Pintos

by Ben Pfaff
USF modifications by Greg Benson

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 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. However, it is much easier to develop and debug kernel code using an x86 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.

These projects are hard. As you know CS 326 has a reputation of taking a lot of time. We will do what I 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 will need to log into one of the USF CS Linux machines. Currently, we only support Pintos development on our PCs running Linux. Although we do have Bochs and Qemu installed on the Macs, we currently do not have a gcc cross compiler installed that will generate x86 binaries. We will test your code on these machines, and the instructions given here assume this environment. However, Pintos and its supporting tools are portable enough that it should build "out of the box" in other environments (e.g. on your own laptop or desktop).

Once you've logged into one of these machines, either locally or remotely, start out by adding the CS 326 bin directory to your `PATH` environment. Under `bash`, the USF CS login shell, you can do so with this command:

```
export PATH = $PATH:/home/public/cs326/bin
```

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

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/public/cs326/pintos/pintos.tar.gz
```

Alternatively, fetch http://www.cs.usfca.edu/benson/cs326/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 USF CS 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 30 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 Project 1 for details on how to add source files.

‘`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 39) 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 does all the dirty work needed to configure the simulator to run your Pintos disk image. For example, it handles command-line argument passing to the kernel by copying the arguments directly into the disk image so that the Pintos kernel can retrieve them. By default, this command uses Qemu as the emulator. If you want to run without a graphics display, use the `-v` option:

```
pintos -v os.dsk -- run alarm-multiple
```

Once the test is finished you can exit Qemu by typing `CTRL-A C`.

If Bochs is selected, the pintos 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 as shown above 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 41). 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 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.

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, algorithms, synchronization, and rationale. An incomplete 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 35, for a sample design document for a fictitious project.

Design quality will also be judged based on 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 design quality 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 the self-consistent at the very least. Use horizontal and vertical white space to make code readable. Add a comment to 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. 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 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 USF's CS 326 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`.

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 17) 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 80x86 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.

Hint: try using "tags" to follow along with references to function and variable names (see Section F.1 [Tags], page 44).

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 21), 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

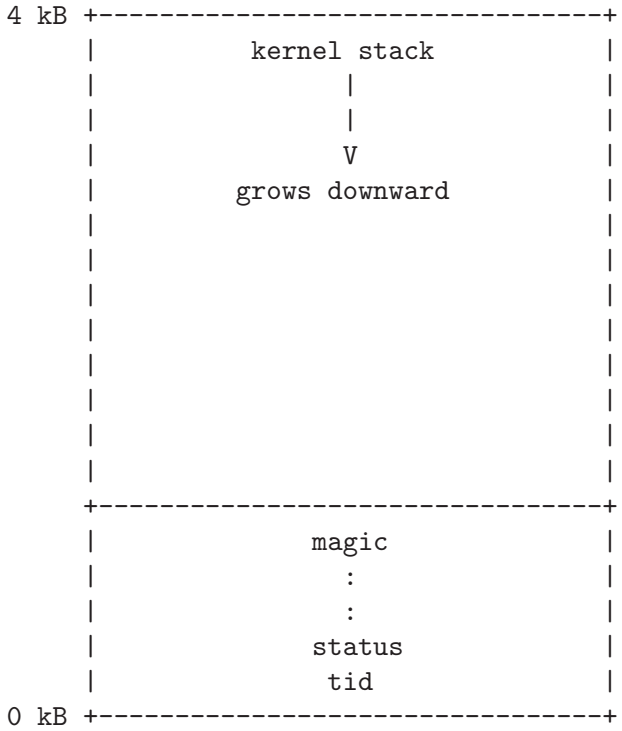
2.2.1.1 `struct thread`

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 23).

`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 20, 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.

`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 22).

`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.

`void 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 15).

`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 38).

`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.

`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 27.

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.

The actual `schedule()` implementation is simple. 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 in Pintos, 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 handlers, which cannot sleep and thus cannot use most other forms of synchronization (see Section 2.2.3.3 [External Interrupt Handling], page 22).

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

[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.

`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 15) 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 16). 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 15).
- *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 various 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 80x86 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 15).

`void intr_register_int (`*uint8_t* `vec`, *int* `dpl`, *enum intr_level* `level`, *intr_handler_func* *`handler`, *const char* *`name``)` [Function]
Registers *func* 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 it can be invoked at any time that interrupts have not been enabled. 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 15) 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 Project 3). An allocation request draw 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 43).

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 43).

The block allocator may not be called from interrupt context.

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 3: System Programming Guide. Operating system support, including segmentation, paging, tasks, interrupt and exception handling.

[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.

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 27). 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

$$\textit{priority} = (\textit{recent\_cpu} / 4) + (\textit{nice} * 2),$$

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:

$$\begin{gathered} x(0) = f(0), \\ x(t) = ax(t-1) + (1-a)f(t), \\ a = k/(k+1), \end{gathered}$$

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:

$$\begin{gathered} x(1) = f(1), \\ x(2) = af(1) + f(2), \\ \vdots \\ x(5) = a^4 f(1) + a^3 f(2) + a^2 f(3) + af(4) + f(5). \end{gathered}$$

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 $t = \log_a w$.

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. In addition, once per second the value of *recent_cpu* is recalculated for every thread (whether running, ready, or blocked), using this formula:

$$\textit{recent\_cpu} = (2*\textit{load\_avg})/(2*\textit{load\_avg} + 1) * \textit{recent\_cpu} + \textit{nice},$$

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.

Because of assumptions made by some of the tests, *recent_cpu* 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.

Take note that *recent_cpu* can be a negative quantity for a thread with a negative *nice* value. Negative values of *recent_cpu* are not changed to 0.

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 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 10 bits of a signed 32-bit integer as fractional bits, so that an integer x represents the real number $x/2^{10}$. This is called a 21.10 fixed-point number representation, because there are 21 bits before the decimal point, 10 bits after it, and one sign bit.[1] A number in 21.10 format represents, at maximum, a value of $(2^{31} - 1)/2^{10} \approx 2{,}097{,}151.999$.

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

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 21.10 format the fraction 59/60 used in the calculation of *load_avg*, above, is $(59/60)2^{10} = 1{,}007$ (rounded to nearest). To convert a fixed-point value back to an integer, divide by f. (The normal '`/`' operator in C rounds down. To round to nearest, add $f/2$ 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 $1{,}007 \times 1{,}007 = 1{,}014{,}049$ is much greater than $2^{10} = 1{,}024$. Shifting q bits right, we get $1{,}014{,}049/2^{10} = 990$, or about 0.97, the correct answer. Second, the multiplication can overflow even though the answer is representable. For example, 128 in 21.10 format is $128 \times 2^{10} = 131{,}072$ and its square $128^2 = 16{,}384$ is well within the 21.10 range, but $131{,}072^2 = 2^{34}$, 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 the opposite complications. 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`.

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 down):	`x / f`
Convert `x` to integer (rounding to nearest):	`(x + f / 2) / f`
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

Please make sure you use a consistent coding styles

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 26).

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 `int`*`n`*`_t` and `uint`*`n`*`_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 `int`*`n`*`_t` defined by '`<stdint.h>`', it provides macros `PRId`*`n`* and `PRIi`*`n`* for formatting values of that type with `"%d"` and `"%i"`. Similarly, for every `uint`*`n`*`_t`, it provides `PRIo`*`n`*, `PRIu`*`n`*, `PRIu`*`x`*, and `PRIu`*`X`*.

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 326     |
                        |  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 in Project 1 for a counter example).

`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 42, 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` (80*x*86) or `i386-elf-addr2line` (SPARC).

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. For example, 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.

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
```

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 2.1.1 from http://bochs.sourceforge.net and extracting it into a directory. Then read '`pintos/src/misc/bochs-2.1.1.patch`' and apply the patches needed. 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 '`/usr/class/cs140/`uname -m`/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-**.