated with those designed for Nachos by current and former CS140 teaching assistants at Stanford University, including at least Yu Ping, Greg Hutchins, Kelly Shaw, Paul Twohey, Sameer Qureshi, and John Rector. If you're not on this list but should be, please let me know.

Example code for condition variables (see Section 2.2.2.4 [Condition Variables], page 19) is from classroom slides originally by Dawson Engler and updated by Mendel Rosenblum.

1.5 Trivia

Pintos originated as a replacement for Nachos with a similar design. Since then Pintos has greatly diverged from the Nachos design. Pintos differs from Nachos in two important ways. First, Pintos runs on real or simulated 80*x*86 hardware, but Nachos runs as a process on a host operating system. Second, Pintos is written in C like most real-world operating systems, but Nachos is written in C++.

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

2 A Tour Through Pintos

This chapter is a brief tour through the Pintos code. It covers the entire code base, but you'll only be using Pintos one part at a time, so you may find that you want to read each part as you work on the corresponding project.

(Actually, the tour is currently incomplete. It fully covers only the threads project.)

We recommend using "tags" to follow along with references to function and variable names (see Section F.1 [Tags], page 98).

2.1 Loading

This section covers the Pintos loader and basic kernel initialization.

2.1.1 The Loader

The first part of Pintos that runs is the loader, in '`threads/loader.S`'. The PC BIOS loads the loader into memory. The loader, in turn, is responsible for initializing the CPU, loading the rest of Pintos into memory, and then jumping to its start. It's not important to understand exactly what the loader does, but if you're interested, read on. You should probably read along with the loader's source. You should also understand the basics of the 80*x*86 architecture as described by chapter 3 of [IA32-v1].

Because the PC BIOS loads the loader, the loader has to play by the BIOS's rules. In particular, the BIOS only loads 512 bytes (one disk sector) into memory. This is a severe restriction and it means that, practically speaking, the loader has to be written in assembly language.

Pintos' loader first initializes the CPU. The first important part of this is to enable the A20 line, that is, the CPU's address line numbered 20. For historical reasons, PCs start out with this address line fixed at 0, which means that attempts to access memory beyond the first 1 MB (2 raised to the 20th power) will fail. Pintos wants to access more memory than this, so we have to enable it.

Next, the loader asks the BIOS for the PC's memory size. Again for historical reasons, the function that we call in the BIOS to do this can only detect up to 64 MB of RAM, so that's the practical limit that Pintos can support. The memory size is stashed away in a location in the loader that the kernel can read after it boots.

Third, the loader creates a basic page table. This page table maps the 64 MB at the base of virtual memory (starting at virtual address 0) directly to the identical physical addresses. It also maps the same physical memory starting at virtual address `LOADER_PHYS_BASE`, which defaults to `0xc0000000` (3 GB). The Pintos kernel only wants the latter mapping, but there's a chicken-and-egg problem if we don't include the former: our current virtual address is roughly `0x7c00`, the location where the BIOS loaded us, and we can't jump to `0xc0007c00` until we turn on the page table, but if we turn on the page table without jumping there, then we've just pulled the rug out from under ourselves.

After the page table is initialized, we load the CPU's control registers to turn on protected mode and paging, and then we set up the segment registers. We aren't equipped to handle interrupts in protected mode yet, so we disable interrupts.

Finally it's time to load the kernel from disk. We use a simple but inflexible method to do this: we program the IDE disk controller directly. We assume that the kernel is stored

starting from the second sector of the first IDE disk (the first sector normally contains the boot loader). We also assume that the BIOS has already set up the IDE controller for us. We read `KERNEL_LOAD_PAGES` pages of data (4 kB per page) from the disk directly into virtual memory, starting `LOADER_KERN_BASE` bytes past `LOADER_PHYS_BASE`, which by default means that we load the kernel starting 1 MB into physical memory.

Then we jump to the start of the compiled kernel image. Using the "linker script" in '`threads/kernel.lds.S`', the kernel has arranged that it begins with the assembly module '`threads/start.S`'. This assembly module just calls `main()`, which never returns.

There's one more trick: the Pintos kernel command line is stored in the boot loader. The `pintos` program actually modifies the boot loader on disk each time it runs the kernel, putting in whatever command line arguments the user supplies to the kernel, and then the kernel at boot time reads those arguments out of the boot loader in memory. This is something of a nasty hack, but it is simple and effective.

2.1.2 Kernel Initialization

The kernel proper starts with the `main()` function. The `main()` function is written in C, as will be most of the code we encounter in Pintos from here on out.

When `main()` starts, the system is in a pretty raw state. We're in protected mode with paging enabled, but hardly anything else is ready. Thus, the `main()` function consists primarily of calls into other Pintos modules' initialization functions. These are usually named *`module`*`_init()`, where *module* is the module's name, '*`module`*`.c`' is the module's source code, and '*`module`*`.h`' is the module's header.

First we initialize kernel RAM in `ram_init()`. The first step is to clear out the kernel's so-called "BSS" segment. The BSS is a segment that should be initialized to all zeros. In most C implementations, whenever you declare a variable outside a function without providing an initializer, that variable goes into the BSS. Because it's all zeros, the BSS isn't stored in the image that the loader brought into memory. We just use `memset()` to zero it out. The other task of `ram_init()` is to read out the machine's memory size from where the loader stored it and put it into the `ram_pages` variable for later use.

Next, `thread_init()` initializes the thread system. We will defer full discussion to our discussion of Pintos threads below. It is called so early in initialization because the console, initialized just afterward, tries to use locks, and locks in turn require there to be a running thread.

Then we initialize the console so that we can use `printf()`. `main()` calls `vga_init()`, which initializes the VGA text display and clears the screen. It also calls `serial_init_poll()` to initialize the first serial port in "polling mode," that is, where the kernel busy-waits for the port to be ready for each character to be output. (We use polling mode until we're ready to set up interrupts later.) Finally we initialize the console device and print a startup message to the console.

`main()` calls `read_command_line()` to break the kernel command line into arguments, then `parse_options()` to read any options at the beginning of the command line. (Executing actions specified on the command line happens later.)

The next block of functions we call initialize the kernel's memory system. `palloc_init()` sets up the kernel page allocator, which doles out memory one or more pages at

a time. `malloc_init()` sets up the allocator that handles odd-sized allocations. `paging_init()` sets up a page table for the kernel.

In projects 2 and later, `main()` also calls `tss_init()` and `gdt_init()`, but we'll talk about those later.

`main()` calls `random_init()` to initialize the kernel random number generator. If the user specified '`-rs`' on the `pintos` command line, `parse_options()` has already done this, but calling it a second time is harmless and has no effect.

We initialize the interrupt system in the next set of calls. `intr_init()` sets up the CPU's *interrupt descriptor table* (IDT) to ready it for interrupt handling (see Section 2.2.3.1 [Interrupt Infrastructure], page 22), then `timer_init()` and `kbd_init()` prepare for handling timer interrupts and keyboard interrupts, respectively. In projects 2 and later, we also prepare to handle interrupts caused by user programs using `exception_init()` and `syscall_init()`.

Now that interrupts are set up, we can start preemptively scheduling threads with `thread_start()`, which also enables interrupts. With interrupts enabled, interrupt-driven serial port I/O becomes possible, so we use `serial_init_queue()` to switch to that mode. Finally, `timer_calibrate()` calibrates the timer for accurate short delays.

If the file system is compiled in, as it will starting in project 2, we now initialize the disks with `disk_init()`, then the file system with `filesys_init()`.

Boot is complete, so we print a message.

Function `run_actions()` now parses and executes actions specified on the kernel command line, such as `run` to run a test (in project 1) or a user program (in later projects).

Finally, if '`-q`' was specified on the kernel command line, we call `power_off()` to terminate the machine simulator. Otherwise, `main()` calls `thread_exit()`, which allows any other running threads to continue running.

2.2 Threads Project

2.2.1 Thread Support

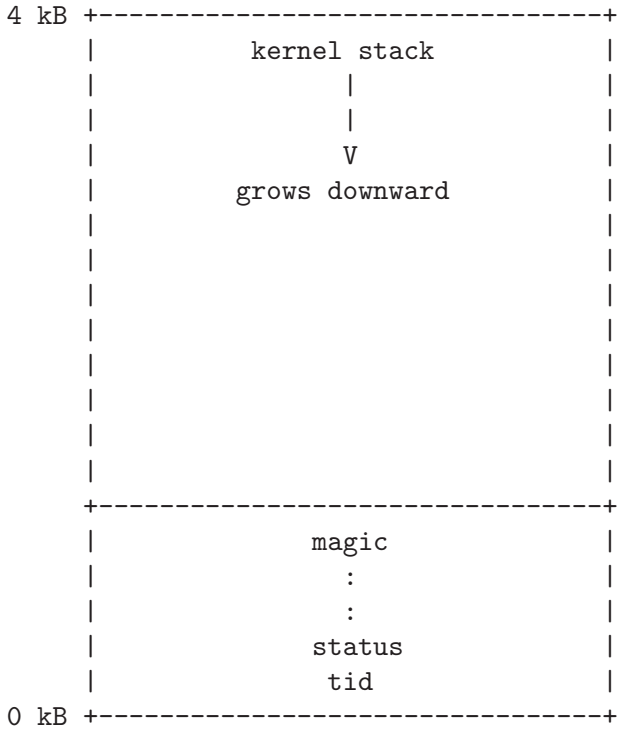
2.2.1.1 `struct thread`

The main Pintos data structure for threads is `struct thread`, declared in '`threads/thread.h`'.

`struct thread` [Structure]

Represents a thread or a user process. In the projects, you will have to add your own members to `struct thread`. You may also change or delete the definitions of existing members.

Every `struct thread` occupies the beginning of its own 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 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 Section 2.2.4 [Memory Allocation], page 25).

`tid_t tid` [Member of `struct thread`]

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. You can change the type and the numbering scheme if you like.

`enum thread_status status` [Member of `struct thread`]

The thread's state, one of the following:

`THREAD_RUNNING` [Thread State]

The thread is running. Exactly one thread is running at a given time. `thread_current()` returns the running thread.

`THREAD_READY` [Thread State]

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_BLOCKED` [Thread State]
: 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()`.

`THREAD_DYING` [Thread State]
: The thread will be destroyed by the scheduler after switching to the next thread.

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

`uint8_t * stack` [Member of `struct thread`]
: 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. See Section 2.2.3 [Interrupt Handling], page 21, for more information.

`int priority` [Member of `struct thread`]
: A thread priority, ranging from `PRI_MIN` (0) to `PRI_MAX` (63). Lower numbers correspond to *higher* priorities, so that priority 0 is the highest priority and priority 63 is the lowest. Pintos as provided ignores thread priorities, but you will implement priority scheduling in project 1 (see Section 3.2.3 [Priority Scheduling], page 40).

`struct list_elem elem` [Member of `struct thread`]
: A "list element" used to put the thread into doubly linked lists, either the list of threads ready to run or a list of threads waiting on a semaphore. Take a look at '`lib/kernel/list.h`' for information on how to use Pintos doubly linked lists.

`uint32_t * pagedir` [Member of `struct thread`]
: Only present in project 2 and later.

`unsigned magic` [Member of `struct thread`]
: Always set to `THREAD_MAGIC`, which is just a random 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 will normally change this value, triggering the assertion. For greatest benefit, as you add members to `struct thread`, leave `magic` as the final member.

2.2.1.2 Thread Functions

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

`void thread_init` (*void*) [Function]
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.

`void thread_start` (*void*) [Function]
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()` (see Section 2.2.3.3 [External Interrupt Handling], page 24).

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

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

`tid_t thread_create` (*const char *name*, *int priority*, *thread_func *func*, *void *aux*) [Function]
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 `struct thread` and stack and initializes its members, then it sets up a set of fake stack frames for it (more about this later). The thread is initialized in the blocked state, so the final action before returning is to unblock it, which allows the new thread to be scheduled.

`void thread_func (void *aux)` [Type]
This is the type of a thread function. Its *aux* argument is the value passed to `thread_create()`.

`void thread_block` (*void*) [Function]
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 (see Section 2.2.2 [Synchronization], page 16).

`void thread_unblock` (*struct thread *thread*) [Function]
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.

`struct thread * thread_current` (*void*) [Function]
Returns the running thread.

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

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

`void thread_exit` (*void*) `NO_RETURN` [Function]
Causes the current thread to exit. Never returns, hence `NO_RETURN` (see Section E.3 [Function and Parameter Attributes], page 92).

`void thread_yield` (*void*) [Function]
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.

`int thread_get_priority` (*void*) [Function]
`void thread_set_priority` (*int* *new_priority*) [Function]
Skeleton to set and get thread priority. See Section 3.2.3 [Priority Scheduling], page 40.

`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*) [Function]
Skeletons for the advanced scheduler. See Appendix B [4.4BSD Scheduler], page 81.

2.2.1.3 Thread Switching

`schedule()` is the function 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 simple 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 as `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, a 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,[1] calls `schedule_tail()` (this special case is why `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.

2.2.2 Synchronization

If sharing of resources between threads is not handled in a careful, controlled fashion, then 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.

2.2.2.1 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

[1] This is because `switch_threads()` takes arguments on the stack and the 80x86 SVR4 calling convention requires the caller, not the called function, to remove them when the call is complete. See [SysV-i386] chapter 3 for details.

handlers, which cannot sleep and thus cannot use most other forms of synchronization (see Section 2.2.3.3 [External Interrupt Handling], page 24).

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

`enum intr_level` [Type]
One of `INTR_OFF` or `INTR_ON`, denoting that interrupts are disabled or enabled, respectively.

`enum intr_level intr_get_level` (*void*) [Function]
Returns the current interrupt state.

`enum intr_level intr_set_level` (*enum intr_level* `level`) [Function]
Turns interrupts on or off according to *level*. Returns the previous interrupt state.

`enum intr_level intr_enable` (*void*) [Function]
Turns interrupts on. Returns the previous interrupt state.

`enum intr_level intr_disable` (*void*) [Function]
Turns interrupts off. Returns the previous interrupt state.

2.2.2.2 Semaphores

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

`struct semaphore` [Type]
Represents a *semaphore*, 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 values larger than 1. These are rarely used.

`void sema_init` (*struct semaphore* *`sema`, *unsigned* `value`) [Function]
Initializes *sema* as a new semaphore with the given initial *value*.

`void sema_down` (*struct semaphore* *`sema`) [Function]
Executes the "down" or "P" operation on *sema*, waiting for its value to become positive and then decrementing it by one.

`bool sema_try_down` (*struct semaphore* ***sema**) [Function]
Tries to execute the "down" or "P" operation on *sema*, without waiting. Returns true if *sema* had a positive value that was successfully decremented, or false if it was already zero and thus could not be decremented. Calling this function in a tight loop wastes CPU time (use `sema_down()` instead, or find a different approach).

`void sema_up` (*struct semaphore* ***sema**) [Function]
Executes the "up" or "V" operation on *sema*, incrementing its value. If any threads are waiting on *sema*, wakes one of them up.

Semaphores are internally built out of disabling interrupt (see Section 2.2.2.1 [Disabling Interrupts], page 16) 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`'.

2.2.2.3 Locks

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

`struct lock` [Type]
Represents a *lock*, a specialized semaphore with an initial value of 1 (see Section 2.2.2.2 [Semaphores], page 17). The difference between a lock and such a semaphore is twofold. First, a semaphore does not have an owner, meaning that one thread can "down" the semaphore and then another one "up" it, but a single thread must both acquire and release a lock. Second, a semaphore can have a value greater than 1, but a lock can only be owned by a single thread at a time. If these restrictions prove onerous, 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.

`void lock_init` (*struct lock* ***lock**) [Function]
Initializes *lock* as a new lock.

`void lock_acquire` (*struct lock* ***lock**) [Function]
Acquires *lock* for use by the current thread, first waiting for any current owner to release it if necessary.

`bool lock_try_acquire` (*struct lock* ***lock**) [Function]
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 (use `lock_acquire()` instead).

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

`bool lock_held_by_current_thread` (*const struct lock* ***lock**) [Function]
Returns true if the running thread owns *lock*, false otherwise.

2.2.2.4 Condition Variables

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

`struct condition` [Type]
Represents a condition variable, which allows one piece of code to signal a condition and cooperating code to receive the signal and act upon it. Each condition variable is associated with a lock. A given condition variable is associated with only a single lock, but one lock may be associated with any number of condition variables. A set of condition variables taken together with their lock is called a "monitor."

A thread that owns the monitor lock is said to be "in the monitor." The thread in the monitor has control over all the data protected by the lock. It may freely examine or modify this data. If it discovers that it needs to wait for some condition to become true, then it "waits" on the associated condition, 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.

Pintos monitors are "Mesa" style, not "Hoare" style. That is, sending and receiving a signal are not an atomic operation. Thus, typically the caller must recheck the condition after the wait completes and, if necessary, wait again.

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

`void cond_wait` (*struct condition* `*cond`, *struct lock* `*lock`) [Function]
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.

`void cond_signal` (*struct condition* `*cond`, *struct lock* `*lock`) [Function]
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.

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

Monitor Example

The classical example of a monitor is handling a buffer into which one "producer" thread writes characters and out of which a second "consumer" thread reads characters. To implement this case we need, besides the monitor lock, two condition variables which we will call *not_full* and *not_empty*:

```
char buf[BUF_SIZE];     /* Buffer. */
size_t n = 0;           /* 0 <= n <= BUF_SIZE: # of characters in buffer. */
size_t head = 0;        /* buf index of next char to write (mod BUF_SIZE). */
size_t tail = 0;        /* buf index of next char to read (mod BUF_SIZE). */
struct lock lock;       /* Monitor lock. */
```

```
struct condition not_empty; /* Signaled when the buffer is not empty. */
struct condition not_full; /* Signaled when the buffer is not full. */

...initialize the locks and condition variables...

void put (char ch) {
  lock_acquire (&lock);
  while (n == BUF_SIZE)             /* Can't add to buf as long as it's full. */
    cond_wait (&not_full, &lock);
  buf[head++ % BUF_SIZE] = ch;      /* Add ch to buf. */
  n++;
  cond_signal (&not_empty, &lock); /* buf can't be empty anymore. */
  lock_release (&lock);
}

char get (void) {
  char ch;
  lock_acquire (&lock);
  while (n == 0)                   /* Can't read buf as long as it's empty. */
    cond_wait (&not_empty, &lock);
  ch = buf[tail++ % BUF_SIZE];     /* Get ch from buf. */
  n--;
  cond_signal (&not_full, &lock); /* buf can't be full anymore. */
  lock_release (&lock);
}
```

2.2.2.5 Memory Barriers

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.

If interrupts are enabled, this code for setting up '`x`' to be printed is clearly incorrect, because the timer interrupt could intervene between the two assignments:

```
timer_do_put = true;            /* INCORRECT CODE */
timer_put_char = 'x';
```

It might not be as obvious that the following code is just as incorrect:

```
timer_put_char = 'x';           /* INCORRECT CODE */
timer_do_put = true;
```

The reason this second example might be a problem is that the compiler is, in general, free to reorder operations when it doesn't have a visible 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 optimization pass 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 changing compiler versions will produce different behavior.

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);
```

Fortunately, real solutions do exist. One possibility is to disable interrupts around the assignments. This does not prevent reordering, but it makes the assignments atomic as observed by the interrupt handler. 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 possibility 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 allows it less latitude for optimization. However, the semantics of '`volatile`' are not well-defined, so it is not a good general solution.

Usually, the best solution is to use a compiler feature called a *memory barrier*, a special statement that prevents the compiler from reordering memory operations across the barrier. In Pintos, '`threads/synch.h`' defines the `barrier()` macro as a memory barrier. Here's how we would use a memory barrier to fix this code:

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

The compiler also treats invocation of any function defined externally, that is, in another source file, as a limited form of a memory 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, and, indeed, this is why the base Pintos code does not need any barriers.

A function defined in the same source file, or in a header included by the source file, cannot be relied upon as a memory 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.

2.2.3 Interrupt Handling

An *interrupt* notifies the CPU of some event. Much of the work of an operating system relates to interrupts in one way or another. For our purposes, we classify interrupts into two broad categories:

- *External interrupts*, that is, interrupts originating outside the CPU. These interrupts come from hardware devices such as the system timer, keyboard, serial ports, and disks. External interrupts are *asynchronous*, meaning that their delivery is not synchronized with normal CPU activities. External interrupts are what `intr_disable()` and related functions postpone (see Section 2.2.2.1 [Disabling Interrupts], page 16).
- *Internal interrupts*, that is, interrupts caused by something executing on the CPU. These interrupts are caused by something unusual happening during instruction execution: accessing invalid memory (a *page fault*), executing invalid instructions, and vari-

ous other disallowed activities. Because they are caused by CPU instructions, internal interrupts are *synchronous* or synchronized with CPU instructions. `intr_disable()` does not disable internal interrupts.

Because the CPU treats all interrupts largely the same way, regardless of source, Pintos uses the same infrastructure for both internal and external interrupts, to a point. The following section describes this common infrastructure, and sections after that give the specifics of external and internal interrupts.

If you haven't already read chapter 3 in [IA32-v1], it is recommended that you do so now. You might also want to skim chapter 5 in [IA32-v3].

2.2.3.1 Interrupt Infrastructure

When an interrupt occurs while the kernel is running, the CPU saves its most essential state on the stack and jumps to an interrupt handler routine. The 80*x*86 architecture allows for 256 possible interrupts, each of which can have its own handler. The handler for each interrupt is defined in an array called the *interrupt descriptor table* or IDT.

In Pintos, `intr_init()` in '`threads/interrupt.c`' sets up the IDT so that each entry points to a unique entry point in '`threads/intr-stubs.S`' named `intr`*NN*`_stub()`, where *NN* is the interrupt number in hexadecimal. Because the CPU doesn't give us any other way to find out the interrupt number, this entry point pushes the interrupt number on the stack. Then it jumps to `intr_entry()`, which pushes all the registers that the processor didn't already save for us, and then calls `intr_handler()`, which brings us back into C in '`threads/interrupt.c`'.

The main job of `intr_handler()` is to call any function that has been registered for handling the particular interrupt. (If no function is registered, it dumps some information to the console and panics.) It does some extra processing for external interrupts that we'll discuss later.

When `intr_handler()` returns, the assembly code in '`threads/intr-stubs.S`' restores all the CPU registers saved earlier and directs the CPU to return from the interrupt.

A few types and functions apply to both internal and external interrupts.

`void intr_handler_func (struct intr_frame *`*frame*`)` [Type]
: This is how an interrupt handler function must be declared. Its *frame* argument (see below) allows it to determine the cause of the interrupt and the state of the thread that was interrupted.

`struct intr_frame` [Type]
: The stack frame of an interrupt handler, as saved by CPU, the interrupt stubs, and `intr_entry()`. Its most interesting members are described below.

`uint32_t edi` [Member of `struct intr_frame`]
`uint32_t esi` [Member of `struct intr_frame`]
`uint32_t ebp` [Member of `struct intr_frame`]
`uint32_t esp_dummy` [Member of `struct intr_frame`]
`uint32_t ebx` [Member of `struct intr_frame`]
`uint32_t edx` [Member of `struct intr_frame`]
`uint32_t ecx` [Member of `struct intr_frame`]

`uint32_t eax` [Member of `struct intr_frame`]
`uint16_t es` [Member of `struct intr_frame`]
`uint16_t ds` [Member of `struct intr_frame`]
Register values in the interrupted thread saved by `intr_entry()`. The `esp_dummy` value isn't actually used (refer to the description of `PUSHA` in [IA32-v2b] for details).

`uint32_t vec_no` [Member of `struct intr_frame`]
The interrupt vector number, ranging from 0 to 255.

`uint32_t error_code` [Member of `struct intr_frame`]
The "error code" pushed on the stack by the CPU for some internal interrupts.

`void (*eip) (`*void*`)` [Member of `struct intr_frame`]
The address of the next instruction to be executed by the interrupted thread.

`void * esp` [Member of `struct intr_frame`]
The interrupted thread's stack pointer.

`const char * intr_name (`*uint8_t* `vec``)` [Function]
Returns the name of the interrupt numbered *vec*, or `"unknown"` if the interrupt has no registered name.

2.2.3.2 Internal Interrupt Handling

When an internal interrupt occurs, it is because the running kernel thread (or, starting from project 2, the running user process) has caused it. Thus, because it is related to a thread (or process), an internal interrupt is said to happen in a "process context."

In an internal interrupt, it can make sense to examine the `struct intr_frame` passed to the interrupt handler, or even to modify it. When the interrupt returns, modified members in `struct intr_frame` become changes to the thread's registers. We'll use this in project 2 to return values from system call handlers.

There are no special restrictions on what an internal interrupt handler can or can't do. Generally they should run with interrupts enabled, just like other code, and so they can be preempted by other kernel threads. Thus, they do need to synchronize with other threads on shared data and other resources (see Section 2.2.2 [Synchronization], page 16).

`void intr_register_int (`*uint8_t* `vec`*, int* `dpl`*, enum intr_level* `level`*,* [Function]
*intr_handler_func **`handler`*, const char **`name`*)*
Registers *handler* to be called when internal interrupt numbered *vec* is triggered. Names the interrupt *name* for debugging purposes.

If *level* is `INTR_OFF` then handling of further interrupts will be disabled while the interrupt is being processed. Interrupts should normally be turned on during the handling of an internal interrupt.

dpl determines how the interrupt can be invoked. If *dpl* is 0, then the interrupt can be invoked only by kernel threads. Otherwise *dpl* should be 3, which allows user processes to invoke the interrupt as well (this is useful only starting with project 2).

2.2.3.3 External Interrupt Handling

Whereas an internal interrupt runs in the context of the thread that caused it, external interrupts do not have any predictable context. They are asynchronous, so they can be invoked at any time that interrupts have not been disabled. We say that an external interrupt runs in an "interrupt context."

In an external interrupt, the `struct intr_frame` passed to the handler is not very meaningful. It describes the state of the thread or process that was interrupted, but there is no way to predict which one that is. It is possible, although rarely useful, to examine it, but modifying it is a recipe for disaster.

The activities of an external interrupt handler are severely restricted. First, only one external interrupt may be processed at a time, that is, nested external interrupt handling is not supported. This means that external interrupts must be processed with interrupts disabled (see Section 2.2.2.1 [Disabling Interrupts], page 16) and that interrupts may not be enabled at any point during their execution.

Second, an interrupt handler must not call any function that can sleep, which rules out `thread_yield()`, `lock_acquire()`, and many others. This is because external interrupts use space on the stack of the kernel thread that was running at the time the interrupt occurred. If the interrupt handler tried to sleep and that thread resumed, then the two uses of the single stack would interfere, which cannot be allowed.

Because an external interrupt runs with interrupts disabled, it effectively monopolizes the machine and delays all other activities. Therefore, external interrupt handlers should complete as quickly as they can. Any activities that require much CPU time should instead run in a kernel thread, possibly a thread whose activity is triggered by the interrupt using some synchronization primitive.

External interrupts are also special because they are controlled by a pair of devices outside the CPU called *programmable interrupt controllers*, *PICs* for short. When `intr_init()` sets up the CPU's IDT, it also initializes the PICs for interrupt handling. The PICs also must be "acknowledged" at the end of processing for each external interrupt. `intr_handler()` takes care of that by calling `pic_end_of_interrupt()`, which sends the proper signals to the right PIC.

The following additional functions are related to external interrupts.

`void intr_register_ext (`*uint8_t* `vec`, *intr_handler_func* `*handler`, *const char* `*name``)` [Function]

Registers *handler* to be called when external interrupt numbered *vec* is triggered. Names the interrupt *name* for debugging purposes. The handler will run with interrupts disabled.

`bool intr_context (`*void*`)` [Function]

Returns true if we are running in an interrupt context, otherwise false. Mainly used at the beginning of functions that might sleep or that otherwise should not be called from interrupt context, in this form:

```
ASSERT (!intr_context ());
```

`void intr_yield_on_return` (*void*) [Function]
When called in an interrupt context, causes `thread_yield()` to be called just before the interrupt returns. This is used, for example, in the timer interrupt handler to cause a new thread to be scheduled when a thread's time slice expires.

2.2.4 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.

2.2.4.1 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, but this can be changed with a kernel command line option (see [Why PAL_USER?], page 70). 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 will only become important starting with project 3. 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 *n* / 2 pages are free! Single-page requests can't fail due to fragmentation, so it is best to limit, as much as possible, the need for multiple contiguous pages.

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 (see Section E.8 [Debugging Tips], page 97).

Page allocator types and functions are described below.

`enum palloc_flags` [Type]
A set of flags that describe how to allocate pages. These flags may be combined in any combination.

`PAL_ASSERT` [Page Allocator Flag]
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.

`PAL_ZERO` [Page Allocator Flag]
Zero all the bytes in the allocated pages before returning them. If not set, the contents of newly allocated pages are unpredictable.

`PAL_USER` [Page Allocator Flag]
Obtain the pages from the user pool. If not set, pages are allocated from the kernel pool.

`void * palloc_get_page (`*enum palloc_flags* `flags)` [Function]
Obtains and returns a single page, allocating it in the manner specified by *flags*. Returns a null pointer if no pages are free.

`void * palloc_get_multiple (`*enum palloc_flags* `flags`, *size_t* `page_cnt)` [Function]
Obtains *page_cnt* contiguous free pages, allocating them in the manner specified by *flags*, and returns them. Returns a null pointer if no pages are free.

`void palloc_free_page (`*void* `*page)` [Function]
Frees *page*, which must have been obtained using `palloc_get_page()` or `palloc_get_multiple()`.

`void palloc_free_multiple (`*void* `*pages`, *size_t* `page_cnt)` [Function]
Frees the *page_cnt* contiguous pages starting at *pages*. All of the pages must have been obtained using `palloc_get_page()` or `palloc_get_multiple()`.

2.2.4.2 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 of these applies to "small" blocks, those 1 kB or smaller (one fourth of the 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 the smae size.

The second strategy applies to allocating "large" blocks, those larger than 1 kB. 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 will not require a new page from the page allocator at all. 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 kB each.

The interface to the block allocator is through the standard C library functions `malloc()`, `calloc()`, and `free()`.

When a block is freed, all of its bytes are cleared to `0xcc`, as a debugging aid (see Section E.8 [Debugging Tips], page 97).

The block allocator may not be called from interrupt context.

2.3 User Programs Project

The tour for this project has not yet been written.

2.4 Virtual Memory Project

The tour for this project is under construction.

2.4.1 Hash Table

Most implementations of the virtual memory project use a hash table to translate virtual page frames to physical page frames. It is possible to do this translation without adding a new data structure, by modifying the code in '`userprog/pagedir.c`'. However, if you do that you'll need to carefully study and understand section 3.7 in [IA32-v3], and in practice it is probably easier to add a new data structure. You may find other uses for hash tables as well.

Pintos provides a hash table data structure in '`lib/kernel/hash.c`'. To use it you will need to manually include its header file, '`lib/kernel/hash.h`', with `#include <hash.h>`. Intentionally, no code provided with Pintos uses the hash table, which means that you are free to use it as is, modify its implementation for your own purposes, or ignore it, as you wish.

2.4.1.1 Data Types

A hash table is represented by `struct hash`.

`struct hash` [Type]
: Represents an entire hash table. The actual members of `struct hash` are "opaque." That is, code that uses a hash table should not access `struct hash` members directly, nor should it need to. Instead, use hash table functions and macros.

The hash table operates on elements of type `struct hash_elem`.

`struct hash_elem` [Type]
: Embed a `struct hash_elem` member in the structure you want to include in a hash table. Like `struct hash`, `struct hash_elem` is opaque. All functions for operating on hash table elements actually take and return pointers to `struct hash_elem`, not pointers to your hash table's real element type.

You will often need to obtain a `struct hash_elem` given a real element of the hash table, and vice versa. Given a real element of the hash table, obtaining a pointer to its `struct hash_elem` is trivial: take the address of the `struct hash_elem` member. Use the `hash_entry()` macro to go the other direction.

type `* hash_entry (`*struct hash_elem *elem*, *type*, *member*`)` [Macro]
: Returns a pointer to the structure that *elem*, a pointer to a `struct hash_elem`, is embedded within. You must provide *type*, the name of the structure that *elem* is inside, and *member*, the name of the member in *type* that *elem* points to.

  For example, suppose `h` is a `struct hash_elem *` variable that points to a `struct thread` member (of type `struct hash_elem`) named `h_elem`. Then, `hash_entry (h, struct thread, h_elem)` yields the address of the `struct thread` that `h` points within.

Each hash table element must contain a key, that is, data that identifies and distinguishes elements in the hash table. Every element in a hash table at a given time must have a unique key. (Elements may also contain non-key data that need not be unique.) While an element is in a hash table, its key data must not be changed. For each hash table, you must write two functions that act on keys: a hash function and a comparison function. These functions must match the following prototypes:

`unsigned hash_hash_func (const struct hash_elem *element, void *aux)` [Type]

Returns a hash of *element*'s data, as a value anywhere in the range of `unsigned int`. The hash of an element should be a pseudo-random function of the element's key. It must not depend on non-key data in the element or on any non-constant data other than the key. Pintos provides the following functions as a suitable basis for hash functions.

`unsigned hash_bytes (const void *buf, size_t *size)` [Function]

Returns a hash of the *size* bytes starting at *buf*. The implementation is the general-purpose Fowler-Noll-Vo hash for 32-bit words.

`unsigned hash_string (const char *s)` [Function]

Returns a hash of null-terminated string *s*.

`unsigned hash_int (int i)` [Function]

Returns a hash of integer *i*.

If your key is a single piece of data of an appropriate type, it is sensible for your hash function to directly return the output of one of these functions. For multiple pieces of data, you may wish to combine the output of more than one call to them using, e.g., the '^' operator. Finally, you may entirely ignore these functions and write your own hash function from scratch, but remember that your goal is to build an operating system kernel, not to design a hash function.

See Section 2.4.1.6 [Hash Auxiliary Data], page 32, for an explanation of *aux*.

`bool hash_less_func (const struct hash_elem *a, const struct hash_elem *b, void *aux)` [Type]

Compares the keys stored in elements *a* and *b*. Returns true if *a* is less than *b*, false if *a* is greater than or equal to *b*.

If two elements compare equal, then they must hash to equal values.

See Section 2.4.1.6 [Hash Auxiliary Data], page 32, for an explanation of *aux*.

A few functions that act on hashes accept a pointer to a third kind of function as an argument:

`void hash_action_func (struct hash_elem *element, void *aux)` [Type]

Performs some kind of action, chosen by the caller, on *element*.

See Section 2.4.1.6 [Hash Auxiliary Data], page 32, for an explanation of *aux*.

2.4.1.2 Basic Functions

These functions create and destroy hash tables and obtain basic information about their contents.

`bool hash_init` (*struct hash *`hash`, hash_hash_func *`hash_func`, hash_less_func *`less_func`, void *`aux`*) [Function]

Initializes *hash* as a hash table using *hash_func* as hash function, *less_func* as comparison function, and *aux* as auxiliary data. Returns true if successful, false on failure. `hash_init()` calls `malloc()` and fails if memory cannot be allocated.

See Section 2.4.1.6 [Hash Auxiliary Data], page 32, for an explanation of *aux*, which is most often a null pointer.

`void hash_clear` (*struct hash *`hash`, hash_action_func *`action`*) [Function]

Removes all the elements from *hash*, which must have been previously initialized with `hash_init()`.

If *action* is non-null, then it is called once for each element in the hash table, which gives the caller an opportunity to deallocate any memory or other resources used by the element. For example, if the hash table elements are dynamically allocated using `malloc()`, then *action* could `free()` the element. This is safe because `hash_clear()` will not access the memory in a given hash element after calling *action* on it. However, *action* must not call any function that may modify the hash table, such as `hash_insert()` or `hash_delete()`.

`void hash_destroy` (*struct hash *`hash`, hash_action_func *`action`*) [Function]

If *action* is non-null, calls it for each element in the hash, with the same semantics as a call to `hash_clear()`. Then, frees the memory held by *hash*. Afterward, *hash* must not be passed to any hash table function, absent an intervening call to `hash_init()`.

`size_t hash_size` (*struct hash *`hash`*) [Function]

Returns the number of elements currently stored in *hash*.

`bool hash_empty` (*struct hash *`hash`*) [Function]

Returns true if *hash* currently contains no elements, false if *hash* contains at least one element.

2.4.1.3 Search Functions

Each of these functions searches a hash table for an element that compares equal to one provided. Based on the success of the search, they perform some action, such as inserting a new element into the hash table, or simply return the result of the search.

`struct hash_elem * hash_insert` (*struct hash *`hash`, struct hash_elem *`element`*) [Function]

Searches *hash* for an element equal to *element*. If none is found, inserts *element* into *hash* and returns a null pointer. If the table already contains an element equal to *element*, returns the existing element without modifying *hash*.

`struct hash_elem * hash_replace` (*struct hash *`hash`, struct hash_elem *`element`*) [Function]

Inserts *element* into *hash*. Any element equal to *element* already in *hash* is removed. Returns the element removed, or a null pointer if *hash* did not contain an element equal to *element*.

The caller is responsible for deallocating any resources associated with the element returned, as appropriate. For example, if the hash table elements are dynamically allocated using `malloc()`, then the caller must `free()` the element after it is no longer needed.

The element passed to the following functions is only used for hashing and comparison purposes. It is never actually inserted into the hash table. Thus, only the key data in the element need be initialized, and other data in the element will not be used. It often makes sense to declare an instance of the element type as a local variable, initialize the key data, and then pass the address of its `struct hash_elem` to `hash_find()` or `hash_delete()`. See Section 2.4.1.5 [Hash Table Example], page 31, for an example. (Large structures should not be allocated as local variables. See Section 2.2.1.1 [struct thread], page 11, for more information.)

`struct hash_elem * hash_find` (*struct hash *`hash`, struct hash_elem *`element`*) [Function]

Searches *hash* for an element equal to *element*. Returns the element found, if any, or a null pointer otherwise.

`struct hash_elem * hash_delete` (*struct hash *`hash`, struct hash_elem *`element`*) [Function]

Searches *hash* for an element equal to *element*. If one is found, it is removed from *hash* and returned. Otherwise, a null pointer is returned and *hash* is unchanged.

The caller is responsible for deallocating any resources associated with the element returned, as appropriate. For example, if the hash table elements are dynamically allocated using `malloc()`, then the caller must `free()` the element after it is no longer needed.

2.4.1.4 Iteration Functions

These functions allow iterating through the elements in a hash table. Two interfaces are supplied. The first requires writing and supplying a *hash_action_func* to act on each element (see Section 2.4.1.1 [Hash Data Types], page 27).

`void hash_apply` (*struct hash *`hash`, hash_action_func *`action`*) [Function]

Calls *action* once for each element in *hash*, in arbitrary order. *action* must not call any function that may modify the hash table, such as `hash_insert()` or `hash_delete()`. *action* must not modify key data in elements, although it may modify any other data.

The second interface is based on an "iterator" data type. Idiomatically, iterators are used as follows:

```
struct hash_iterator i;

hash_first (&i, h);
```

```
while (hash_next (&i))
  {
    struct foo *f = hash_entry (hash_cur (&i), struct foo, elem);
    ...do something with f...
  }
```

`struct hash_iterator` [Type]
Represents a position within a hash table. Calling any function that may modify a hash table, such as `hash_insert()` or `hash_delete()`, invalidates all iterators within that hash table.

Like `struct hash` and `struct hash_elem`, `struct hash_elem` is opaque.

`void hash_first` (*struct hash_iterator* **iterator*, *struct hash* **hash*) [Function]
Initializes *iterator* to just before the first element in *hash*.

`struct hash_elem * hash_next` (*struct hash_iterator* **iterator*) [Function]
Advances *iterator* to the next element in *hash*, and returns that element. Returns a null pointer if no elements remain. After `hash_next()` returns null for *iterator*, calling it again yields undefined behavior.

`struct hash_elem * hash_cur` (*struct hash_iterator* **iterator*) [Function]
Returns the value most recently returned by `hash_next()` for *iterator*. Yields undefined behavior after `hash_first()` has been called on *iterator* but before `hash_next()` has been called for the first time.

2.4.1.5 Hash Table Example

Suppose you have a structure, called `struct page`, that you want to put into a hash table. First, define `struct page` to include a `struct hash_elem` member:

```
struct page
  {
    struct hash_elem hash_elem; /* Hash table element. */
    void *addr;                 /* Virtual address. */
    /* ...other members... */
  };
```

We write a hash function and a comparison function using *addr* as the key. A pointer can be hashed based on its bytes, and the '`<`' operator works fine for comparing pointers:

```
/* Returns a hash value for page p. */
unsigned
page_hash (const struct hash_elem *p_, void *aux UNUSED)
{
  const struct page *p = hash_entry (p_, struct page, hash_elem);
  return hash_bytes (&p->addr, sizeof p->addr);
}

/* Returns true if page a precedes page b. */
bool
page_less (const struct hash_elem *a_, const struct hash_elem *b_,
```

```
            void *aux UNUSED)
{
  const struct page *a = hash_entry (a_, struct page, hash_elem);
  const struct page *b = hash_entry (b_, struct page, hash_elem);

  return a->addr < b->addr;
}
```

(The use of `UNUSED` in these functions' prototypes suppresses a warning that *aux* is unused. See Section E.3 [Function and Parameter Attributes], page 92, for information about `UNUSED`. See Section 2.4.1.6 [Hash Auxiliary Data], page 32, for an explanation of *aux*.)

Then, we can create a hash table like this:

```
struct hash pages;

hash_init (&pages, page_hash, page_less, NULL);
```

Now we can manipulate the hash table we've created. If *p* is a pointer to a `struct page`, we can insert it into the hash table with:

```
hash_insert (&pages, &p->hash_elem);
```

If there's a chance that *pages* might already contain a page with the same *addr*, then we should check `hash_insert()`'s return value.

To search for an element in the hash table, use `hash_find()`. This takes a little setup, because `hash_find()` takes an element to compare against. Here's a function that will find and return a page based on a virtual address, assuming that *pages* is defined at file scope:

```
/* Returns the page containing the given virtual address,
   or a null pointer if no such page exists. */
struct page *
page_lookup (const void *address)
{
  struct page p;
  struct hash_elem *e;

  p.addr = address;
  e = hash_find (&pages, &p.hash_elem);
  return e != NULL ? hash_entry (e, struct page, hash_elem) : NULL;
}
```

`struct page` is allocated as a local variable here on the assumption that it is fairly small. Large structures should not be allocated as local variables. See Section 2.2.1.1 [struct thread], page 11, for more information.

A similar function could delete a page by address using `hash_delete()`.

2.4.1.6 Auxiliary Data

In simple cases like the example above, there's no need for the *aux* parameters. In these cases, just pass a null pointer to `hash_init()` for *aux* and ignore the values passed to the hash function and comparison functions. (You'll get a compiler warning if you don't use the *aux* parameter, but you can turn that off with the `UNUSED` macro, as shown in the example, or you can just ignore it.)

aux is useful when you have some property of the data in the hash table that's both constant and needed for hashing or comparisons, but which is not stored in the data items themselves. For example, if the items in a hash table contain fixed-length strings, but the items themselves don't indicate what that fixed length is, you could pass the length as an *aux* parameter.

2.4.1.7 Synchronization

The hash table does not do any internal synchronization. It is the caller's responsibility to synchronize calls to hash table functions. In general, any number of functions that examine but do not modify the hash table, such as `hash_find()` or `hash_next()`, may execute simultaneously. However, these function cannot safely execute at the same time as any function that may modify a given hash table, such as `hash_insert()` or `hash_delete()`, nor may more than one function that can modify a given hash table execute safely at once.

It is also the caller's responsibility to synchronize access to data in hash table elements. How to synchronize access to this data depends on how it is designed and organized, as with any other data structure.

2.5 File Systems Project

The tour for this project has not yet been written.

3 Project 1: Threads

In this assignment, we give you a minimally functional thread system. Your job is to extend the functionality of this system to gain a better understanding of synchronization problems.

You will be working primarily in the '`threads`' directory for this assignment, with some work in the '`devices`' directory on the side. Compilation should be done in the '`threads`' directory.

Before you read the description of this project, you should read all of the following sections: Chapter 1 [Introduction], page 1, Appendix C [Coding Standards], page 86, Appendix E [Debugging Tools], page 92, and Appendix F [Development Tools], page 98. You should at least skim the material in Section 2.2 [Threads Tour], page 11 and especially Section 2.2.2 [Synchronization], page 16. To complete this project you will also need to read Appendix B [4.4BSD Scheduler], page 81.

3.1 Background

3.1.1 Understanding Threads

The first step is to read and understand the code for the initial thread system. Pintos already implements thread creation and thread completion, a simple scheduler to switch between threads, and synchronization primitives (semaphores, locks, condition variables, and memory barriers).

Some of this code might seem slightly mysterious. If you haven't already compiled and run the base system, as described in the introduction (see Chapter 1 [Introduction], page 1), you should do so now. You can read through parts of the source code to see what's going on. If you like, you can add calls to `printf()` almost anywhere, then recompile and run to see what happens and in what order. You can also run the kernel in a debugger and set breakpoints at interesting spots, single-step through code and examine data, and so on.

When a thread is created, you are creating a new context to be scheduled. You provide a function to be run in this context as an argument to `thread_create()`. The first time the thread is scheduled and runs, it starts from the beginning of that function and executes in that context. When the function returns, the thread terminates. Each thread, therefore, acts like a mini-program running inside Pintos, with the function passed to `thread_create()` acting like `main()`.

At any given time, exactly one thread runs and the rest, if any, become inactive. The scheduler decides which thread to run next. (If no thread is ready to run at any given time, then the special "idle" thread, implemented in `idle()`, runs.) Synchronization primitives can force context switches when one thread needs to wait for another thread to do something.

The mechanics of a context switch are in '`threads/switch.S`', which is 80*x*86 assembly code. (You don't have to understand it.) It saves the state of the currently running thread and restores the state of the thread we're switching to.

Using the `gdb` debugger, slowly trace through a context switch to see what happens (see Section E.5 [gdb], page 95). You can set a breakpoint on `schedule()` to start out, and

then single-step from there.[1] Be sure to keep track of each thread's address and state, and what procedures are on the call stack for each thread. You will notice that when one thread calls `switch_threads()`, another thread starts running, and the first thing the new thread does is to return from `switch_threads()`. You will understand the thread system once you understand why and how the `switch_threads()` that gets called is different from the `switch_threads()` that returns. See Section 2.2.1.3 [Thread Switching], page 15, for more information.

Warning: In Pintos, each thread is assigned a small, fixed-size execution stack just under 4 kB in size. The kernel tries to detect stack overflow, but it cannot do so perfectly. You may cause bizarre problems, such as mysterious kernel panics, if you declare large data structures as non-static local variables, e.g. '`int buf[1000];`'. Alternatives to stack allocation include the page allocator and the block allocator (see Section 2.2.4 [Memory Allocation], page 25).

3.1.2 Source Files

Here is a brief overview of the files in the '`threads`' directory. You will not need to modify most of this code, but the hope is that presenting this overview will give you a start on what code to look at.

'`loader.S`'
'`loader.h`'
: The kernel loader. Assembles to 512 bytes of code and data that the PC BIOS loads into memory and which in turn loads the kernel into memory, does basic processor initialization, and jumps to the beginning of the kernel. See Section 2.1.1 [Pintos Loader], page 9, for details. You should not need to look at this code or modify it.

'`kernel.lds.S`'
: The linker script used to link the kernel. Sets the load address of the kernel and arranges for '`start.S`' to be at the very beginning of the kernel image. See Section 2.1.1 [Pintos Loader], page 9, for details. Again, you should not need to look at this code or modify it, but it's here in case you're curious.

'`start.S`' Jumps to `main()`.

'`init.c`'
'`init.h`' Kernel initialization, including `main()`, the kernel's "main program." You should look over `main()` at least to see what gets initialized. You might want to add your own initialization code here. See Section 2.1.2 [Kernel Initialization], page 10, for details.

'`thread.c`'
'`thread.h`'
: Basic thread support. Much of your work will take place in these files. '`thread.h`' defines `struct thread`, which you are likely to modify in all four projects. See Section 2.2.1.1 [struct thread], page 11 and Section 2.2.1 [Thread Support], page 11 for more information.

[1] `gdb` might tell you that `schedule()` doesn't exist, which is arguably a `gdb` bug. You can work around this by setting the breakpoint by filename and line number, e.g. `break thread.c:`*ln* where *ln* is the line number of the first declaration in `schedule()`.

‘`switch.S`’
‘`switch.h`’
: Assembly language routine for switching threads. Already discussed above. See Section 2.2.1.2 [Thread Functions], page 13, for more information.

‘`palloc.c`’
‘`palloc.h`’
: Page allocator, which hands out system memory in multiples of 4 kB pages. See Section 2.2.4.1 [Page Allocator], page 25, for more information.

‘`malloc.c`’
‘`malloc.h`’
: A simple implementation of `malloc()` and `free()` for the kernel. See Section 2.2.4.2 [Block Allocator], page 26, for more information.

‘`interrupt.c`’
‘`interrupt.h`’
: Basic interrupt handling and functions for turning interrupts on and off. See Section 2.2.3 [Interrupt Handling], page 21, for more information.

‘`intr-stubs.S`’
‘`intr-stubs.h`’
: Assembly code for low-level interrupt handling. See Section 2.2.3.1 [Interrupt Infrastructure], page 22, for more information.

‘`synch.c`’
‘`synch.h`’ Basic synchronization primitives: semaphores, locks, condition variables, and memory barriers. You will need to use these for synchronization in all four projects. See Section 2.2.2 [Synchronization], page 16, for more information.

‘`io.h`’ Functions for I/O port access. This is mostly used by source code in the ‘`devices`’ directory that you won’t have to touch.

‘`mmu.h`’ Functions and macros related to memory management, including page directories and page tables. This will be more important to you in project 3. For now, you can ignore it.

‘`flags.h`’ Macros that define a few bits in the 80*x*86 “flags” register. Probably of no interest. See [IA32-v1], section 3.4.3, for more information.

3.1.2.1 ‘`devices`’ code

The basic threaded kernel also includes these files in the ‘`devices`’ directory:

‘`timer.c`’
‘`timer.h`’ System timer that ticks, by default, 100 times per second. You will modify this code in this project.

‘`vga.c`’
‘`vga.h`’ VGA display driver. Responsible for writing text to the screen. You should have no need to look at this code. `printf()` calls into the VGA display driver for you, so there’s little reason to call this code yourself.

‘`serial.c`’
‘`serial.h`’
: Serial port driver. Again, `printf()` calls this code for you, so you don’t need to do so yourself. Feel free to look through it if you’re curious.

‘`disk.c`’
‘`disk.h`’ Supports reading and writing sectors on up to 4 IDE disks. This won’t actually be used until project 2.

‘`intq.c`’
‘`intq.h`’ Interrupt queue, for managing a circular queue that both kernel threads and interrupt handlers want to access. Used by the keyboard and serial drivers.

3.1.2.2 ‘`lib`’ files

Finally, ‘`lib`’ and ‘`lib/kernel`’ contain useful library routines. (‘`lib/user`’ will be used by user programs, starting in project 2, but it is not part of the kernel.) Here’s a few more details:

‘`ctype.h`’
‘`inttypes.h`’
‘`limits.h`’
‘`stdarg.h`’
‘`stdbool.h`’
‘`stddef.h`’
‘`stdint.h`’
‘`stdio.c`’
‘`stdio.h`’
‘`stdlib.c`’
‘`stdlib.h`’
‘`string.c`’
‘`string.h`’
: A subset of the standard C library. See Section C.2 [C99], page 86, for information on a few recently introduced pieces of the C library that you might not have encountered before. See Section C.3 [Unsafe String Functions], page 87, for information on what’s been intentionally left out for safety.

‘`debug.c`’
‘`debug.h`’ Functions and macros to aid debugging. See Appendix E [Debugging Tools], page 92, for more information.

‘`random.c`’
‘`random.h`’
: Pseudo-random number generator.

‘`round.h`’ Macros for rounding.

‘`syscall-nr.h`’
: System call numbers. Not used until project 2.

‘`kernel/list.c`’
‘`kernel/list.h`’
: Doubly linked list implementation. Used all over the Pintos code, and you’ll probably want to use it a few places yourself in project 1.

‘`kernel/bitmap.c`’
‘`kernel/bitmap.h`’
: Bitmap implementation. You can use this in your code if you like, but you probably won’t have any need for it in project 1.

‘`kernel/hash.c`’
‘`kernel/hash.h`’
: Hash table implementation. Likely to come in handy for project 3.

‘`kernel/console.c`’
‘`kernel/console.h`’
‘`kernel/stdio.h`’
: Implements `printf()` and a few other functions.

3.1.3 Synchronization

Proper synchronization is an important part of the solutions to these problems. Any synchronization problem can be easily solved by turning interrupts off: while interrupts are off, there is no concurrency, so there’s no possibility for race conditions. Therefore, it’s tempting to solve all synchronization problems this way, but **don’t**. Instead, use semaphores, locks, and condition variables to solve the bulk of your synchronization problems. Read the tour section on synchronization (see Section 2.2.2 [Synchronization], page 16) or the comments in ‘`threads/synch.c`’ if you’re unsure what synchronization primitives may be used in what situations.

In the Pintos projects, the only class of problem best solved by disabling interrupts is coordinating data shared between a kernel thread and an interrupt handler. Because interrupt handlers can’t sleep, they can’t acquire locks. This means that data shared between kernel threads and an interrupt handler must be protected within a kernel thread by turning off interrupts.

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

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

You may need to add or modify code where interrupts are already disabled, such as in `sema_up()` or `sema_down()`. You should still try to keep this code as short as you can.

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.

3.1.4 Development Suggestions

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 a source code control system such as CVS (see Section F.2 [CVS], page 98). This is less likely to produce surprises, because everyone can see everyone else's code as it is written, instead of just when it is finished. These systems also make it possible to review changes and, when a change introduces a bug, drop back to working versions of code.

You should expect to run into bugs that you simply don't understand while working on this and subsequent projects. When you do, reread the appendix on debugging tools, which is filled with useful debugging tips that should help you to get back up to speed (see Appendix E [Debugging Tools], page 92). Be sure to read the section on backtraces (see Section E.4 [Backtraces], page 93), which will help you to get the most out of every kernel panic or assertion failure.

3.2 Requirements

3.2.1 Design Document

Before you turn in your project, you must copy the project 1 design document template into your source tree under the name '`pintos/src/threads/DESIGNDOC`' and fill it in. We recommend that you read the design document template before you start working on the project. See Appendix D [Project Documentation], page 89, for a sample design document that goes along with a fictitious project.

3.2.2 Alarm Clock

Reimplement `timer_sleep()`, defined in '`devices/timer.c`'. Although a working implementation is provided, it "busy waits," that is, it spins in a loop checking the current time and calling `thread_yield()` until enough time has gone by. Reimplement it to avoid busy waiting.

`void timer_sleep (`*int64_t* ***ticks***`)` [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 amount of time.

`timer_sleep()` is useful for threads that operate in real-time, e.g. for blinking the cursor once per second.

The argument to `timer_sleep()` is expressed in timer ticks, not in milliseconds or any another unit. There are `TIMER_FREQ` timer ticks per second, where `TIMER_FREQ` is

a macro defined in `devices/timer.h`. The default value is 100. We don't recommend changing this value, because any change is likely to cause many of the tests to fail.

Separate functions `timer_msleep()`, `timer_usleep()`, and `timer_nsleep()` do exist for sleeping a specific number of milliseconds, microseconds, or nanoseconds, respectively, but these will call `timer_sleep()` automatically when necessary. You do not need to modify them.

If your delays seem too short or too long, reread the explanation of the '`-r`' option to `pintos` (see Section 1.1.4 [Debugging versus Testing], page 4).

The alarm clock implementation is not needed for later projects, although it could be useful for project 4.

3.2.3 Priority Scheduling

Implement priority scheduling in Pintos. When a thread is added to the ready list that has a higher priority than the currently running thread, the current thread should immediately yield the processor to the new thread. Similarly, when threads are waiting for a lock, semaphore, or condition variable, the highest priority waiting thread should be woken up first. A thread may raise or lower its own priority at any time, but lowering its priority such that it no longer has the highest priority must cause it to immediately yield the CPU.

Thread priorities range from `PRI_MIN` (0) to `PRI_MAX` (63). Lower numbers correspond to *higher* priorities, so that priority 0 is the highest priority and priority 63 is the lowest. The initial thread priority is passed as an argument to `thread_create()`. If there's no reason to choose another priority, use `PRI_DEFAULT` (31). The `PRI_` macros are defined in '`threads/thread.h`', and you should not change their values.

One issue with priority scheduling is "priority inversion". Consider high, medium, and low priority threads H, M, and L, respectively. If H needs to wait for L (for instance, for a lock held by L), and M is on the ready list, then H will never get the CPU because the low priority thread will not get any CPU time. A partial fix for this problem is for H to "donate" its priority to L while L is holding the lock, then recall the donation once L releases (and thus H acquires) the lock.

Implement priority donation. You will need to account for all different situations in which priority donation is required. Be sure to handle multiple donations, in which multiple priorities are donated to a single thread. You must also handle nested donation: if H is waiting on a lock that M holds and M is waiting on a lock that L holds, then both M and L should be boosted to H's priority. If necessary, you may impose a reasonable limit on depth of nested priority donation, such as 8 levels.

You must implement priority donation for locks. You need not implement priority donation for semaphores or condition variables (but you are welcome to do so). You do need to implement priority scheduling in all cases.

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

`void thread_set_priority (`*int* *`new_priority`*`)` [Function]

Sets the current thread's priority to *new_priority*. If the current thread no longer has the highest priority, yields.

`int thread_get_priority` (*void*) [Function]
: Returns the current thread's priority. In the presence of priority donation, returns the higher (donated) priority.

You need not provide any interface to allow a thread to directly modify other threads' priorities.

The priority scheduler is not used in any later project.

3.2.4 Advanced Scheduler

Implement a multilevel feedback queue scheduler similar to the 4.4BSD scheduler to reduce the average response time for running jobs on your system. See Appendix B [4.4BSD Scheduler], page 81, for detailed requirements.

Like the priority scheduler, the advanced scheduler chooses the thread to run based on priorities. However, the advanced scheduler does not do priority donation. Thus, we recommend that you have the priority scheduler working, except possibly for priority donation, before you start work on the advanced scheduler.

You must write your code to allow us to choose a scheduling algorithm policy at Pintos startup time. By default, the priority scheduler must be active, but we must be able to choose the 4.4BSD scheduler with the '`-mlfqs`' kernel option. Passing this option sets `enable_mlfqs`, declared in '`threads/init.h`', to true.

When the 4.4BSD scheduler is enabled, threads no longer directly control their own priorities. The *priority* argument to `thread_create()` should be ignored, as well as any calls to `thread_set_priority()`, and `thread_get_priority()` should return the thread's current priority as set by the scheduler.

The advanced scheduler is not used in any later project.

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.

```
devices/timer.c       |   42 +++++-
threads/fixed-point.h |  120 ++++++++++++++++++
threads/synch.c       |   88 ++++++++++++-
threads/thread.c      |  196 ++++++++++++++++++++++++++++----
threads/thread.h      |   23 +++
5 files changed, 440 insertions(+), 29 deletions(-)
```

How do I update the '`Makefile`'s when I add a new source file?

To add a '`.c`' file, edit the top-level '`Makefile.build`'. Add the new file to variable '*dir*`_SRC`', where *dir* is the directory where you added the file. For

this project, that means you should add it to `threads_SRC` or `devices_SRC`. Then run `make`. If your new file doesn't get compiled, run `make clean` and then try again.

When you modify the top-level '`Makefile.build`' and re-run `make`, the modified version should be automatically copied to '`threads/build/Makefile`'. The converse is not true, so any changes will be lost the next time you run `make clean` from the '`threads`' directory. Unless your changes are truly temporary, you should prefer to edit '`Makefile.build`'.

A new '`.h`' file does not require editing the '`Makefile`'s.

What does `warning: no previous prototype for '`*`func`*`'` mean?

It means that you defined a non-`static` function without preceding it by a prototype. Because non-`static` functions are intended for use by other '`.c`' files, for safety they should be prototyped in a header file included before their definition. To fix the problem, add a prototype in a header file that you include, or, if the function isn't actually used by other '`.c`' files, make it `static`.

What is the interval between timer interrupts?

Timer interrupts occur `TIMER_FREQ` times per second. You can adjust this value by editing '`devices/timer.h`'. The default is 100 Hz.

We don't recommend changing this value, because any changes are likely to cause many of the tests to fail.

How long is a time slice?

There are `TIME_SLICE` ticks per time slice. This macro is declared in '`threads/thread.c`'. The default is 4 ticks.

We don't recommend changing this value, because any changes are likely to cause many of the tests to fail.

How do I run the tests?

See Section 1.2.1 [Testing], page 5.

Why do I get a test failure in `pass()`?

You are probably looking at a backtrace that looks something like this:

```
0xc0108810: debug_panic (../../lib/kernel/debug.c:32)
0xc010a99f: pass (../../tests/threads/tests.c:93)
0xc010bdd3: test_mlfqs_load_1 (../../tests/threads/mlfqs-load-1.c:33)
0xc010a8cf: run_test (../../tests/threads/tests.c:51)
0xc0100452: run_task (../../threads/init.c:283)
0xc0100536: run_actions (../../threads/init.c:333)
0xc01000bb: main (../../threads/init.c:137)
```

This is just confusing output from the `backtrace` program. It does not actually mean that `pass()` called `debug_panic()`. In fact, `fail()` called `debug_panic()` (via the `PANIC()` macro). GCC knows that `debug_panic()` does not return, because it is declared `NO_RETURN` (see Section E.3 [Function and Parameter Attributes], page 92), so it doesn't include any code in `pass()` to take control when `debug_panic()` returns. This means that the return address on the stack looks like it is at the beginning of the function that happens to follow `fail()` in memory, which in this case happens to be `pass()`.

See Section E.4 [Backtraces], page 93, for more information.

3.3.1 Alarm Clock FAQ

Do I need to account for timer values overflowing?

Don't worry about the possibility of timer values overflowing. Timer values are expressed as signed 64-bit numbers, which at 100 ticks per second should be good for almost 2,924,712,087 years. By then, we expect Pintos to have been phased out of the CS 3204 curriculum.

3.3.2 Priority Scheduling FAQ

Doesn't priority scheduling lead to starvation?

Yes, strict priority scheduling can lead to starvation because a thread will not run if any higher-priority thread is runnable. The advanced scheduler introduces a mechanism for dynamically changing thread priorities.

Strict priority scheduling is valuable in real-time systems because it offers the programmer more control over which jobs get processing time. High priorities are generally reserved for time-critical tasks. It's not "fair," but it addresses other concerns not applicable to a general-purpose operating system.

What thread should run after a lock has been released?

When a lock is released, the highest priority thread waiting for that lock should be unblocked and put on the list of ready threads. The scheduler should then run the highest priority thread on the ready list.

If the highest-priority thread yields, does it continue running?

Yes. As long as there is a single highest-priority thread, it continues running until it blocks or finishes, even if it calls `thread_yield()`. If multiple threads have the same highest priority, `thread_yield()` should switch among them in "round robin" order.

What happens to the priority of a donating thread?

Priority donation only changes the priority of the donee thread. The donor thread's priority is unchanged. Priority donation is not additive: if thread A (with priority 3) donates to thread B (with priority 5), then B's new priority is 3, not 8.

Can a thread's priority change while it is on the ready queue?

Yes. Consider this case: low-priority thread L holds a lock that high-priority thread H wants, so H donates its priority to L. L releases the lock and thus loses the CPU and is moved to the ready queue. Now L's old priority is restored while it is in the ready queue.

Can a thread added to the ready list preempt the processor?

Yes. If a thread added to the ready list has higher priority than the running thread, the correct behavior is to immediately yield the processor. It is not acceptable to wait for the next timer interrupt. The highest priority thread should run as soon as it is runnable, preempting whatever thread is currently running.

How does `thread_set_priority()` affect a thread receiving donations?

It should do something sensible, but no particular behavior is required. None of the test cases call `thread_set_priority()` from a thread while it is receiving a priority donation.

Calling `printf()` in `sema_up()` or `sema_down()` reboots!

Yes. These functions are called before `printf()` is ready to go. You could add a global flag initialized to false and set it to true just before the first `printf()` in `main()`. Then modify `printf()` itself to return immediately if the flag isn't set.

3.3.3 Advanced Scheduler FAQ

How does priority donation interact with the advanced scheduler?

It doesn't have to. We won't test priority donation and the advanced scheduler at the same time.

Can I use one queue instead of 64 queues?

Yes, that's fine. It's easiest to describe the algorithm in terms of 64 separate queues, but that doesn't mean you have to implement it that way.

If you use a single queue, it should probably be sorted.

4 Project 2: User Programs

Now that you've worked with Pintos and are becoming familiar with its infrastructure and thread package, it's time to start working on the parts of the system that allow running user programs. The base code already supports loading and running user programs, but no I/O or interactivity is possible. In this project, you will enable programs to interact with the OS via system calls.

You will be working out of the '`userprog`' directory for this assignment. However, you will also be interacting with almost every other part of the code for this assignment. We will describe the relevant parts below.

You can build project 2 on top of your project 1 submission or you can start with a fresh copy. No code from project 1 is required for this assignment. The "alarm clock" functionality may be useful in projects 3 and 4, but it is not strictly required.

You might find it useful to go back and reread how to run the tests (see Section 1.2.1 [Testing], page 5). In particular, the tests for project 2 and later projects will probably run faster if you use the qemu emulator, e.g. via `make check PINTOSOPTS='--qemu'`. The qemu emulator is available only on the Linux machines.

4.1 Background

Up to now, all of the code you have run under Pintos has been part of the operating system kernel. This means, for example, that all the test code from the last assignment ran as part of the kernel, with full access to privileged parts of the system. Once we start running user programs on top of the operating system, this is no longer true. This project deals with consequences of the change.

We allow more than one process to run at a time. Each process has one thread (multi-threaded processes are not supported). User programs are written under the illusion that they have the entire machine. This means that when you load and run multiple processes at a time, you must manage memory, scheduling, and other state correctly to maintain this illusion.

In the previous project, we compiled our test code directly into your kernel, so we had to require certain specific function interfaces within the kernel. From now on, we will test your operating system by running user programs. This gives you much greater freedom. You must make sure that the user program interface meets the specifications described here, but given that constraint you are free to restructure or rewrite kernel code however you wish.

4.1.1 Source Files

The easiest way to get an overview of the programming you will be doing is to simply go over each part you'll be working with. In '`userprog`', you'll find a small number of files, but here is where the bulk of your work will be:

'`process.c`'
'`process.h`'
: Loads ELF binaries and starts processes.

‘`pagedir.c`’
‘`pagedir.h`’
: A simple manager for 80x86 page directories and page tables. Although you probably won't want to modify this code for this project, you may want to call some of its functions.

‘`syscall.c`’
‘`syscall.h`’
: Whenever a user process wants to access some kernel functionality, it invokes a system call. This is a skeleton system call handler. Currently, it just prints a message and terminates the user process. In part 2 of this project you will add code to do everything else needed by system calls.

‘`exception.c`’
‘`exception.h`’
: When a user process performs a privileged or prohibited operation, it traps into the kernel as an “exception” or “fault.”[1] These files handle exceptions. Currently all exceptions simply print a message and terminate the process. Some, but not all, solutions to project 2 require modifying `page_fault()` in this file.

‘`gdt.c`’
‘`gdt.h`’
: The 80x86 is a segmented architecture. The Global Descriptor Table (GDT) is a table that describes the segments in use. These files set up the GDT. **You should not need to modify these files for any of the projects.** You can read the code if you're interested in how the GDT works.

‘`tss.c`’
‘`tss.h`’
: The Task-State Segment (TSS) is used for 80x86 architectural task switching. Pintos uses the TSS only for switching stacks when a user process enters an interrupt handler, as does Linux. **You should not need to modify these files for any of the projects.** You can read the code if you're interested in how the TSS works.

4.1.2 Using the File System

You will need to use some file system code for this project. First, user programs are loaded from the file system. Second, many of the system calls you must implement deal with the file system. However, the focus of this project is not on the file system code, so we have provided a simple file system in the ‘`filesys`’ directory. You will want to look over the ‘`filesys.h`’ and ‘`file.h`’ interfaces to understand how to use the file system, and especially its many limitations. **You should not modify the file system code for this project.** Proper use of the file system routines now will make life much easier for project 4, when you improve the file system implementation. Until then, you will have to put up with the following limitations:

- No synchronization. Concurrent accesses will interfere with one another. You should use a global lock 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.

[1] We will treat these terms as synonymous. There is no standard distinction between them, although Intel processor manuals define them slightly differently on 80x86.

- File data is allocated as a single extent, that is, 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.

One important feature is included:

- Unix-like semantics for `filesys_remove()` are implemented. That is, if a file is open when it is removed, its blocks are not deallocated and it may still be accessed by any threads that have it open until the last one closes it. See [Removing an Open File], page 57, for more information.

You need to be able to create simulated disks. The `pintos-mkdisk` program provides this functionality. From the '`userprog/build`' directory, execute `pintos-mkdisk fs.dsk 2`. This command creates a 2 MB simulated disk named '`fs.dsk`'. Then format the disk by passing '`-f -q`' on the kernel's command line: `pintos -f -q`. The '`-f`' option causes the disk 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`'.

Incidentally, these commands work by passing special commands `put` and `get` on the kernel's command line and copying to and from a special simulated "scratch" disk. If you're very curious, you can look at the `pintos` program as well as '`filesys/fsutil.c`' to learn the implementation details.

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

```
pintos-mkdisk fs.dsk 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 fs.dsk 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 `--fs-disk=`*`n`* option creates a temporary disk 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 --fs-disk=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.

4.1.3 How User Programs Work

Pintos can run normal C programs. In fact, Pintos can run any program you want, as long as it's compiled into the proper file format and uses 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 compile your own programs as well.

Pintos can load *ELF* executables with the loader provided for you in '`userprog/process.c`'. ELF is a file format used by Linux, Solaris, and many other operating systems for object files, shared libraries, and executables. You can actually use any compiler and linker that output 80*x*86 ELF executables to produce programs for Pintos. (We've provided compilers and linkers that should do just fine.)

You should realize immediately that, until you copy a test program to the emulated disk, Pintos will be unable to do useful work. You won't be able to do interesting things until you copy a variety of programs to the disk. You might want to create a clean reference disk and copy that over whenever you trash your '`fs.dsk`' beyond a useful state, which may happen occasionally while debugging.

4.1.4 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/mmu.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 directory.

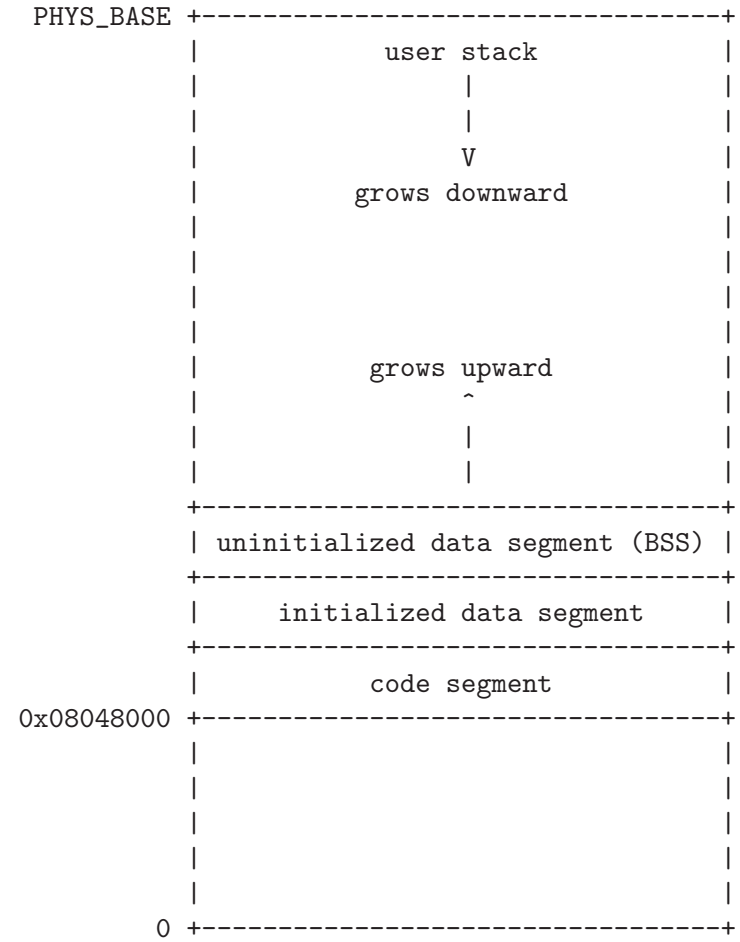
Kernel virtual memory is global. It is always mapped the same way, regardless of what user process or kernel thread is running. In Pintos, kernel virtual memory is mapped one-to-one to physical memory, starting at `PHYS_BASE`. That is, virtual address `PHYS_ADDR` accesses physical address 0, virtual address `PHYS_ADDR + 0x1234` access 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 a user virtual address that doesn't have a page mapped into it will cause a page fault.

You must handle memory fragmentation gracefully, that is, a process that needs N pages of user virtual memory must not require those pages to be contiguous in kernel virtual memory.

4.1.4.1 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:



In this project, the user stack is fixed in size, but in project 3 it will be allowed to grow. Traditionally, the size of the uninitialized data segment can be adjusted with a system call, but you will not have to implement this.

The code segment in Pintos starts at user virtual address `0x08084000`, approximately 128 MB from the bottom of the address space. This value is specified in [SysV-i386] and has no deep significance.

The linker sets the layout of a user program in memory, as directed by a "linker script" that tells it the names and locations of the various program segments. You can learn more about linker scripts by reading the "Scripts" chapter in the linker manual, accessible via '`info ld`'.

To view the layout of a particular executable, run `objdump` (80x86) or `i386-elf-objdump` (SPARC) with the '`-p`' option.

4.1.5 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. The first method is to 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/mmu.h`'. This is the simplest way to handle user memory access.

The second method is to 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.cc`'. 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 a page of memory. 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:

```
/* Tries to copy a byte from user address USRC to kernel address KDST.
   Returns true if successful, false if USRC is invalid. */
static inline bool get_user (uint8_t *kdst, const uint8_t *usrc) {
  int eax;
  asm ("movl $1f, %%eax; movb %2, %%al; movb %%al, %0; 1:"
       : "=m" (*kdst), "=&a" (eax) : "m" (*usrc));
  return eax != 0;
}

/* Tries to write BYTE to user address UDST.
   Returns true if successful, false if UDST is invalid. */
static inline bool put_user (uint8_t *udst, uint8_t byte) {
  int eax;
  asm ("movl $1f, %%eax; movb %b2, %0; 1:"
       : "=m" (*udst), "=&a" (eax) : "r" (byte));
  return eax != 0;
}
```

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 causes `eax` to be set to 0 and its former value copied into `eip`.

4.2 Suggested Order of Implementation

We suggest first implementing the following, which can happen in parallel:

- Argument passing (see Section 4.3.3 [Argument Passing], page 52). Every user programs will page fault immediately until argument passing is implemented.

  For now, you may simply wish to change

  ```
  *esp = PHYS_BASE;
  ```

  to

  ```
  *esp = PHYS_BASE - 12;
  ```

  in `setup_stack()`. That will work for any test program that doesn't examine its arguments, although its name will be printed as `(null)`.
- User memory access (see Section 4.1.5 [Accessing User Memory], page 50). All system calls need to read user memory. Few system calls need to write to user memory.
- System call infrastructure (see Section 4.3.4 [System Calls], page 52). Implement enough code to read the system call number from the user stack and dispatch to a handler based on it.
- The `exit` system call. 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`').
- The `write` system call for writing to fd 1, the system console. All of our test programs write to the console (the user process version of `printf()` is implemented this way), so they will all malfunction until `write` is available.
- For now, change `process_wait()` to an infinite loop (one that waits forever). The provided implementation returns immediately, so Pintos will power off before any processes actually get to run. You will eventually need to provide a correct implementation.

After the above are implemented, user processes should work minimally. At the very least, they can write to the console and exit correctly. You can then refine your implementation so that some of the tests start to pass.

4.3 Requirements

4.3.1 Design Document

Before you turn in your project, you must copy the project 2 design document template into your source tree under the name '`pintos/src/userprog/DESIGNDOC`' and fill it in. We recommend that you read the design document template before you start working on the project. See Appendix D [Project Documentation], page 89, for a sample design document that goes along with a fictitious project.

4.3.2 Process Termination Messages

Whenever a user process terminates, because it called `exit` or for any other reason, print the process's name and exit code, formatted as if printed by `printf ("%s: exit(%d)\n", ...);`. The name printed should be the full name passed to `process_execute()`, omitting command-line arguments. Do not print these messages when a kernel thread that is not a

user process terminates, or when the `halt` system call is invoked. The message is optional when a process fails to load.

Aside from this, don't print any other messages that Pintos as provided doesn't already print. You may find extra messages useful during debugging, but they will confuse the grading scripts and thus lower your score.

4.3.3 Argument Passing

Currently, `process_execute()` does not support passing arguments to new processes. Implement this functionality, by extending `process_execute()` so that instead of simply taking a program file name as its argument, it divides it into words at spaces. The first word is the program name, the second word is the first argument, and so on. That is, `process_execute("grep foo bar")` should run `grep` passing two arguments `foo` and `bar`.

Within a command line, multiple spaces are equivalent to a single space, so that `process_execute("grep foo bar")` is equivalent to our original example. You can impose a reasonable limit on the length of the command line arguments. For example, you could limit the arguments to those that will fit in a single page (4 kB). (There is an unrelated limit of 128 bytes on command-line arguments that the `pintos` utility can pass to the kernel.)

You can parse argument strings any way you like. If you're lost, look at `strtok_r()`, prototyped in '`lib/string.h`' and implemented with thorough comments in '`lib/string.c`'. You can find more about it by looking at the man page (run `man strtok_r` at the prompt).

See Section 4.5.1 [Program Startup Details], page 58, for information on exactly how you need to set up the stack.

4.3.4 System Calls

Implement the system call handler in '`userprog/syscall.c`'. The skeleton implementation we provide "handles" system calls by terminating the process. It will need to retrieve the system call number, then any system call arguments, and carry appropriate actions.

Implement the following system calls. The prototypes listed are those seen by a user program that includes '`lib/user/syscall.h`'. (This header and all other files in '`lib/user`' are for use by user programs only.) System call numbers for each system call are defined in '`lib/syscall-nr.h`':

`void halt (`*void*`)` [System Call]
: Terminates Pintos by calling `power_off()` (declared in '`threads/init.h`'). This should be seldom used, because you lose some information about possible deadlock situations, etc.

`void exit (`*int status*`)` [System Call]
: Terminates the current user program, returning *status* to the kernel. If the process's parent `wait`s for it (see below), this is the status that will be returned. Conventionally, a *status* of 0 indicates success and nonzero values indicate errors.

`pid_t exec (`*const char *cmd_line*`)` [System Call]
: 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.

`int wait (`*pid_t* `pid)` [System Call]

Waits for process *pid* to die and returns the status it passed to `exit`. Returns -1 if *pid* was terminated by the kernel (e.g. killed due to an exception). If *pid* is does not refer to a child of the calling thread, or if `wait` has already been successfully called for the given *pid*, returns -1 immediately, without waiting.

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()`.

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.

Children are not inherited: if A has child B and B has child C, then `wait(C)` always returns immediately when called from A, even if B is dead.

Consider all the ways a wait can occur: nested waits (A waits for B, then B waits for C), multiple waits (A waits for B, then A waits for C), and so on.

`bool create (`*const char* `*file`, *unsigned* `initial_size)` [System Call]

Creates a new file called *file* initially *initial_size* bytes in size. Returns true if successful, false otherwise.

Consider implementing this function in terms of `filesys_create()`.

`bool remove (`*const char* `*file)` [System Call]

Deletes the file called *file*. Returns true if successful, false otherwise.

Consider implementing this function in terms of `filesys_remove()`.

`int open (`*const char* `*file)` [System Call]

Opens the file called *file*. Returns a nonnegative integer handle called a "file descriptor" (fd), or -1 if the file could not be opened. All open files associated with a process should be closed when the process exits or is terminated.

File descriptors numbered 0 and 1 are reserved for the console: fd 0 is standard input (`stdin`), fd 1 is standard output (`stdout`). These special file descriptors are valid as system call arguments only as explicitly described below.

Consider implementing this function in terms of `filesys_open()`.

`int filesize (`*int* `fd)` [System Call]

Returns the size, in bytes, of the file open as *fd*.

Consider implementing this function in terms of `file_length()`.

`int read (`*int* `fd`, *void* `*buffer`, *unsigned* `size)` [System Call]

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 `kbd_getc()`. (Keyboard input will not work if you pass the '`-v`' option to `pintos`.)

Consider implementing this function in terms of `file_read()`.

`int write (`*int* `fd`*, const void* `*buffer`*, unsigned* `size``)` [System Call]

Writes *size* bytes from *buffer* to the open file *fd*. Returns the number of bytes actually written, or -1 if the file 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 -1 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.

Consider implementing this function in terms of `file_write()`.

`void seek (`*int* `fd`*, unsigned* `position``)` [System Call]

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.

Consider implementing this function in terms of `file_seek()`.

`unsigned tell (`*int* `fd``)` [System Call]

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.

Consider implementing this function in terms of `file_tell()`.

`void close (`*int* `fd``)` [System Call]

Closes file descriptor *fd*.

Consider implementing this function in terms of `file_close()`.

The file defines other syscalls. Ignore them for now. You will implement some of them in project 3 and the rest in project 4, so be sure to design your system with extensibility in mind.

To implement syscalls, you need to provide ways to read and write data in user virtual address space. You need this ability before you can even obtain the system call number, because the system call number is on the user's stack in the user's virtual address space. This can be a bit tricky: what if the user provides an invalid pointer, a pointer into kernel memory, or a block partially in one of those regions? You should handle these cases by terminating the user process. We recommend writing and testing this code before implementing any other system call functionality.

You must synchronize system calls so that any number of user processes can make them at once. In particular, it is not safe to call into the file system code provided in the '`filesys`' directory from multiple threads at once. For now, we recommend adding a

single lock that controls access to the file system code. You should acquire this lock before calling any functions in the '`filesys`' directory, and release it afterward. Don't forget that `process_execute()` also accesses files. **For now, we recommend against modifying code in the '`filesys`' directory.**

We have provided you a user-level function for each system call in '`lib/user/syscall.c`'. These provide a way for user processes to invoke each system call from a C program. Each uses a little inline assembly code to invoke the system call and (if appropriate) returns the system call's return value.

When you're done with this part, and forevermore, Pintos should be bulletproof. Nothing that a user program can do should ever cause the OS to crash, panic, fail an assertion, or otherwise malfunction. It is important to emphasize this point: our tests will try to break your system calls in many, many ways. You need to think of all the corner cases and handle them. The sole way a user program should be able to cause the OS to halt is by invoking the `halt` system call.

If a system call is passed an invalid argument, acceptable options include returning an error value (for those calls that return a value), returning an undefined value, or terminating the process.

See Section 4.5.2 [System Call Details], page 59, for details on how system calls work.

4.3.5 Denying Writes to Executables

Add code to deny writes to files in use as executables. Many OSes do this because of the unpredictable results if a process tried to run code that was in the midst of being changed on disk. This is especially important once virtual memory is implemented in project 3, but it can't hurt even now.

You can use `file_deny_write()` to prevent writes to an open file. Calling `file_allow_write()` on the file will re-enable them (unless the file is denied writes by another opener). Closing a file will also re-enable writes. Thus, to deny writes to process's executable, you must keep it open as long as the process is still running.

4.4 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 disk (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.

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 4.5.1 [Program Startup Details], page 58).

How can I can disassemble user programs?

The `objdump` (80*x*86) 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 (see Section E.5 [gdb], page 95).

Why do many C include files not work in 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.)

Can I use lib*foo* in my Pintos programs?

The chances are good that lib*foo* 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 [Debugging User Programs], page 96.

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.

Keyboard input doesn't work with `pintos` option '`-v`'.
: Serial input isn't implemented. Don't use '`-v`' if you want to use the shell or otherwise need keyboard input.

4.4.1 Argument Passing FAQ

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

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.

4.4.2 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. In the next project, you will implement a better solution.

4.5 80x86 Calling Convention

This section summarizes important points of the convention used for normal function calls on 32-bit 80*x*86 implementations of Unix. Some details are omitted for brevity. If you do want all the details, you can refer to [SysV-i386].

The basic 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.
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 80*x*86 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 80*x*86 `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 stack addresses are arbitrary:

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

4.5.1 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 is responsible for setting up the arguments for the initial function on the stack, in accordance with the calling convention explained in the preceding section, before it allows the user program to begin executing.

Consider the following example command: '`/bin/ls -l foo bar`'. First, the kernel must break the command into words, as '`/bin/ls`', '`-l`', '`foo`', and '`bar`', and place them 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 order ensure 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 show 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
`0xbffffffc`	`argv[3][...]`	'bar\0'	`char[4]`
`0xbffffff8`	`argv[2][...]`	'foo\0'	`char[4]`
`0xbffffff5`	`argv[1][...]`	'-l\0'	`char[3]`
`0xbfffffed`	`argv[0][...]`	'/bin/ls\0'	`char[8]`
`0xbfffffec`	word-align	0	`uint8_t`
`0xbfffffe8`	`argv[4]`	0	`char *`
`0xbfffffe4`	`argv[3]`	`0xbffffffc`	`char *`
`0xbfffffe0`	`argv[2]`	`0xbffffff8`	`char *`
`0xbfffffdc`	`argv[1]`	`0xbffffff5`	`char *`
`0xbfffffd8`	`argv[0]`	`0xbfffffed`	`char *`
`0xbfffffd4`	`argv`	`0xbfffffd8`	`char **`
`0xbfffffd0`	`argc`	4	`int`
`0xbfffffcc`	return address	0	`void (*) ()`

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

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/mmu.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:

```
bffffc0                                         00 00 00 00 |            ....|
bffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |................|
bffffe0  f8 ff ff bf fc ff ff bf-00 00 00 00 00 2f 62 69 |............./bi|
bfffff0  6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|
```

4.5.2 System Call Details

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" inter-

rupts, because they are caused by entities outside the CPU (see Section 2.2.3.3 [External Interrupt Handling], page 24).

The operating system also deals with software exceptions, which are events that occur in program code (see Section 2.2.3.2 [Internal Interrupt Handling], page 23). 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.

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.

5 Project 3: Virtual Memory

By now you should be familiar with the inner workings of Pintos. Your OS can properly handle multiple threads of execution with proper synchronization, and can load multiple user programs at once. However, the number and size of programs that can run is limited by the machine's main memory size. In this assignment, you will remove that limitation.

You will build this assignment on top of the last one. It will benefit you to get your project 2 in good working order before this assignment so those bugs don't keep haunting you. Test programs from the previous project should also work with this project.

You will continue to handle Pintos disks and file systems the same way you did in the previous assignment (see Section 4.1.2 [Using the File System], page 46).

5.1 Background

5.1.1 Source Files

You will work in the '`vm`' directory for this project. The '`vm`' directory contains only '`Makefile`'s. The only change from '`userprog`' is that this new '`Makefile`' turns on the setting '`-DVM`'. All code you write will be in new files or in files introduced in earlier projects.

You will probably be encountering just a few files for the first time:

'`devices/disk.h`'
'`devices/disk.c`'
: Provides access to the physical disk, abstracting away the rather awful IDE interface. You will use this interface to access the swap disk.

5.1.2 Page Faults

For the last assignment, whenever a context switch occurred, the new process installed its own page table into the machine. The new page table contained all the virtual-to-physical translations for the process. Whenever the processor needed to look up a translation, it consulted the page table. As long as the process only accessed memory that it owned, all was well. If the process accessed memory it didn't own, it "page faulted" and `page_fault()` terminated the process.

When we implement virtual memory, the rules have to change. A page fault is no longer necessarily an error, since it might only indicate that the page must be brought in from a disk file or from swap. You will have to implement a more sophisticated page fault handler to handle these cases.

5.1.2.1 Page Table Structure

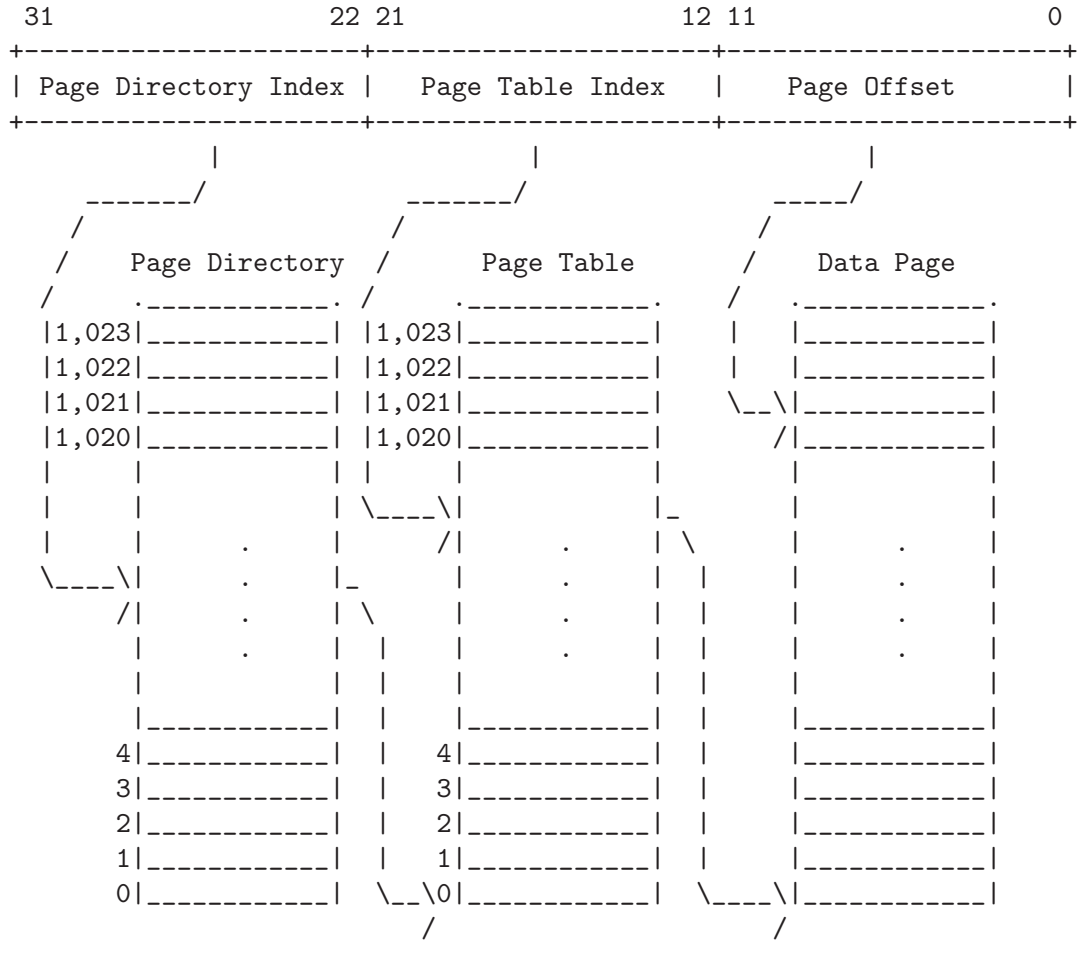
On the 80*x*86, the page table format is fixed by hardware. We have provided code for managing page tables for you to use in '`userprog/pagedir.c`'. The functions in there provide an abstract interface to all the page table functionality that you should need to complete the project. However, you may still find it worthwhile to understand a little about the hardware page table format, so we'll go into a little of detail about that in this section.

The top-level paging data structure is a page called the "page directory" (PD) arranged as an array of 1,024 32-bit page directory entries (PDEs), each of which represents 4 MB

of virtual memory. Each PDE may point to the physical address of another page called a "page table" (PT) arranged, similarly, as an array of 1,024 32-bit page table entries (PTEs), each of which translates a single 4 kB virtual page to a physical page.

Translation of a virtual address into a physical address follows the three-step process illustrated in the diagram below:[1]

1. The most-significant 10 bits of the virtual address (bits 22...32) index the page directory. If the PDE is marked "present," the physical address of a page table is read from the PDE thus obtained. If the PDE is marked "not present" then a page fault occurs.
2. The next 10 bits of the virtual address (bits 12...21) index the page table. If the PTE is marked "present," the physical address of a data page is read from the PTE thus obtained. If the PTE is marked "not present" then a page fault occurs.
3. The least-significant 12 bits of the virtual address (bits 0...11) are added to the data page's physical base address, yielding the final physical address.



[1] Actually, virtual to physical translation on the 80x86 architecture occurs via an intermediate "linear address," but Pintos (and most other 80x86 OSes) set up the CPU so that linear and virtual addresses are one and the same. Thus, you can effectively ignore this CPU feature.

5.1.2.2 Working with Virtual Addresses

Header '`threads/mmu.h`' has useful functions for various operations on virtual addresses. You should look over the header yourself. The most important functions are described below.

`uintptr_t pd_no (const void *va)` [Function]
Returns the page directory index for virtual address *va*.

`uintptr_t pt_no (const void *va)` [Function]
Returns the page table index for virtual address *va*.

`unsigned pg_ofs (const void *va)` [Function]
Returns the page offset of virtual address *va*.

`void * pg_round_down (const void *va)` [Function]
Returns *va* rounded down to the nearest page boundary, that is, *va* with its page offset set to 0.

`void * pg_round_up (const void *va)` [Function]
Returns *va* rounded up to the nearest page boundary.

5.1.2.3 Accessed and Dirty Bits

Most of the page table is under the control of the operating system, but two bits in each page table entry are also manipulated by the CPU. On any read or write to the page referenced by a PTE, the CPU sets the PTE's *accessed bit* to 1; on any write, the CPU sets the *dirty bit* to 1. The CPU never resets these bits to 0, but the OS may do so.

You will need to use the accessed and dirty bits in your submission to choose which pages to evict from memory and to decide whether evicted pages need to be written to disk. The page table code in '`userprog/pagedir.c`' provides functions for checking and setting these bits. These functions are described at the end of this section.

You need to watch out for *aliases*, that is, two (or more) different virtual pages that refer to the same physical page frame. When an aliased page is accessed, the accessed and dirty bits are updated in only one page table entry (the one for the virtual address used to access the page). The accessed and dirty bits for the other aliased virtual addresses are not updated.

In Pintos, every user virtual page is aliased to its kernel virtual address. You must manage these aliases somehow. For example, your code could check and update the accessed and dirty bits for both addresses. Alternatively, the kernel could avoid the problem by only accessing user data through the user virtual address.

Other aliases should only arise if you implement sharing, as extra credit (see [VM Extra Credit], page 69), or as bugs elsewhere in your code.

`bool pagedir_is_dirty (uint32_t *pd, const void *vpage)` [Function]
`bool pagedir_is_accessed (uint32_t *pd, const void *vpage)` [Function]
Returns true if page directory *pd* contains a page table entry for virtual page *vpage* that is marked dirty (or accessed). Otherwise, returns false.

`void pagedir_set_dirty (`*uint32_t *`**pd**, const void *`**vpage**, bool* `**value**`) [Function]
`void pagedir_set_accessed (`*uint32_t *`**pd**, const void *`**vpage**, bool* `**value**`) [Function]
If page directory *pd* has a page table entry for *vpage*, then its dirty (or accessed) bit is set to *value*.

5.1.3 Disk as Backing Store

VM systems effectively use memory as a cache for disk. From another perspective, disk is a "backing store" for memory. This provides the abstraction of an (almost) unlimited virtual memory size. You must implement such a system, with the additional constraint that performance should be close to that provided by physical memory. You can use dirty bits to tell whether pages need to be written back to disk when they're evicted from main memory, and the accessed bits for page replacement algorithms (see Section 5.1.2.3 [Accessed and Dirty Bits], page 63).

As with any caching system, performance depends on the policy used to decide what to keep in the cache and what to evict. On a page fault, the kernel must decide which page to replace. Ideally, it will throw out a page that will not be referenced for a long time, keeping in memory those pages that are soon to be referenced. Another consideration is that if the replaced page has been modified, the page must be first written to disk before the needed page can be brought in. Many virtual memory systems avoid this extra overhead by writing modified pages to disk in advance, so that later page faults can be completed more quickly (but you do not have to implement this optimization).

5.1.4 Memory Mapped Files

The file system is most commonly accessed with `read` and `write` system calls. A secondary interface is to "map" the file into the virtual address space. The program can then use load and store instructions directly on the file data. An alternative view is to see the file system is as "durable memory": files just store data structures, so if you access ordinary data structures using normal program instructions, why not access durable data structures the same way?

Suppose file '`foo`' is `0x1000` bytes (4 kB, or one page) long. If '`foo`' is mapped into memory starting at address `0x5000`, then any memory accesses to locations `0x5000...0x5fff` will access the corresponding bytes of '`foo`'.

A consequence of memory mapped files is that address spaces are sparsely populated with lots of segments, one for each memory mapped file (plus one each for code, data, and stack).

5.2 Requirements

This assignment is an open-ended design problem. We are going to say as little as possible about how to do things. Instead we will focus on what functionality we require your OS to support. We will expect you to come up with a design that makes sense. You will have the freedom to choose how to handle page faults, how to organize the swap disk, how to implement paging, etc.

5.2.1 Design Document

Before you turn in your project, you must copy the project 3 design document template into your source tree under the name '`pintos/src/vm/DESIGNDOC`' and fill it in. We recommend that you read the design document template before you start working on the project. See Appendix D [Project Documentation], page 89, for a sample design document that goes along with a fictitious project.

5.2.2 Page Table Management

Implement page directory and page table management to support virtual memory. You will need data structures to accomplish the following tasks:

- Some way of translating in software from virtual page frames to physical page frames. Pintos provides a hash table that you may find useful for this purpose (see Section 2.4.1 [Hash Table], page 27).

  It is possible to do this translation without adding a new data structure, by modifying the code in '`userprog/pagedir.c`'. However, if you do that you'll need to carefully study and understand section 3.7 in [IA32-v3], and in practice it is probably easier to add a new data structure.

- Some way of finding a page on disk (in a file or in swap) if it is not in memory.

  You can generalize the virtual-to-physical page table, so that it allows you to locate a page wherever it is in physical memory or on disk, or you can make this a separate table.

- Some way of translating from physical page frames back to virtual page frames, so that when you evict a physical page from its frame, you can invalidate its page table entry (or entries).

The page fault handler, `page_fault()` in '`threads/exception.c`', needs to do roughly the following:

1. Locate the page backing the virtual address that faulted. It might be in the file system, in swap, or it might be an invalid virtual address. If you implement sharing, it might even already be in physical memory, but not in the page table.

   If the virtual address is invalid, that is, if there's nothing assigned to go there, or if the virtual address is above `PHYS_BASE`, meaning that it belongs to the kernel instead of the user, then the process's memory access must be disallowed. In this case, terminate the process and free all of its resources.

2. If the page is not in physical memory, fetch it by appropriate means. If necessary to make room, first evict some other page from memory. (When you do that you need to first remove references to the page from any page table that refers to it.)

3. Point the page table entry for the faulting virtual address to the physical page. You can use the functions in '`userprog/pagedir.c`'.

You'll need to modify the ELF loader in '`userprog/process.c`' to do page table management according to your new design. As supplied, it reads all the process's pages from disk and initializes the page tables for them at the same time. As a first step, you might want to leave the code that reads the pages from disk, but use your new page table management code to construct the page tables only as page faults occur for them.

You should use the `palloc_get_page()` function to get the page frames that you use for storing user virtual pages. Be sure to pass the `PAL_USER` flag to this function when you do so, because that allocates pages from a "user pool" separate from the "kernel pool" that other calls to `palloc_get_page()` make (see [Why PAL_USER?], page 70).

You might find the Pintos bitmap code, in '`lib/kernel/bitmap.c`' and '`lib/kernel/bitmap.h`', useful for tracking pages. A bitmap is an array of bits, each of which can be true or false. Bitmaps are typically used to track usage in a set of (identical) resources: if resource *n* is in use, then bit *n* of the bitmap is true.

There are many possible ways to implement virtual memory. The above is simply an outline of our suggested implementation.

5.2.3 Paging To and From Disk

Implement paging to and from files and the swap disk. You may use the disk on interface `hd1:1` as the swap disk, using the disk interface prototyped in `devices/disk.h`. From the '`vm/build`' directory, use the command `pintos-mkdisk swap.dsk n` to create an *n* MB swap disk named '`swap.dsk`'. Afterward, '`swap.dsk`' will automatically be attached as `hd1:1` when you run `pintos`. Alternatively, you can tell `pintos` to use a temporary *n*-MB swap disk for a single run with '`--swap-disk=n`'.

You will need routines to move a page from memory to disk and from disk to memory, where "disk" is either a file or the swap disk. If you do a good job, your VM should still work when you implement your own file system for the next assignment.

To fully handle page faults, you will need a way to track pages that are used by a process but which are not in physical memory. Pages in swap should not be constrained to any particular ordering. You will also need a way to track physical page frames, to find an unused one when needed, or to evict a page when memory is needed but no empty pages are available. The page table data structure that you designed should do most of the work for you.

Implement a global page replacement algorithm. You should be able to use the "accessed" and "dirty" bits (see Section 5.1.2.3 [Accessed and Dirty Bits], page 63) to implement an approximation to LRU. Your algorithm should perform at least as well as the "second chance" or "clock" algorithm.

Your design should allow for parallelism. Multiple processes should be able to process page faults at once. If one page fault require I/O, in the meantime processes that do not fault should continue executing and other page faults that do not require I/O should be able to complete. These criteria require some synchronization effort.

5.2.4 Lazy Loading

Since you will already be paging from disk, you should implement a "lazy" loading scheme for new processes. When a process is created, it will not need all of its resources immediately, so it doesn't make sense to load all its code, data, and stack into memory when the process is created. Instead, bring pages in from the executable only as needed. Use the executable file itself as backing store for read-only segments, since these segments won't change. This means that read-only pages should not be written to swap.

The core of the program loader is the loop in `load_segment()` in '`userprog/process.c`'. Each time around the loop, `read_bytes` receives the number of bytes to read from the

executable file and `zero_bytes` receives the number of bytes to initialize to zero following the bytes read. The two always sum to `PGSIZE` (4,096). The handling of a page depends on these variables' values:

- If `read_bytes` equals `PGSIZE`, the page should be demand paged from disk on its first access.
- If `zero_bytes` equals `PGSIZE`, the page does not need to be read from disk at all because it is all zeroes. You should handle such pages by creating a new page consisting of all zeroes at the first page fault.
- If neither `read_bytes` nor `zero_bytes` equals `PGSIZE`, then part of the page is to be read from disk and the remainder zeroed. This is a special case. You are allowed to handle it by reading the partial page from disk at executable load time and zeroing the rest of the page. This is the only case in which we will allow you to load a page in a non-"lazy" fashion. Many real OSes such as Linux do not load partial pages lazily.

Incidentally, if you have trouble handling the third case above, you can eliminate it temporarily by linking the test programs with a special "linker script." Read '`Makefile.userprog`' for details. We will not test your submission with this special linker script, so the code you turn in must properly handle all cases.

5.2.5 Stack Growth

Implement stack growth. In project 2, the stack was a single page at the top of the user virtual address space, and programs were limited to that much stack. Now, if the stack grows past its current size, allocate additional pages as necessary.

Allocate additional pages only if they "appear" to be stack accesses. Devise a heuristic that attempts to distinguish stack accesses from other accesses.

User programs are buggy if they write to the stack below the stack pointer, because typical real OSes may interrupt a process at any time to deliver a "signal," which pushes data on the stack.[2] However, the 80x86 `PUSH` instruction checks access permissions before it adjusts the stack pointer, so it may cause a page fault 4 bytes below the stack pointer. (Otherwise, `PUSH` would not be restartable in a straightforward fashion.) Similarly, the `PUSHA` instruction pushes 32 bytes at once, so it can fault 32 bytes below the stack pointer.

You will need to be able to obtain the current value of the user program's stack pointer. Within a system call or a page fault generated by a user program, you can retrieve it from `esp` member of the `struct intr_frame` passed to `syscall_handler()` or `page_fault()`, respectively. If you verify user pointers before accessing them (see Section 4.1.5 [Accessing User Memory], page 50), these are the only cases you need to handle. On the other hand, if you depend on page faults to detect invalid memory access, you will need to handle another case, where a page fault occurs in the kernel. Reading `esp` out of the `struct intr_frame` passed to `page_fault()` in that case will obtain the kernel stack pointer, not the user stack pointer. You will need to arrange another way, e.g. by saving `esp` into `struct thread` on the initial transition from user to kernel mode.

[2] This rule is common but not universal. One modern exception is the x86-64 System V ABI, which designates 128 bytes below the stack pointer as a "red zone" that may not be modified by signal or interrupt handlers.

You may impose some absolute limit on stack size, as do most OSes. Some OSes make the limit user-adjustable, e.g. with the `ulimit` command on many Unix systems. On many GNU/Linux systems, the default limit is 8 MB.

The first stack page need not be allocated lazily. You can initialize it with the command line arguments at load time, with no need to wait for it to be faulted in. (Even if you did wait, the very first instruction in the user program is likely to be one that faults in the page.)

5.2.6 Memory Mapped Files

Implement memory mapped files, including the following system calls.

`mapid_t mmap (`*int* `fd`*, void **`addr`*`)` [System Call]

Maps the file open as *fd* into the process's virtual address space. The entire file is mapped into consecutive virtual pages starting at *addr*.

If the file's length is not a multiple of `PGSIZE`, then some bytes in the final mapped page "stick out" beyond the end of the file. Set these bytes to zero when the page is faulted in from disk, and discard them when the page is written back to disk. A partial page need not be lazily loaded, as in the case of a partial page in an executable (see Section 5.2.4 [Lazy Loading], page 66).

If successful, this function returns a "mapping ID" that uniquely identifies the mapping within the process. On failure, it must return -1, which otherwise should not be a valid mapping id, and the process's mappings must be unchanged.

A call to `mmap` may fail if the file open as *fd* has a length of zero bytes. It must fail if *addr* is not page-aligned or if the range of pages mapped overlaps any existing set of mapped pages, including the stack or pages mapped at executable load time. It must also fail if *addr* is 0, because some Pintos code assumes virtual page 0 is not mapped. Finally, file descriptors 0 and 1, representing console input and output, are not mappable.

Your VM system should use the `mmap`'d file itself as backing store for the mapping. That is, to evict a page mapped by `mmap`, write it to the file it was mapped from. (In fact, you may choose to implement executable mappings as special, copy-on-write file mappings.)

`void munmap (`*mapid_t* `mapping`*`)`* [System Call]

Unmaps the mapping designated by *mapping*, which must be a mapping ID returned by a previous call to `mmap` by the same process that has not yet been unmapped.

All mappings are implicitly unmapped when a process exits, whether via `exit` or by any other means. When a mapping is unmapped, whether implicitly or explicitly, all pages written to by the process are written back to the file, and pages not written must not be. The pages are then removed from the process's list of virtual pages.

Closing or removing a file does not unmap any of its mappings. Once created, a mapping is valid until `munmap` is called or the process exits, following the Unix convention. See [Removing an Open File], page 57, for more information.

If two or more processes map the same file, there is no requirement that they see consistent data. Unix handles this by making the two mappings share the same physical page,

but the `mmap` system call also has an argument allowing the client to specify whether the page is shared or private (i.e. copy-on-write).

5.3 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.

This summary is relative to the Pintos base code, but the reference solution for project 3 starts from the reference solution to project 2. See Section 4.4 [Project 2 FAQ], page 55, for the summary of project 2.

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.

```
Makefile.build       |    4
devices/timer.c      |   42 ++
threads/init.c       |    5
threads/interrupt.c  |    2
threads/thread.c     |   31 +
threads/thread.h     |   37 +-
userprog/exception.c |   12
userprog/pagedir.c   |   10
userprog/process.c   |  319 ++++++++++++++-----
userprog/syscall.c   |  545 ++++++++++++++++++++++++++++++-
userprog/syscall.h   |    1
vm/frame.c           |  162 +++++++++
vm/frame.h           |   23 +
vm/page.c            |  297 +++++++++++++++++
vm/page.h            |   50 ++
vm/swap.c            |   85 ++++
vm/swap.h            |   11
17 files changed, 1532 insertions(+), 104 deletions(-)
```

Do we need a working Project 2 to implement Project 3?

Yes.

What extra credit is available?

You may implement sharing: when multiple processes are created that use the same executable file, share read-only pages among those processes instead of creating separate copies of read-only segments for each process. If you carefully designed your page table data structures, sharing of read-only pages should not make this part significantly harder.

5.3.1 Page Table and Paging FAQ

Does the virtual memory system need to support data segment growth?

No. The size of the data segment is determined by the linker. We still have no dynamic allocation in Pintos (although it is possible to "fake" it at the user level by using memory-mapped files). Supporting data segment growth should add little additional complexity to a well-designed system.

Why should I use `PAL_USER` for allocating page frames?

Passing `PAL_USER` to `palloc_get_page()` causes it to allocate memory from the user pool, instead of the main kernel pool. Running out of pages in the user pool just causes user programs to page, but running out of pages in the kernel pool will cause many failures because so many kernel functions need to obtain memory. You can layer some other allocator on top of `palloc_get_page()` if you like, but it should be the underlying mechanism.

Also, you can use the '`-u`' option to `pintos` to limit the size of the user pool, which makes it easy to test your VM implementation with various user memory sizes.

Data pages might need swap space. Can I swap them out at process load?

No. Reading data pages from the executable and writing them to swap immediately at program startup is not demand paging. You need to demand page everything (except partial pages).

5.3.2 Memory Mapped File FAQ

How do we interact with memory-mapped files?

Let's say you want to map a file called '`foo`' into your address space at address `0x10000000`. You open the file then use `mmap`:

```
#include <stdio.h>
#include <syscall.h>

int main (void)
{
    void *addr = (void *) 0x10000000;
    int fd = open ("foo");
    mapid_t map = mmap (fd, addr);
    if (map != -1)
        printf ("success!\n");
}
```

Suppose '`foo`' is a text file and you want to print the first 64 bytes on the screen (assuming, of course, that the length of the file is at least 64). Without `mmap`, you'd need to allocate a buffer, use `read` to get the data from the file into the buffer, and finally use `write` to put the buffer out to the display. But with the file mapped into your address space, you can directly address it like so:

```
write (addr, 64, STDOUT_FILENO);
```

Similarly, if you wanted to replace the first byte of the file, all you need to do is:

```
addr[0] = 'b';
```

When you're done using the memory-mapped file, you simply unmap it:

```
munmap (map);
```

The `mcp` program in '`src/examples`' shows how to copy a file using memory-mapped I/O.

6 Project 4: File Systems

In the previous two assignments, you made extensive use of a file system without actually worrying about how it was implemented underneath. For this last assignment, you will fill in the implementation of the file system. You will be working primarily in the '`filesys`' directory.

You may build project 4 on top of project 2 or project 3. In either case, all of the functionality needed for project 2 must work in your filesys submission. If you build on project 3, then all of the project 3 functionality must work also, and you will need to edit '`filesys/Make.vars`' to enable VM functionality. You can receive up to 5% extra credit if you do enable VM.

6.1 Background

6.1.1 New Code

Here are some files that are probably new to you. These are in the '`filesys`' directory except where indicated:

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

'`filesys.h`'
'`filesys.c`'
: Top-level interface to the file system. See Section 4.1.2 [Using the File System], page 46, for an introduction.

'`directory.h`'
'`directory.c`'
: Translates file names to inodes. The directory data structure is stored as a file.

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

'`file.h`'
'`file.c`'
: Translates file reads and writes to disk sector reads and writes.

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

Our file system has a Unix-like interface, so you may also wish to read the Unix man pages for `creat`, `open`, `close`, `read`, `write`, `lseek`, and `unlink`. Our file system has calls that are similar, but not identical, to these. The file system translates these calls into physical disk operations.

All the basic functionality is there in the code above, so that the file system is usable from the start, as you've seen in the previous two projects. However, it has severe limitations which you will remove.

While most of your work will be in '`filesys`', you should be prepared for interactions with all previous parts (as usual).

6.2 Requirements

6.2.1 Design Document

Before you turn in your project, you must copy the project 4 design document template into your source tree under the name '`pintos/src/filesys/DESIGNDOC`' and fill it in. We recommend that you read the design document template before you start working on the project. See Appendix D [Project Documentation], page 89, for a sample design document that goes along with a fictitious project.

6.2.2 Indexed and Extensible Files

The basic file system allocates files as a single extent, making it vulnerable to external fragmentation. 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.)

You can assume that the disk will not be larger than 8 MB. You must support files as large as the disk (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 with indexed inodes file growth is 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 UNIX and most other 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 disk (minus metadata). This also applies to the root directory file, which should now be allowed to expand beyond its initial limit of 16 files.

The user is 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.

6.2.3 Subdirectories

Implement a hierarchical name space. 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, at your option. You must allow full path names to be much longer than 14 characters.

The current directory is maintained separately for each process. At startup, the initial process's current directory is the root 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.

Update the existing system calls so that, anywhere a file name is provided by the caller, an absolute or relative path name may used.

Update the `remove` system call so that it can delete empty directories in addition to regular files. Directories can only be deleted if they do not contain any files or subdirectories.

Implement the following new system calls:

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

`bool mkdir (`*const char *dir*`)` [System Call]
: 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.

`void lsdir (`*void*`)` [System Call]
: Prints a list of files in the current directory to `stdout`, one per line, in no particular order.

We have provided `ls` and `mkdir` user programs, which are straightforward once the above syscalls are implemented. In Unix, these are programs rather than built-in shell commands, but `cd` is a shell command.

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

You may support '`.`' and '`..`' for a small amount of extra credit.

6.2.4 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 stored in the cache, and if so, fetch it immediately from the cache without going to disk. Otherwise, fetch the block from disk into cache, evicting an older entry if necessary. You are limited to a cache no greater than 64 sectors in size.

Be sure to choose an intelligent cache replacement algorithm. Experiment to see what combination of accessed, dirty, and other information results in the best performance, as measured by the number of disk accesses. For example, metadata is generally more valuable to cache than data.

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.

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.

Your implementation should also include the following features:

write-behind:

Keep dirty blocks in the cache, instead of immediately writing modified data to disk. Write dirty blocks to disk whenever they are evicted. Because write-behind makes your file system more fragile in the face of crashes, in addition you should periodically write all dirty, cached blocks back to disk. The cache should also be written back to disk in `filesys_done()`, so that halting Pintos flushes the cache.

If you have `timer_sleep()` from the first project working, this is an excellent application for it. If you're still using the base implementation of `timer_sleep()`, be aware that it busy-waits, which is not an acceptable solution. If `timer_sleep()`'s delays seem too short or too long, reread the explanation of the '`-r`' option to `pintos` (see Section 1.1.4 [Debugging versus Testing], page 4).

read-ahead:

Your buffer cache should automatically fetch the next block of a file into the cache when one block of a file is read, in case that block is about to be 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. These groups have often turned in projects that failed most or all of the tests.

6.2.5 Synchronization

The provided file system requires external synchronization, that is, callers must ensure that only one thread can be running in the file system code at once. Your submission must adopt a finer-grained synchronization strategy that does not require external synchronization. To the extent possible, operations on independent entities should be independent, so that they do not need to wait on each other.

Operations on different cache blocks must be independent. In particular, when I/O is required on a particular block, operations on other blocks that do not require I/O should proceed without having to wait for the I/O to complete.

Multiple processes must be able to access a single file at once. Multiple reads of a single file must be able to complete without waiting for one another. When writing to a file does not extend the file, multiple processes should also be able to write a single file at once. A read of a file by one process when the file is being written by another process is allowed to show that none, all, or part of the write has completed. (However, after the `write` system call returns to its caller, all subsequent readers must see the change.) Similarly, when two processes simultaneously write to the same part of a file, their data may be interleaved.

On the other hand, extending a file and writing data into the new section must be atomic. Suppose processes A and B both have a given file open and both are positioned at

end-of-file. If A reads and B writes the file at the same time, A may read all, part, or none of what B writes. However, A may not read data other than what B writes, e.g. if B's data is all nonzero bytes, A is not allowed to see any zeros.

Operations on different directories should take place concurrently. Operations on the same directory may wait for one another.

6.3 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.

This summary is relative to the Pintos base code, but the reference solution for project 4 is based on the reference solution to project 3. Thus, the reference solution runs with virtual memory enabled. See Section 5.3 [Project 3 FAQ], page 69, for the summary of project 3.

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.

```
Makefile.build       |    5
devices/timer.c      |   42 ++
filesys/Make.vars    |    6
filesys/cache.c      |  473 ++++++++++++++++++++++++
filesys/cache.h      |   23 +
filesys/directory.c  |   99 ++++-
filesys/directory.h  |    3
filesys/file.c       |    4
filesys/filesys.c    |  194 +++++++++-
filesys/filesys.h    |    5
filesys/free-map.c   |   45 +-
filesys/free-map.h   |    4
filesys/fsutil.c     |    8
filesys/inode.c      |  444 ++++++++++++++++++++-----
filesys/inode.h      |   11
threads/init.c       |    5
threads/interrupt.c  |    2
threads/thread.c     |   32 +
threads/thread.h     |   38 +-
userprog/exception.c |   12
userprog/pagedir.c   |   10
userprog/process.c   |  332 +++++++++++++----
userprog/syscall.c   |  582 ++++++++++++++++++++++++++++++-
userprog/syscall.h   |    1
vm/frame.c           |  161 ++++++++
```

```
vm/frame.h           |   23 +
vm/page.c            |  297 +++++++++++++++++
vm/page.h            |   50 ++
vm/swap.c            |   85 ++++
vm/swap.h            |   11
30 files changed, 2721 insertions(+), 286 deletions(-)
```

What extra credit opportunities are available?

You may implement Unix-style support for '`.`' and '`..`' in relative paths in their projects.

You may submit with VM enabled.

Can `DISK_SECTOR_SIZE` change?

No, `DISK_SECTOR_SIZE` is fixed at 512. This is a fixed property of IDE disk hardware.

What's the directory separator character?

Forward slash ('`/`').

6.3.1 Indexed Files FAQ

What is the largest file size that we are supposed to support?

The disk we create will be 8 MB or smaller. However, individual files will have to be smaller than the disk to accommodate the metadata. You'll need to consider this when deciding your inode organization.

6.3.2 Subdirectories FAQ

Why is `cd` a shell command?

The current directory of each process is independent. A `cd` program could change its own current directory, but that would have no effect on the shell. In fact, Unix-like systems don't provide any way for one process to change another process's current working directory.

6.3.3 Buffer Cache FAQ

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`, if you want, and you can 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 '`filesys/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()` so that it reads the `struct inode_disk` from the storage hierarchy before using it.

Appendix A References

A.1 Hardware References

[IA32-v1]. IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture. Basic 80x86 architecture and programming environment.

[IA32-v2a]. IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference A-M. 80x86 instructions whose names begin with A through M.

[IA32-v2b]. IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference N-Z. 80x86 instructions whose names begin with N through Z.

[IA32-v3]. IA-32 Intel Architecture Software Developer's Manual Volume 3a: System Programming Guide. Operating system support, including segmentation, paging, tasks, interrupt and exception handling.

IA-32 Intel Architecture Software Developer's Manual Volume 3b: System Programming Guide. Debugging, performance monitoring, system management mode, and Intel Virtualization Technology.

[FreeVGA]. FreeVGA Project. Documents the VGA video hardware used in PCs.

[kbd]. Keyboard scancodes. Documents PC keyboard interface.

[ATA-3]. AT Attachment-3 Interface (ATA-3) Working Draft. Draft of an old version of the ATA aka IDE interface for the disks used in most desktop PCs.

[PC16550D]. National Semiconductor PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs. Datasheet for a chip used for PC serial ports.

[8254]. Intel 8254 Programmable Interval Timer. Datasheet for PC timer chip.

[8259A]. Intel 8259A Programmable Interrupt Controller (8259A/8259A-2). Datasheet for PC interrupt controller chip.

A.2 Software References

[ELF1]. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book I: Executable and Linking Format. The ubiquitous format for executables in modern Unix systems.

[ELF2]. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book II: Processor Specific (Intel Architecture). 80x86-specific parts of ELF.

[ELF3]. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 Book III: Operating System Specific (UNIX System V Release 4). Unix-specific parts of ELF.

[SysV-ABI]. System V Application Binary Interface: Edition 4.1. Specifies how applications interface with the OS under Unix.

[SysV-i386]. System V Application Binary Interface: Intel386 Architecture Processor Supplement: Fourth Edition. 80x86-specific parts of the Unix interface.

[SysV-ABI-update]. System V Application Binary Interface—DRAFT—24 April 2001. A draft of a revised version of [SysV-ABI] which was never completed.

A.3 Operating System Design References

[4.4BSD]. M. K. McKusick, K. Bostic, M. J. Karels, J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley 1996.

Appendix B 4.4BSD Scheduler

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

For project 1, you must implement the scheduler described in this appendix. Our scheduler resembles the one described in [4.4BSD], which is one example of a *multilevel feedback queue* scheduler. This type of scheduler maintains several queues of ready-to-run threads, where each queue holds threads with a different priority. At any given time, the scheduler chooses a thread from the highest-priority non-empty queue. If the highest-priority queue contains multiple threads, then they run in "round robin" order.

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

The 4.4BSD scheduler does not include priority donation.

B.1 Niceness

Thread priority is dynamically determined by the scheduler using a formula given below. However, each thread also has an integer *nice* value that determines how "nice" the thread should be to other threads. A *nice* of zero does not affect thread priority. A positive *nice*, to the maximum of 20, increases the numeric priority of a thread, decreasing its effective priority, and causes it to give up some CPU time it would otherwise receive. On the other hand, a negative *nice*, to the minimum of -20, tends to take away CPU time from other threads.

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

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

`void thread_set_nice (`*int* `new_nice`*`)`* [Function]
Sets the current thread's *nice* value to *new_nice* and recalculates the thread's priority based on the new value (see Section B.2 [Calculating Priority], page 81). If the running thread no longer has the highest priority, yields.

B.2 Calculating Priority

Our scheduler has 64 priorities and thus 64 ready queues, numbered 0 (`PRI_MIN`) through 63 (`PRI_MAX`). Lower numbers correspond to *higher* priorities, so that priority 0 is the

highest priority and priority 63 is the lowest. 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

```
priority = (recent_cpu / 4) + (nice * 2),
```

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

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

B.3 Calculating *recent_cpu*

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

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

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

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

$$x(1) = f(1),$$
$$x(2) = af(1) + f(2),$$
$$\vdots$$
$$x(5) = a^4 f(1) + a^3 f(2) + a^2 f(3) + af(4) + f(5).$$

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

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:

```
recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice,
```

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

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

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

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

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

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

B.4 Calculating *load_avg*

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

load_avg = (59/60)**load_avg* + (1/60)**ready_threads*,

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

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

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

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

B.5 Summary

This section summarizes the calculations required to implement the scheduler. It is not a complete description of scheduler requirements.

Every thread has a *nice* value between -20 and 20 directly under its control. Each thread also has a priority, between 0 (`PRI_MIN`) through 63 (`PRI_MAX`), which is recalculated using the following formula whenever the value of either term changes:

priority = (*recent_cpu* / 4) + (*nice* * 2).

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

recent_cpu = (2**load_avg*)/(2**load_avg* + 1) * *recent_cpu* + *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:

```
load_avg = (59/60)*load_avg + (1/60)*ready_threads.
```

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

B.6 Fixed-Point Real Arithmetic

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

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

Suppose that we are using a $p.q$ fixed-point format, and let $f = 2^q$. By the definition above, we can convert an integer or real number into $p.q$ format by multiplying with f. For example, in 17.14 format the fraction 59/60 used in the calculation of *load_avg*, above, is $(59/60)2^{14} = 16{,}111$ (rounded to nearest). To convert a fixed-point value back to an integer, divide by f. (The normal '`/`' operator in C rounds toward zero, that is, it rounds positive numbers down and negative numbers up. To round to nearest, add $f/2$ to a positive number, or subtract it from a negative number, before dividing.)

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

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

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

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

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

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

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

Appendix C Coding Standards

All of you should have taken a class like CS 2604, so we expect you to be familiar with some set of coding standards such as CS 2604 General Programming Standards. We also recommend that you review the Stanford CS 107 Coding Standards. We expect code at the "Peer-Review Quality" level described in that document.

Our standards for coding are most important for grading. We want to stress that aside from the fact that we are explicitly basing part of your grade on these things, good coding practices will improve the quality of your code. This makes it easier for your partners to interact with it, and ultimately, will improve your chances of having a good working program. That said once, the rest of this document will discuss only the ways in which our coding standards will affect our grading.

C.1 Style

Style, for the purposes of our grading, refers to how readable your code is. At minimum, this means that your code is well formatted, your variable names are descriptive and your functions are decomposed and well commented. Any other factors which make it hard (or easy) for us to read or use your code will be reflected in your style grade.

The existing Pintos code is written in the GNU style and largely follows the GNU Coding Standards. We encourage you to follow the applicable parts of them too, especially chapter 5, "Making the Best Use of C." Using a different style won't cause actual problems, but it's ugly to see gratuitous differences in style from one function to another. If your code is too ugly, it will cost you points.

Please limit C source file lines to at most 79 characters long.

Pintos comments sometimes refer to external standards or specifications by writing a name inside square brackets, like this: `[IA32-v3]`. These names refer to the reference names used in this documentation (see Appendix A [References], page 79).

If you remove existing Pintos code, please delete it from your source file entirely. Don't just put it into a comment or a conditional compilation directive, because that makes the resulting code hard to read.

We're only going to do a compile in the directory for the project being submitted. You don't need to make sure that the previous projects also compile.

Project code should be written so that all of the subproblems for the project function together, that is, without the need to rebuild with different macros defined, etc. If you do extra credit work that changes normal Pintos behavior so as to interfere with grading, then you must implement it so that it only acts that way when given a special command-line option of the form '`-`*`name`*', where *name* is a name of your choice. You can add such an option by modifying `parse_options()` in '`threads/init.c`'.

The introduction describes additional coding style requirements (see Section 1.2.2 [Design], page 6).

C.2 C99

The Pintos source code uses a few features of the "C99" standard library that were not in the original 1989 standard for C. Many programmers are unaware of these feature, so we will describe them. The new features used in Pintos are mostly in new headers:

‘<stdbool.h>’
: Defines macros `bool`, a 1-bit type that takes on only the values 0 and 1, `true`, which expands to 1, and `false`, which expands to 0.

‘<stdint.h>’
: On systems that support them, this header defines types `intn_t` and `uintn_t` for n = 8, 16, 32, 64, and possibly other values. These are 2's complement signed and unsigned types, respectively, with the given number of bits.

  On systems where it is possible, this header also defines types `intptr_t` and `uintptr_t`, which are integer types big enough to hold a pointer.

  On all systems, this header defines types `intmax_t` and `uintmax_t`, which are the system's signed and unsigned integer types with the widest ranges.

  For every signed integer type *type*`_t` defined here, as well as for `ptrdiff_t` defined in ‘<stddef.h>’, this header also defines macros *TYPE*`_MAX` and *TYPE*`_MIN` that give the type's range. Similarly, for every unsigned integer type *type*`_t` defined here, as well as for `size_t` defined in ‘<stddef.h>’, this header defines a *TYPE*`_MAX` macro giving its maximum value.

‘<inttypes.h>’
: ‘<stdint.h>’ provides no straightforward way to format the types it defines with `printf()` and related functions. This header provides macros to help with that. For every `intn_t` defined by ‘<stdint.h>’, it provides macros `PRIdn` and `PRIin` for formatting values of that type with `"%d"` and `"%i"`. Similarly, for every `uintn_t`, it provides `PRIon`, `PRIun`, `PRIux`, and `PRIuX`.

  You use these something like this, taking advantage of the fact that the C compiler concatenates adjacent string literals:

```
#include <inttypes.h>
...
int32_t value = ...;
printf ("value=%08"PRId32"\n", value);
```

  The ‘%’ is not supplied by the `PRI` macros. As shown above, you supply it yourself and follow it by any flags, field width, etc.

‘<stdio.h>’
: The `printf()` function has some new type modifiers for printing standard types:

  ‘j’ For `intmax_t` (e.g. ‘%jd’) or `uintmax_t` (e.g. ‘%ju’).

  ‘z’ For `size_t` (e.g. ‘%zu’).

  ‘t’ For `ptrdiff_t` (e.g. ‘%td’).

  Pintos `printf()` also implements a nonstandard ‘’’ flag that groups large numbers with commas to make them easier to read.

C.3 Unsafe String Functions

A few of the string functions declared in the standard ‘<string.h>’ and ‘<stdio.h>’ headers are notoriously unsafe. The worst offenders are intentionally not included in the Pintos C library:

`strcpy()` When used carelessly this function can overflow the buffer reserved for its output string. Use `strlcpy()` instead. Refer to comments in its source code in `lib/string.c` for documentation.

`strncpy()`
This function can leave its destination buffer without a null string terminator. It also has performance problems. Again, use `strlcpy()`.

`strcat()` Same issue as `strcpy()`. Use `strlcat()` instead. Again, refer to comments in its source code in `lib/string.c` for documentation.

`strncat()`
The meaning of its buffer size argument is surprising. Again, use `strlcat()`.

`strtok()` Uses global data, so it is unsafe in threaded programs such as kernels. Use `strtok_r()` instead, and see its source code in `lib/string.c` for documentation and an example.

`sprintf()`
Same issue as `strcpy()`. Use `snprintf()` instead. Refer to comments in `lib/stdio.h` for documentation.

`vsprintf()`
Same issue as `strcpy()`. Use `vsnprintf()` instead.

If you try to use any of these functions, the error message will give you a hint by referring to an identifier like `dont_use_sprintf_use_snprintf`.

Appendix D Project Documentation

This chapter presents a sample assignment and a filled-in design document for one possible implementation. Its purpose is to give you an idea of what we expect to see in your own design documents.

D.1 Sample Assignment

Implement `thread_join()`.

`void thread_join (`*tid_t* `tid)` [Function]

Blocks the current thread until thread *tid* exits. If *A* is the running thread and *B* is the argument, then we say that "*A* joins *B*."

Incidentally, the argument is a thread id, instead of a thread pointer, because a thread pointer is not unique over time. That is, when a thread dies, its memory may be, whether immediately or much later, reused for another thread. If thread *A* over time had two children *B* and *C* that were stored at the same address, then `thread_join(B)` and `thread_join(C)` would be ambiguous.

A thread may only join its immediate children. Calling `thread_join()` on a thread that is not the caller's child should cause the caller to return immediately. Children are not "inherited," that is, if *A* has child *B* and *B* has child *C*, then *A* always returns immediately should it try to join *C*, even if *B* is dead.

A thread need not ever be joined. Your solution should properly free all of a thread's resources, including its `struct thread`, whether it is ever joined or not, and regardless of whether the child exits before or after its parent. That is, a thread should be freed exactly once in all cases.

Joining a given thread is idempotent. That is, joining a thread multiple times is equivalent to joining it once, because it has already exited at the time of the later joins. Thus, joins on a given thread after the first should return immediately.

You must handle all the ways a join can occur: nested joins (*A* joins *B*, then *B* joins *C*), multiple joins (*A* joins *B*, then *A* joins *C*), and so on.

D.2 Sample Design Document

```
                        +-----------------+
                        |      CS 3204    |
                        |  SAMPLE PROJECT |
                        | DESIGN DOCUMENT |
                        +-----------------+

---- GROUP ----

Ben Pfaff <blp@stanford.edu>

---- PRELIMINARIES ----
```

```
>> If you have any preliminary comments on your submission, notes for
>> the TAs, or extra credit, please give them here.

(This is a sample design document.)

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation,
>> course text, and lecture notes.

None.

                                JOIN
                                ====

---- DATA STRUCTURES ----

>> Copy here the declaration of each new or changed ‘struct’ or ‘struct’
>> member, global or static variable, ‘typedef’, or enumeration.
>> Identify the purpose of each in 25 words or less.

A "latch" is a new synchronization primitive.  Acquires block
until the first release.  Afterward, all ongoing and future
acquires pass immediately.

    /* Latch. */
    struct latch
      {
        bool released;              /* Released yet? */
        struct lock monitor_lock;   /* Monitor lock. */
        struct condition rel_cond;  /* Signaled when released. */
      };

Added to struct thread:

    /* Members for implementing thread_join(). */
    struct latch ready_to_die;   /* Release when thread about to die. */
    struct semaphore can_die;    /* Up when thread allowed to die. */
    struct list children;        /* List of child threads. */
    list_elem children_elem;     /* Element of ‘children’ list. */

---- ALGORITHMS ----

>> Briefly describe your implementation of thread_join() and how it
>> interacts with thread termination.

thread_join() finds the joined child on the thread’s list of
children and waits for the child to exit by acquiring the child’s
```

```
ready_to_die latch.  When thread_exit() is called, the thread
releases its ready_to_die latch, allowing the parent to continue.

---- SYNCHRONIZATION ----

>> Consider parent thread P with child thread C.  How do you ensure
>> proper synchronization and avoid race conditions when P calls wait(C)
>> before C exits?  After C exits?  How do you ensure that all resources
>> are freed in each case?  How about when P terminates without waiting,
>> before C exits?  After C exits?  Are there any special cases?

C waits in thread_exit() for P to die before it finishes its own
exit, using the can_die semaphore "down"ed by C and "up"ed by P as
it exits.  Regardless of whether whether C has terminated, there
is no race on wait(C), because C waits for P's permission before
it frees itself.

Regardless of whether P waits for C, P still "up"s C's can_die
semaphore when P dies, so C will always be freed.  (However,
freeing C's resources is delayed until P's death.)

The initial thread is a special case because it has no parent to
wait for it or to "up" its can_die semaphore.  Therefore, its
can_die semaphore is initialized to 1.

---- RATIONALE ----

>> Critique your design, pointing out advantages and disadvantages in
>> your design choices.

This design has the advantage of simplicity.  Encapsulating most
of the synchronization logic into a new "latch" structure
abstracts what little complexity there is into a separate layer,
making the design easier to reason about.  Also, all the new data
members are in 'struct thread', with no need for any extra dynamic
allocation, etc., that would require extra management code.

On the other hand, this design is wasteful in that a child thread
cannot free itself before its parent has terminated.  A parent
thread that creates a large number of short-lived child threads
could unnecessarily exhaust kernel memory.  This is probably
acceptable for implementing kernel threads, but it may be a bad
idea for use with user processes because of the larger number of
resources that user processes tend to own.
```

Appendix E Debugging Tools

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

E.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 (but see [printf Reboots], page 44 for a counterexample).

`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 `print()` 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 E.6 [Debugging by Infinite Loop], page 96, for a related technique.

E.2 `ASSERT`

Assertions are useful because they can catch problems early, before they'd otherwise be noticed. Pintos provides the `ASSERT`, defined in '`<debug.h>`', for assertions. 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.

When an assertion proves untrue, the kernel panics. The panic message should help you to find the problem. See the description of backtraces below for more information.

E.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` [Macro]
: 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` [Macro]
: 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` [Macro]
: 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`*`)` [Macro]

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.

E.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.

The addresses in a backtrace are listed as raw hexadecimal numbers, which are meaningless by themselves. You can translate them into function names and source file line numbers using a tool called `addr2line`.

The output format of `addr2line` is not ideal, so we've supplied a wrapper for it simply called `backtrace`. 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` does not correspond to the kernel that produced the backtrace.

Sometimes backtraces can be confusing without implying 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. In the threads project, this is commonly seen in backtraces for test failures; see [`pass()` Fails], page 42), for more information.

E.4.1 Example

Here's an example. Suppose that Pintos printed out this following call stack, which is taken from an actual Pintos submission for the file system project:

```
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: ?? (threads/intr-stubs.S:1554)
0x804812c: ?? (??:0)
0x8048a96: ?? (??:0)
0x8048ac8: ?? (??:0)
```

(You will probably not get the same results if you run the command above on your own kernel binary, because the source code you compiled from is different from the source code that panicked.)

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 grow-too-big 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67
0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
```

The results look like this:

```
0xc0106eff: ?? (??:0)
0xc01102fb: ?? (??:0)
0xc010dc22: ?? (??:0)
0xc010cf67: ?? (??:0)
0xc0102319: ?? (??:0)
0xc010325a: ?? (??:0)
0x804812c: test_main (../../tests/filesys/extended/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.
```

E.5 gdb

You can run the Pintos kernel under the supervision of the `gdb` (80*x*86) or `i386-elf-gdb` (SPARC) debugger. First, start Pintos with the '`--gdb`' option, e.g. `pintos --gdb -- run mytest`. Second, in a separate terminal, invoke `gdb` (or `i386-elf-gdb`) on '`kernel.o`':

```
gdb kernel.o
```

and issue the following `gdb` command:

```
target remote localhost:1234
```

(If the `target remote` command fails, then make sure that both `gdb` and `pintos` are running on the same machine by running `hostname` in each terminal. If the names printed differ, then you need to open a new terminal for `gdb` on the machine running `pintos`.)

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⟩. If you want `gdb` to stop when Pintos starts running, set a breakpoint on `main()` with the command `break main` before '`c`'.

You can read the `gdb` manual by typing `info gdb` at a terminal command prompt, or you can view it in Emacs with the command *`C-h i`*. Here's a few commonly useful `gdb` commands:

`c`
: Continues execution until ⟨Ctrl+C⟩ or the next breakpoint.

`break` *`function`*
`break` *`filename`*`:`*`linenum`*
`break *`*`address`*
: Sets a breakpoint at the given function, line number, or address. (Use a '`0x`' prefix to specify an address in hex.)

`p` *`expression`*
: Evaluates the given C 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 the given 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 the given address. (Use a '`0x`' prefix to specify an address in hex.)

`diassemble` *`function`*
: Disassembles the specified *function*.

If you notice other strange behavior while using `gdb`, there are three possibilities: a bug in your modified Pintos, a bug in Bochs's interface to `gdb` or in `gdb` itself, or a bug in the original Pintos code. The first and second are quite likely, and you should seriously consider both. We hope that the third is less likely, but it is also possible.

You can also use `gdb` to debug a user program running under Pintos. Start by issuing this `gdb` command to load the program's symbol table:

```
add-symbol-file program
```

where *program* is the name of the program's executable (in the host file system, not in the Pintos file system). 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. 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`'.)

E.6 Debugging by Infinite Loop

If you get yourself into a situation where the machine reboots in a loop, that's probably a "triple fault." In such a 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. In such a situation, you might want to try what I call "debugging by infinite loop."

What you do is pick a place in the Pintos code, insert the statement `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.

E.7 Modifying Bochs

An advanced debugging technique is to modify and recompile the simulator. This proves useful when the simulated hardware has more information than it makes available to the OS. For example, page faults have a long list of potential causes, but the hardware does not report to the OS exactly which one is the particular cause. Furthermore, a bug in the kernel's handling of page faults can easily lead to recursive faults, but a "triple fault" will cause the CPU to reset itself, which is hardly conducive to debugging.

In a case like this, you might appreciate being able to make Bochs print out more debug information, such as the exact type of fault that occurred. It's not very hard. You start by retrieving the source code for Bochs 2.2.5 from `http://bochs.sourceforge.net` and extracting it into a directory. If desired, apply

'`pintos/src/misc/bochs-2.2.5.jitter.patch`'. Then run '`./configure`', supplying the options you want (some suggestions are in the patch file). Finally, run `make`. This will compile Bochs and eventually produce a new binary '`bochs`'. To use your '`bochs`' binary with `pintos`, put it in your `PATH`, and make sure that it is earlier than '`/home/courses/cs3204/bin/bochs`'.

Of course, to get any good out of this you'll have to actually modify Bochs. Instructions for doing this are firmly out of the scope of this document. However, if you want to debug page faults as suggested above, a good place to start adding `printf()`s is `BX_CPU_C::dtranslate_linear()` in '`cpu/paging.cc`'.

E.8 Tips

The page allocator in '`threads/palloc.c`' and the block allocator in '`threads/malloc.c`' both clear all the bytes in pages and blocks to `0xcc` when they are freed. 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)`, 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.

Appendix F Development Tools

Here are some tools that you might find useful while developing code.

F.1 Tags

Tags are an index to the functions and global variables declared in a program. Many editors, including Emacs and `vi`, can use them. The '`Makefile`' in '`pintos/src`' produces Emacs-style tags with the command `make TAGS` or `vi`-style tags with `make tags`.

In Emacs, use *M-.* to follow a tag in the current window, *C-x 4 .* in a new window, or *C-x 5 .* in a new frame. If your cursor is on a symbol name for any of those commands, it becomes the default target. If a tag name has multiple definitions, *M-0 M-.* jumps to the next one. To jump back to where you were before you followed the last tag, use *M-**.

F.2 CVS

CVS is a version-control system. That is, you can use it to keep track of multiple versions of files. The idea is that you do some work on your code and test it, then check it into the version-control system. If you decide that the work you've done since your last check-in is no good, you can easily revert to the last checked-in version. Furthermore, you can retrieve any old version of your code as of some given day and time. The version control logs tell you who made changes and when.

CVS is not the best version control system out there, but it's free and is fairly easy to use.

For more information, visit the CVS home page.

F.2.1 Setting Up CVS

To set up CVS for use with Pintos, start by choosing one group member as the keeper of the CVS repository. Everyone in the group will be able to use the CVS repository, but the keeper will actually create the repository and maintain permissions for its contents.

The following instructions are specific to our local setup for the Spring 2006 semester. Even if you've used CVS before, we ask that you read the instructions in their entirety.

Repositories must be created on the machine 'fortran.cslab'. This machine contains a directory that was specially set up for CS 3204 students' CVS repositories. To access the repository from the other machines, you should first configure ssh to log you on automatically, without requiring a password every time. See Section F.2.4 [Setting Up ssh], page 101, for more information. To connect to this machine use '`ssh fortran`' from any of the machines in McB 124. You should not be prompted for a password if you have configured ssh properly.

The keeper has to perform several steps to set up the repository. First, log on to 'fortran.cslab' and create a directory for storing the repository. The new directory must be created in the directory '`/home/cs3204`' and should be named '`Proj-keeper_pid`', where *keeper_pid* is the pid of the keeper. Next, configure access to repository using the command '`setfacl --set u::rwx,g:::---,o:::--- Proj-keeper_pid`'. This command ensures that the user, i.e the keeper has the required permissions to access the repository, and no one else does. To set permissions for the other members in the group, use '`setfacl -m`

`u:`*`member_pid`*`:rwx Proj-`*`keeper_pid`*' for each of the other members in the group, replacing *member_pid* with the pid of the group member.

Next, set the permissions of the directories and files that would be created inside the repository using the command '`setfacl -d --set u::rwx,g::---,o::--- Proj-`*`keeper_pid`*'. To permit all the members of the group access to all the files and directories created in the repository, use '`setfacl -d -m u:`*`member_pid`*`:rwx Proj-`*`keeper_pid`*' once for each group member (should be used once for the keeper too), replacing *member_pid* with the pid of the group member. To make sure that the permissions are set correctly, use '`getfacl Proj-`*`keeper_pid`*'.

Note that neither (Unix-) group members nor others should have read access to your CVS repository, hence the '`g::---,o::---`' part of the access control list. (Giving access to group members (in the Unix sense) would give access to, for instance, all CS majors if your default (Unix-) group is Major. We use ACLs to give individual access to your CS 3204 group members.) **Failing to protect your repository in this way is an honor code violation.**

Now initialize the repository. To initialize the repository, execute '`cvs -d /home/cs3204/Proj-`*`keeper_pid`* `init`'.

Finally, import the Pintos sources into the newly initialized repository. If you have an existing set of Pintos sources you want to add to the repository, '`cd`' to its '`pintos`' directory now. Otherwise, to import the base Pintos source tree, '`cd`' to '`/home/courses/cs3204/pintos/pintos`' (note the doubled '`pintos`'). After changing the current directory, execute this command:

```
cvs -d /home/cs3204/Proj-keeper_pid import -m "Imported sources" pintos foobar start
```

Here is a summary of the commands you have now executed:

```
ssh fortran
cd /home/cs3204
mkdir Proj-keeper_pid
setfacl --set u::rwx,g::---,o::--- Proj-keeper_pid
# for all other group members do:
setfacl -m u:member-pid:rwx Proj-keeper_pid
setfacl -d --set u::rwx,g::---,o::--- Proj-keeper_pid
# for all group members, including the keeper, do:
setfacl -d -m u:member_pid:rwx Proj-keeper_pid
cvs -d /home/cs3204/Proj-keeper_pid init
cd /home/courses/cs3204/pintos/pintos
cvs -d /home/cs3204/Proj-keeper_pid import -m "Imported sources" pintos foobar start
```

The repository is now ready for use by any group member, as described below. Having set the repository up, you need not log on to 'fortran.cslab' for any other purposes. Keep in mind that the repository should only be accessed using CVS commands—it is not generally useful to examine the repository files by hand, and you should definitely not modify them yourself.

Due to space constraints, 'fortran.cslab'should be used only to store the repository and not for development purposes. Do not store any other files there and do not run any other programs on this machine. The reason for this somewhat unusual setup is that our shared file servers currently do not support the '`setfacl`' commands, making it impossible to protect your CVS repository.

F.2.2 Using CVS

Some of the CVS commands require you to specify the location of the repository. As the repository has been set up in the machine 'fortran.cslab' and you would not be using this machine for development purposes, you have to use '`:ext:`*`your_pid`*`@fortran:/home/cs3204/Proj-`*`keeper_pid`*' as the location of the repository. *your_pid* is your pid and is needed to log you on to 'fortran.cslab'. CVS runs on top of ssh. Therefore, before using any of the CVS commands, make sure you have configured ssh to log you on without prompting for password (See Section F.2.4 [Setting Up ssh], page 101, for more information) and set the environment variable CVS_RSH to '`/usr/bin/ssh`'. Under csh you can set this environment variable using '`setenv CVS_RSH /usr/bin/ssh`'. To avoid having to type this line everytime you log on, add this line to the '.cshrc' file in your home directory.

To use CVS, start by checking out a working copy of the contents of the CVS repository into a directory named '*`dir`*'. To do so, execute '`cvs -d :ext:`*`your_pid`*`@fortran:/home/cs3204/Proj-`*`keeper_pid checkout -d `*`dir`*` pintos`'. If this fails due to some kind of permission problem, the CVS repository may not be initialized properly.

Note that there are two '`-d`' switches in the previous command. The first switch specifies the location of the CVS repository to which the command applies. In this case, the repository is located on the machine *fortran* and is reachable via ssh with your_pid. The second '`-d`' switch is specific to the cvs checkout command. It specifies the local directory into which to check out the module 'pintos'. If omitted, pintos will be checked out into a directory called 'pintos'.

Your working copy is kept in your undergrad file space. Unlike the CVS repository, this directory is shared among the lab machines, so you do not need to be logged on to any specific machine to use it. Like the CVS repository, you must read-protect your working copy from (Unix-) group members and others to comply with the honor code. '`chmod -R go-rwx `*`dir`*' will read-protect your working directory.

At this point, you can modify any of the files in the working copy. You can see the changes you've made with '`cvs diff -u`'. If you want to commit these changes back to the repository, making them visible to the other group members, you can use the CVS commit command. Within the '`pintos`' directory, execute '`cvs commit`'. This will figure out the files that have been changed and fire up a text editor for you to describe the changes. By default, this editor is '`vi`', but you can select a different editor by setting the `CVSEDITOR` environment variable, e.g. with '`setenv CVSEDITOR emacs`' (add this line to your '`.cshrc`' to make it permanent).

Suppose another group member has committed changes. You can see the changes committed to the repository since the time you checked it out (or updated from it) with '`cvs diff -u -r BASE -r HEAD`'. You can merge those change into your working copy using '`cvs update`'. If any of your local changes conflict with the committed changes, the CVS command output should tell you. In that case, edit the files that contain conflicts, looking for '`<<<`' and '`>>>`' that denote the conflicts, and fix the problem.

You can view the history of *file* in your working directory, including the log messages, with '`cvs log `*`file`*'.

You can give a particular set of file versions a name called a *tag*. First '`cd`' to the root of the working copy, then execute '`cvs tag` *`name`*'. It's best to have no local changes in the working copy when you do this, because the tag will not include uncommitted changes. To recover the tagged repository later, use the '`checkout`' command in the form '`cvs -d :ext:`*`your_pid`*`@fortran:/home/cs3204/Proj-`*`keeper_pid` `checkout -r` *`tag`* `-d` *`dir`* `pintos`', where *dir* is the directory to put the tagged repository into.

If you add a new file to the source tree, you'll need to add it to the repository with '`cvs add` *`file`*'. This command does not have lasting effect until the file is committed later with '`cvs commit`'.

To remove a file from the source tree, first remove it from the file system with `rm`, then tell CVS with '`cvs remove` *`file`*'. Again, only '`cvs commit`' will make the change permanent.

To discard your local changes for a given file, without committing them, use '`cvs update -C` *`file`*'.

To check out a version of your repository as of a particular date, use the command '`cvs -d :ext:`*`your_pid`*`@fortran:/home/cs3204/Proj-`*`keeper_pid` `checkout -D 'date' -d` *`dir`* `pintos`', where *dir* is the directory to put the tagged repository into. A typical format for *date* is '`YYYY-MM-DD HH:MM`', but CVS accepts several formats, even something like '`1 hour ago`'.

For more information, visit the CVS home page.

If you are using an IDE, check whether it supports CVS automatically.

F.2.3 CVS Locking

You might occasionally see a message like this while using CVS:

```
waiting for member_pid's lock in /home/cs3204/Proj-keeper_pid/cvsroot/foo
```

This normally means that more than one user is accessing the repository at the same time. CVS should automatically retry after 30 seconds, at which time the operation should normally be able to continue.

If you encounter a long wait for a lock, of more than a minute or so, it may indicate that a CVS command did not complete properly and failed to remove its locks. If you think that this is the case, ask the user in question about it. If it appears that an operation did go awry, then you (or the named user) can delete files whose names start with '`#cvs.rfl`', '`#cvs.wfl`', or '`#cvs.lock`' in the directory mentioned in the message. Doing so should allow your operation to proceed. Do not delete or modify other files.

F.2.4 Setting Up ssh

Ssh can be configured to log you on to any of the machines in McB 124 from any other machine in McB 124, without you having to enter your password. To enable automatic login, perform the following steps after logging on to any of the machines in McB 124.

- '`ssh-keygen -t rsa -N ""`' On your screen you should see something similar to what is shown below.

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ugrads/your_pid/.ssh/id_rsa):
Your identification has been saved in /home/ugrads/your_pid/.ssh/id_rsa.
Your public key has been saved in /home/ugrads/your_pid/.ssh/id_rsa.pub.
```

```
The key fingerprint is:
34:45:6d:4a:51:4e:1f:af:fe:66:dd:a9:a5:23:46:bb your_pid@some_machine.cslab
```

Accept the defaults. This command creates a new file '`id_rsa.pub`' in the directory '`$HOME/.ssh`' if the default location is chosen.

- '`cd $HOME/.ssh`'
- '`cat id_rsa.pub >> authorized_keys`'
- '`cd $HOME`'
- '`chmod +700 .ssh`'

To make sure that you have configured it correctly, try ssh'ing to another machine in McB 124 (e.g, '`ssh fortran`'). You should not be prompted for your password. If it is the first time you are ssh'ing to some machine, you might have to type '`yes`' to continue connecting.

F.3 VNC

VNC stands for Virtual Network Computing. It is, in essence, a remote display system which allows you to view a computing "desktop" environment not only on the machine where it is running, but from anywhere on the Internet and from a wide variety of machine architectures. It is already installed on the machines in McB 124. For more information, look at the VNC Home Page.

F.4 Cygwin

Cygwin provides a Linux-compatible environment for Windows. It includes ssh and an X11 server, Cygwin/X. If your primary work environment is Windows, you will find Cygwin/X extremely useful for these projects. Install Cygwin/X, then start the X server and open a new xterm. Use '`ssh -XY your_pid@rlogin.cs.vt.edu`' to log on to the lab machines. The X11 server will allow you to run pintos while displaying the bochs-emulated console in your windows desktop. In addition, you can set up Cygwin's ssh client for password-less login as described earlier. See Section F.2.4 [Setting Up ssh], page 101.