CS 162 Project 0: `Introduction to Pintos`

Code and Report Due: | Monday, February 1, 2021 at 11:59 PM PST

Contents

1 Introduction

Our projects in CS 162 will use Pintos, an educational operating system. They're designed to give you practical experience with the central ideas of operating systems in the context of developing a real, working kernel, without being excessively complex. The skeleton code for Pintos has several limitations in its file system, thread scheduler, and support for user programs. In the course of these projects, you will greatly improve Pintos in each of these areas.

Our project specifications in CS 162 will be organized as follows. For clarity, the details of the assignment itself will be in the Your Task section at the start of the document. We will also provide additional material in the Reference section that will hopefully be useful as you design and implement a solution to the project. You may find it useful to begin with the Reference section for an overview of Pintos before trying to understand all of the details of the assignment in Your Task.

In this exercise you will learn more about some of the basics of Pintos, such as debugging, and will be given an opportunity to fix a bug in Pintos. **Unlike future projects, you will turn in your own code and report**. This project is meant to give you a feel for how Pintos is structured and how to use some general debugging tools.

1.1 Getting Started

You will be working in the `~/code/personal/proj0` directory for this project. Log in to your VM and grab the Pintos skeleton code from the staff repository:

```
cd ~/code/personal
git pull staff master
cp ~/code/personal/proj0/src/utils/pintos-test ~/.bin/
```

2 Your Task

In order to design good solutions for future Pintos projects, you'll first need a good understanding of the existing codebase. The goal of this exercise is to help you start developing some familiarity with Pintos' code.

2.1 Find the Faulting Instruction

First, run `make` and `make check` in the `proj0/src/userprog` directory, and observe that currently no tests pass. We will step through the execution of the `do-nothing` test in GDB to learn how we can modify Pintos so that the test passes, and understand how Pintos' existing support for user programs is implemented.

We are using `do-nothing` because it is the simplest test of Pintos' user program support. You should read `proj0/src/tests/userprog/do-nothing.c`; it is a Pintos user application that does nothing. Its `main()` function is merely the statement `return 162;`, indicating that it returns the exit code `162` to the operating system. The specific value of the exit code is immaterial to the test; we chose a value other than `0` so that it's easier to track how the Pintos kernel handles this value through GDB (note `162` = `0xa2`). When you ran `make`, `do-nothing.c` was compiled to create an executable program `do-nothing`, which you can find at `proj0/src/userprog/build/tests/userprog/do-nothing`. The `do-nothing` test simply runs the `do-nothing` executable in Pintos.

View the file `proj0/src/userprog/build/tests/userprog/do-nothing.result`. This file shows the output of the Pintos testing framework when running the `do-nothing` test. The testing framework expected Pintos to output "`do-nothing: exit(162)`". This is the standard message that Pintos prints when a process exits (you'll encounter this again in the "System Calls" section of the Project 1 spec). However, as shown in the diff, Pintos did not output this message; instead, the `do-nothing` program

crashed in userspace due to a memory access violation (a segmentation fault). Based on the contents of the `do-nothing.result` file, please answer the following questions on Gradescope:

1. What virtual address did the program try to access from userspace that caused it to crash?
2. What is the virtual address of the instruction that resulted in the crash?
3. To investigate, disassemble the `do-nothing` binary using `objdump` (you used this tool in Homework 0). What is the name of the function the program was in when it crashed? Copy the disassembled code for that function onto Gradescope, and identify the instruction at which the program crashed.
4. Find the C code for the function you identified above (hint: it was executed in userspace, so it's either in `do-nothing.c` or one of the files in `proj0/src/lib` or `proj0/src/lib/user`), and copy it onto Gradescope. For each instruction in the disassembled function in #3, explain in a few words why it's necessary and/or what it's trying to do. Hint: see 80x86 Calling Convention.
5. Why did the instruction you identified in #3 try to access memory at the virtual address you identified in #1? Don't explain this in terms of the values of registers; we're looking for a higher-level explanation.

2.2 Step Through the Crash

Now that we understand why the `do-nothing` program crashes, using GDB we will step through the execution of the `do-nothing` test in Pintos, starting from when the kernel boots. Our goal is to find out how we can modify the Pintos user program loader so that `do-nothing` does not crash, while becoming acquainted with how Pintos supports user programs. To do this, change your working directory to `proj0/src/userprog/` and run

```
FORCE_SIMULATOR=--bochs PINTOS_DEBUG=1 pintos-test do-nothing
```

GDB should now be open. Type `debugpintos`, then hit Enter in order to connect to the Pintos process. When you first run `debugpintos`, the processor's execution has not yet started. At a high level, the following must happen before Pintos can start the `do-nothing` process.

- The BIOS reads the Pintos bootloader (`proj0/src/threads/loader.S`) from the first sector of the disk into memory at address `0x7c00`.
- The bootloader reads the kernel code from disk into memory at address `0x20000` and then jumps to the kernel entrypoint (`proj0/src/threads/start.S`).
- The code at the kernel entrypoint switches to 32-bit protected mode[1] and then calls `main()` (`proj0/src/threads/init.c`).
- The `main()` function boots Pintos by initalizing the scheduler, memory subsystem, interrupt vector, hardware devices, and file system.

You're welcome to read the code to learn more about this setup, but **you don't need to understand how this works for the Pintos projects or for this class**. Set a breakpoint at `run_task` and continue in GDB to skip the setup. As you can see in the code for `run_task`, Pintos executes the `do-nothing` program (specified on the Pintos command line), by invoking

```
process_wait(process_execute("do-nothing"));
```

from `run_task()`. Both `process_wait` and `process_execute` are in `proj0/src/userprog/process.c`.

Now, answer the following questions.

[1] `https://en.wikipedia.org/wiki/Protected_mode`

6. Step into the `process_execute` function. What is the name and address of the thread running this function? What other threads are present in Pintos at this time? Copy their `struct thread`s. (Hint: for the last part `dumplist &all_list thread allelem` may be useful.)

7. What is the backtrace for the current thread? Copy the backtrace from GDB as your answer and also copy down the line of C code corresponding to each function call.

8. Set a breakpoint at `start_process` and continue to that point. What is the name and address of the thread running this function? What other threads are present in Pintos at this time? Copy their `struct thread`s.

9. Where is the thread running `start_process` created? Copy down this line of code.

10. Step through the `start_process()` function until you have stepped over the call to `load()`. Note that `load()` sets the `eip` and `esp` fields in the `if_` structure. Print out the value of the `if_` structure, displaying the values in hex (hint: `print/x if_`).

11. The first instruction in the `asm volatile` statement sets the stack pointer to the bottom of the `if_` structure. The second one jumps to `intr_exit`. The comments in the code explain what's happening here. Step into the `asm volatile` statement, and then step through the instructions. As you step through the `iret` instruction, observe that the function "returns" into userspace. Why does the processor switch modes when executing this function? Feel free to explain this in terms of the values in memory and/or registers at the time `iret` is executed, and the functionality of the `iret` instruction.

12. Once you've executed `iret`, type `info registers` to print out the contents of registers. Include the output of this command on Gradescope. How do these values compare to those when you printed out `if_`?

13. Notice that if you try to get your current location with `backtrace` you'll only get a hex address. This is because because the debugger only loads in the symbols from the kernel. Now that we are in userspace, we have to load in the symbols from the Pintos executable we are running, namely `do-nothing`. To do this, use `loadusersymbols tests/userprog/do-nothing`. Now, using `backtrace`, you'll see that you're currently in the `_start` function. Using the `disassemble` and `stepi` commands, step through userspace instruction by instruction until the page fault occurs. At this point, the processor has immediately entered kernel mode to handle the page fault, so `backtrace` will show the current stack in kernel mode, not the user stack at the time of the page fault. However, you can use `btpagefault` to find the user stack at the time of the page fault. Copy down the output of `btpagefault`.

2.3 Debug

The faulting instruction you observed in GDB should match the one you found in #3. Now that you have determined the faulting instruction, understood the purpose of the instruction, and walked through how the kernel initializes a user process, you are in a position to modify the kernel so that `do-nothing` runs correctly.

14. Modify the Pintos kernel so that `do-nothing` no longer crashes. Your change should be in the Pintos kernel, not the userspace program (`do-nothing.c`) or libraries in `proj0/src/lib`. This should not involve extensive changes to the Pintos source code. Our staff solution solves this with a single-line change to `process.c`. Explain the change you made to Pintos and why it was necessary. After making this change, the `do-nothing` test should pass but all others will still fail.

15. It is possible that your fix also works for the `stack-align-0` test, but there are solutions for `do-nothing` that do not. Take a look at the `stack-align-0` test. It behaves similarly to `do-nothing`, but it returns the value of `esp % 16`. Write down what this program should return (hint: this can be found in `stack-align-0.ck`) as well as why this is the case. You may wish to review stack alignment from Section 0[2].) Then make sure that your previous fix for `do-nothing` also passes `stack-align-0`.

16. Re-run GDB as before. Execute the `loadusersymbols` command, set a breakpoint at `_start`, and continue, to skip directly to the beginning of userspace execution. Using the `disassemble` and `stepi` commands, execute the `do-nothing` program instruction by instruction until you reach the `int $0x30` instruction in `proj0/src/lib/user/syscall.c`. At this point, print the top two words at the top of the stack by examining memory (hint: `x/2xw $esp`) and copy the output.

17. The `int $0x30` instruction switches to kernel mode and pushes an interrupt stack frame onto the kernel stack for this process. Continue stepping through instruction-by-instruction until you reach `syscall_handler`. What are the values of `args[0]` and `args[1]`, and how do they relate to your answer to the previous question?

Now, you can continue stepping through Pintos. Having completed running `do-nothing`, Pintos will proceed to shut down because we provided the `-q` option on the kernel command line. You can step through this in GDB if you're curious how Pintos shuts down.

Congratulations! You've walked through Pintos starting up, running a user program to completion, and shutting down, in GDB. Hopefully this guided exercise helped you get acquainted with Pintos. **Be sure to push your code, with the small change you made in order to make the `do-nothing` and `stack-align-0` tests pass, to GitHub. You should now receive full credit for the `proj0` assignment on the autograder.**

3 Deliverables

3.1 Answers

Write your answers to the 17 questions on Gradescope.

3.2 `do-nothing`

You should receive full credit for the `proj0` assignment on the autograder after making the small fix necessary. This means passing both the `do-nothing` and `stack-align-0` tests.

4 Reference

4.1 Pintos

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

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

[2]https://cs162.eecs.berkeley.edu/static/sections/section0.pdf#page=9

give you the ability to inspect and debug an operating system while it runs. In class we will use the Bochs[3] and QEMU[4] simulators.

4.1.1 Getting Started

Log in to the Vagrant Virtual Machine that you set up in hw0. You should already have a copy of the Pintos skeleton code in `~/code/personal/proj0` on your VM. If you don't see the `proj0` folder, run `git pull staff master` to get the latest copy of the starter code.

Once you have made some progress on your project, you can push your code to the autograder just by running `git push origin master` (or just `git push` for short). For example:

```
$ git commit -m "Added feature X to Pintos"
$ git push origin master
```

To compile Pintos and run the Project 1 tests:

```
$ cd ~/code/personal/proj0/src/userprog
$ make
$ make check
```

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

4.1.2 Overview of the Pintos Source Tree

The Pintos source code in `~/code/personal/proj0/src/` is organized into the following subdirectories:

`threads/`
The base Pintos kernel, including the bootloader, kernel entrypoint, base interrupt handler, page allocator, subpage memory allocator, and CPU scheduler. Most of your code for Project 2 will be in this directory. You will also have to make some modifications in this directory for Project 1.

`userprog/`
Pintos user program support, including management of page/segment tables, handlers for system calls, page faults, and other traps, and the program loader. Most of your code for Project 1 will be in this directory.

`vm/`
We will not use this directory.

`filesys/`
The Pintos file system. You will use this file system in Project 1 and modify it in Project 3.

`devices/`
Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in Project 2.

`lib/`
An implementation of a subset of the C standard library. The code in this directory is compiled into

[3]https://en.wikipedia.org/wiki/Bochs
[4]https://en.wikipedia.org/wiki/QEMU

both the Pintos kernel and user programs that run inside it. You can include header files from this directory using the `#include <...>` notation.[5] You should not have to modify this code.

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

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

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

`examples/`
Example user programs that can run on Pintos. Once you complete Project 1, some of these programs can run on Pintos.

`misc/`, `utils/`
These files help you run Pintos. You should not need to interact with them directly.

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

4.1.3 Building Pintos

For this project, you should build Pintos by running `make` in the `userprog` directory. This section describes the interesting files inside `build` directory, which appears when you run make as above. In later projects, where you will run `make` in the `threads` and `filesys` directories, these files will appear in `threads/build` and `userprog/build`, respectively.

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

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

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

[5] The `#include <...>` notation causes the compiler to search for the file in the include paths specified with `-I` at compile time and system paths. In contrast, the `#include "..."` notation causes the compiler to search for the file in the current directory first, before searching in the include paths and system paths.

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

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

4.1.4 Using the File System from the Pintos Kernel

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

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

4.1.5 Running Pintos Tests

To run an individual test from the Pintos test suite, first enter the relevant subdirectory of `src/` for the project you are working on, which, in this case is `userprog/`. Make sure you have built the relevant executables by running `make`, then run `pintos-test` with the name of the test you would like to run. For example, to run the `stack-align-0` test, you would do the following:

```
cd ~/code/personal/proj0/src/userprog
make
pintos-test stack-align-0
```

Alternatively, if you've forgotten the exact name of the test you'd like to run, or would like to browse through the available tests, you can run `pintos-test` with just *part* of a test's name, and then use the interactive menu to select which test to actually run. For example, try substituting `pintos-test align` for `pintos-test stack-align-0` in the previous example. You'll be presented with a menu containing all tests that fuzzily matched the argument you gave to `pintos-test`, which in this case should show you

all of the `stack-align-N` tests, since the argument provided was `align`. You can type more characters to narrow down the selection further (for example type '2' and you'll see the results are narrowed down to just `stack-align-2`), as well as using the up/down arrow keys to scroll through the options. Once you've selected the test you'd like to run, simply press Enter and the test will be run.

4.1.6 Debugging Pintos Tests

Running the debugger for a individual Pintos test is *very* similar to just running the test. The steps for doing so are nearly identical to those given in 4.1.5, with the only difference being that you'll need to add the `PINTOS_DEBUG` environment variable.
For example, to debug the `stack-align-0` test, you would do the following:

```
cd ~/code/personal/proj0/src/userprog
make
PINTOS_DEBUG=1 pintos-test stack-align-0
```

This will open GDB in your current terminal. Simply enter `debugpintos` to connect to the Pintos process, then proceed with setting breakpoints, continuing, etc. When you are done debugging, type `quit` in GDB, and you then will see whatever output the test created while it was running.

When you want to debug a test, running `pintos-test` with `PINTOS_DEBUG=1` is what you want to do 99% of the time. Rarely however you will want to see what the test is outputting *while* you are stepping through it in the debugger. To do this, the instructions are slightly different. Most of the above will remain the same, except that you will set `PINTOS_DEBUG=2` (note **2** instead of 1). When you run `pintos-test` with `PINTOS_DEBUG=2`, GDB will *not* open in your current terminal, instead the output of the test will be shown live in your current terminal. You'll need to open a second terminal (either through `tmux` or by just ssh'ing into your VM again in a separate terminal window), navigate to the `build/` subdirectory of the directory you ran `pintos-test` in, then run `pintos-gdb kernel.o`. For example to debug the `open-twice` test while being able to observe its live output, you would do the following:

```
cd ~/code/personal/proj0/src/userprog
make
PINTOS_DEBUG=2 pintos-test open-twice
```

Then open a second terminal, and do the following:

```
cd ~/code/personal/proj0/src/userprog/build
pintos-gdb kernel.o
```

In your second terminal, GDB should now be open. Enter `debugpintos` to connect to the Pintos process, and proceed with debugging as usual. The only difference is that now the live output of the process you're debugging will appear in your first terminal.

4.1.7 Debugging Page Faults

Below are some examples for debugging page faults using the Bochs emulator. We recommend using Bochs, rather than QEMU, to debug kernel crashes because QEMU exits when the kernel crashes, precluding after-the-fact debugging. In order to use the Bochs emulator for a specific test-run/debugging-session, use the `FORCE_SIMULATOR` environment variable. For example, to debug the `do-nothing` test using the Bochs emulator, you would run:

```
FORCE_SIMULATOR=--bochs PINTOS_DEBUG=1 pintos-test do-nothing
```

In the event that you encounter a bug that only shows up in QEMU, you can try setting a breakpoint in the page fault handler to allow for debugging before QEMU exits.

If you encounter a page fault during a test, you should use the method in Debugging Pintos Tests to debug Pintos with GDB.

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

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

```
(gdb) debugpintos
(gdb) continue

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

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

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

4.1.8 Debugging Kernel Panics

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

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

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

```
$ cd ~/code/personal/proj0/src/threads/build/
$ backtrace kernel.o 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319 ...
```

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

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

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

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

You can transform it into this:

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

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

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

4.1.9 Adding Source Files

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

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

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

4.1.10 Why Pintos?

Why the name "Pintos"? First, like nachos (the operating system previously used in CS 162), 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.

4.2 User Programs

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

4.2.1 Overview of Source Files for Project 1

threads/thread.h Contains the `struct thread` definition, which is the Pintos thread control block. The fields in `#ifdef USERPROG ... #endif` are collectively the process control block. We expect that you will add fields to the process control block in this project.

userprog/process.c Loads ELF binaries, starts processes, and switches page tables on context switch.

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

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

lib/user/syscall.c Provides library functions for user programs to invoke system calls from a C program. Each function uses inline assembly code to prepare the syscall arguments and invoke the system call. We do expect you to understand the calling conventions used for syscalls (also in Reference).

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

userprog/exception.c Handle exceptions. Currently all exceptions simply print a message and terminate the process. Some, but not all, solutions to Project 1 involve modifying page_fault() in this file.

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

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

4.2.2 How User Programs Work

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

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

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

4.2.3 Virtual Memory Layout

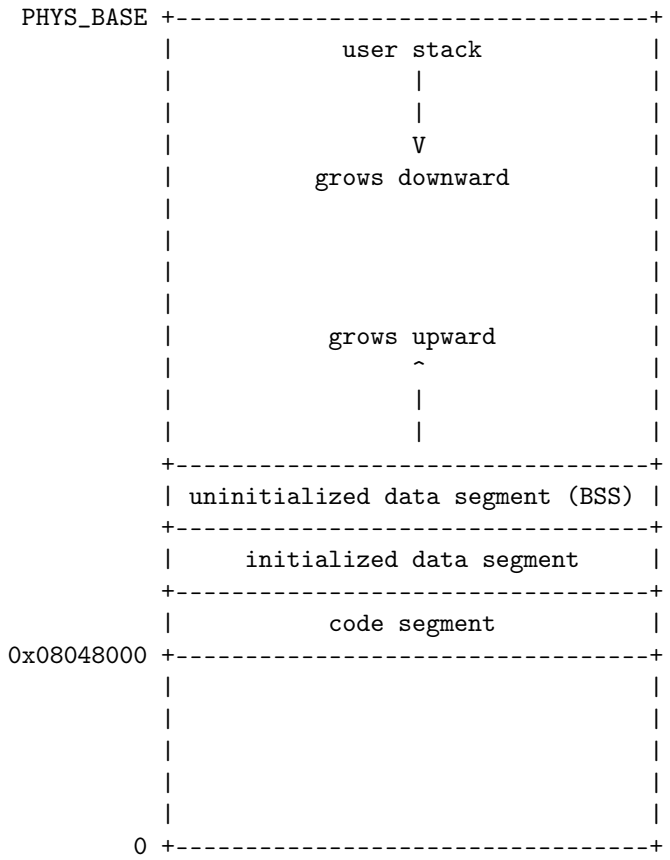
Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address `0` up to `PHYS_BASE`, which is defined in `threads/vaddr.h` and defaults to `0xc0000000` (3 GB). Kernel virtual memory occupies the rest of the virtual address space, from `PHYS_BASE` up to 4 GB.

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

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

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

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



4.2.4 Accessing User Memory

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

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

- verify the validity of a user-provided pointer, then dereference it. If you choose this route, you'll want to look at the functions in `userprog/pagedir.c` and in `threads/vaddr.h`. This is the simplest way to handle user memory access.
- check only that a user pointer points below `PHYS_BASE`, then dereference it. An invalid user pointer will cause a "page fault" that you can handle by modifying the code for `page_fault()` in `userprog/exception.c`. This technique is normally faster because it takes advantage of the processor's MMU, so it tends to be used in real kernels (including Linux).

In either case, you need to make sure not to "leak" resources. For example, suppose that your system call has acquired a lock or allocated memory with `malloc()`. If you encounter an invalid user pointer afterward, you must still be sure to release the lock or free the page of memory. If you choose to verify user pointers before dereferencing them, this should be straightforward. It's more difficult to handle if an invalid pointer causes a page fault, because there's no way to return an error code from a memory

access. Therefore, for those who want to try the latter technique, we'll provide a little bit of helpful code:

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

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

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

If you do choose to use the second option (rely on the processor's MMU to detect bad user pointers), do not feel pressured to use the `get_user` and `put_user` functions from above. There are other ways to modify the page fault handler to identify and terminate processes that pass bad pointers as arguments to system calls, some of which are simpler and faster than using `get_user` and `put_user` to handle each byte.

4.2.5 Switching Processes

It is sometimes useful while debugging to switch between processes. You'll need to use the Bochs emulator for this to work (see Debugging Page Faults for instructions on how to do this). To observe the process switch, do the following in a fresh (i.e. with no existing breakpoints set) GDB session. Lines below starting with "#" are comments.

```
debugpintos
break process.c:process_exit
continue
# Shows you the current thread id
call thread_current()->tid
# Sets a breakpoint immediately after the call to sema_down in process_wait
break 95
continue
# You should see a different thread id than before
call thread_current()->tid
# Shows other threads currently present in Pintos
dumplist &all_list thread allelem
```

4.2.6 80x86 Calling Convention

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

The calling convention works like this:

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

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

2. The caller pushes the address of its next instruction (the *return address*) on the stack and jumps to the first instruction of the callee. A single 80x86 instruction, `call`, does both.

3. The callee executes. When it takes control, the stack pointer points to the return address, the first argument is just above it, the second argument is just above the first argument, and so on.

4. If the callee has a return value, it stores it into register `eax`.

5. The callee returns by popping the return address from the stack and jumping to the location it specifies, using the 80x86 `ret` instruction.

6. The caller pops the arguments off the stack.

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

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

4.2.7 Program Startup Details

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

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

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

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

Then, push the address of each string plus a null pointer sentinel, on the stack, in right-to-left order. These are the elements of `argv`. The null pointer sentinel ensures that `argv[argc]` is a null pointer, as required by the C standard. The order ensures that `argv[0]` is at the lowest virtual address. The x86 ABI requires that `%esp` be aligned to a 16-byte boundary at the time the `call` instruction is executed (e.g., at the point where all arguments are pushed to the stack), so make sure to leave enough empty space on the stack so that this is achieved.

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

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

Address	Name	Data	Type
0xbffffffc	argv[3][...]	bar\0	char[4]
0xbffffff8	argv[2][...]	foo\0	char[4]
0xbffffff5	argv[1][...]	-l\0	char[3]
0xbfffffed	argv[0][...]	/bin/ls\0	char[8]
0xbfffffec	stack-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

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

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

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

```
bffffc0                                      00 00 00 00 |            ....|
bffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |................|
```

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

4.2.8 Adding New Tests to Pintos

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

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

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

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

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

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

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

4.3 Threads

4.3.1 Understanding Threads

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

Some of this code might seem slightly mysterious. You can read through parts of the source code to see what's going on. If you like, you can add calls to `printf()` almost anywhere, then recompile and run to see what happens and in what order. You can also run the kernel in a debugger and set breakpoints at interesting spots, step through code and examine data, and so on.

When a thread is created, the creator specifies a function for the thread to run, as one of the arguments to `thread_create()`. The first time the thread is scheduled and runs, it starts executing from the beginning of that function. When the function returns, the thread terminates. Each thread, therefore, acts like a mini-program running inside Pintos, with the function passed to `thread_create()` acting like `main()`.

At any given time, exactly one thread runs and the rest become inactive. The scheduler decides which thread to run next. (If no thread is ready to run, then the special "idle" thread runs.)

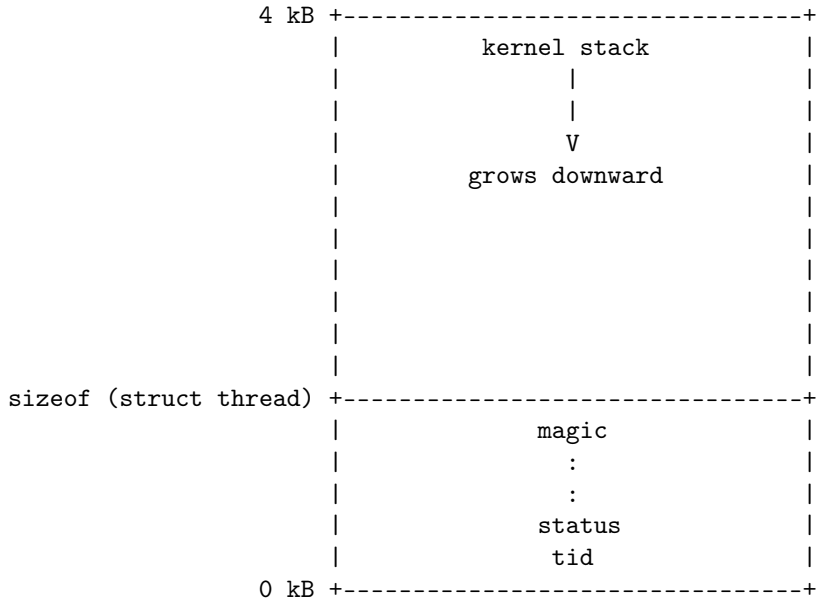
The mechanics of a context switch are in `threads/switch.S`, which is x86 assembly code. It saves the state of the currently running thread and restores the state of the next thread onto the CPU.

Using GDB, try tracing through a context switch to see what happens. You can set a breakpoint on `schedule()` to start out, and then single-step from there (use "`step`" instead of "`next`"). Be sure to keep track of each thread's address and state, and what procedures are on the call stack for each thread (try "`backtrace`"). You will notice that when one thread calls `switch_threads()`, another thread starts running, and the first thing the new thread does is to return from `switch_threads()`. You will understand the thread system once you understand why and how the `switch_threads()` that gets called is different from the `switch_threads()` that returns.

4.3.2 The Thread Struct

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

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



This layout has two consequences. First, struct thread must not be allowed to grow too big. If it does, then there will not be enough room for the kernel stack. The base struct thread is only a few bytes in size. It probably should stay well under 1 kB.

Second, kernel stacks must not be allowed to grow too large. If a stack overflows, it will corrupt the thread state. Thus, kernel functions should not allocate large structures or arrays as non-static local variables. Use dynamic allocation with `malloc()` or `palloc_get_page()` instead. See the Memory Allocation section for more details.

- **Member of struct thread:** `tid_t tid`
  The thread's thread identifier or *tid*. Every thread must have a tid that is unique over the entire lifetime of the kernel. By default, `tid_t` is a `typedef` for `int` and each new thread receives the numerically next higher tid, starting from 1 for the initial process.

- **Member of struct thread:** `enum thread_status status`
  The thread's state, one of the following:

  - **Thread State:** `THREAD_RUNNING`
    The thread is running. Exactly one thread is running at a given time. `thread_current()` returns the running thread.
  - **Thread State:** `THREAD_READY`
    The thread is ready to run, but it's not running right now. The thread could be selected to run the next time the scheduler is invoked. Ready threads are kept in a doubly linked list called `ready_list`.
  - **Thread State:** `THREAD_BLOCKED`
    The thread is waiting for something, e.g. a lock to become available, an interrupt to be invoked. The thread won't be scheduled again until it transitions to the `THREAD_READY` state with a call to `thread_unblock()`. This is most conveniently done indirectly, using one of the Pintos synchronization primitives that block and unblock threads automatically.
  - **Thread State:** `THREAD_DYING`
    The thread has exited and will be destroyed by the scheduler after switching to the next thread.

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

- **Member of struct thread:** `uint8_t *stack`
  Every thread has its own stack to keep track of its state. When the thread is running, the CPU's stack pointer register tracks the top of the stack and this member is unused. But when the CPU switches to another thread, this member saves the thread's stack pointer. No other members are needed to save the thread's registers, because the other registers that must be saved are saved on the stack.

  When an interrupt occurs, whether in the kernel or a user program, an "`struct intr_frame`" is pushed onto the stack. When the interrupt occurs in a user program, the "`struct intr_frame`" is always at the very top of the page.

- **Member of struct thread:** `int priority`
  A thread priority, ranging from `PRI_MIN` (0) to `PRI_MAX` (63). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. Pintos currently ignores these priorities, but you will implement priority scheduling in this project.

- **Member of struct thread:** `struct list_elem allelem`
  This "list element" is used to link the thread into the list of all threads. Each thread is inserted into this list when it is created and removed when it exits. The `thread_foreach()` function should be used to iterate over all threads.

- **Member of struct thread:** `struct list_elem elem`
  A "list element" used to put the thread into doubly linked lists, either `ready_list` (the list of threads ready to run) or a list of threads waiting on a semaphore in `sema_down()`. It can do double duty because a thread waiting on a semaphore is not ready, and vice versa.

- **Member of struct thread:** `uint32_t *pagedir`
  (Used in Projects 1 and 3.) The page table for the process, if this is a user process.

- **Member of struct thread:** `unsigned magic`
  Always set to `THREAD_MAGIC`, which is just an arbitrary number defined in `threads/thread.c`, and used to detect stack overflow. `thread_current()` checks that the `magic` member of the running thread's `struct thread` is set to `THREAD_MAGIC`. Stack overflow tends to change this value, triggering the assertion. For greatest benefit, as you add members to `struct thread`, leave `magic` at the end.

4.3.3 Thread Functions

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

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

  Before `thread_init()` runs, `thread_current()` will fail because the running thread's `magic` value is incorrect. Lots of functions call `thread_current()` directly or indirectly, including `lock_acquire()` for locking a lock, so `thread_init()` is called early in Pintos initialization.

- **Function:** `struct thread *thread_current (void)`
  Returns the running thread.

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

4.4 Synchronization

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

4.4.1 Disabling Interrupts

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

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

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

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

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

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

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

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

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

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

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

4.4.2 Semaphores

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

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

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

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

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

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

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

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

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

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

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

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

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

4.4.3 Locks

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

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

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

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

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

- **Function:** `void lock_init (struct lock *lock)`
  Initializes `lock` as a new lock. The lock is not initially owned by any thread.

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

- **Function:** `bool lock_try_acquire (struct lock *lock)`
  Tries to acquire `lock` for use by the current thread, without waiting. Returns true if successful, false if the lock is already owned. Calling this function in a tight loop is a bad idea because it wastes CPU time, so use `lock_acquire` instead.

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

- **Function:** `bool lock_held_by_current_thread (const struct lock *lock)`
  Returns true if the running thread owns `lock`, false otherwise. There is no function to test whether an arbitrary thread owns a lock, because the answer could change before the caller could act on it.

4.4.4 Monitors

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

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

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

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

- **Type:** `struct condition`
  Represents a condition variable.

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

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

  Sending a signal and waking up from a wait are not an atomic operation. Thus, typically `cond_wait`'s caller must recheck the condition after the wait completes and, if necessary, wait again.

- **Function:** `void cond_signal (struct condition *cond, struct lock *lock)`
  If any threads are waiting on `cond` (protected by monitor lock `lock`), then this function wakes up one of them. If no threads are waiting, returns without performing any action. `lock` must be held before calling this function.

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

4.4.5 Optimization Barriers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5 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.

4.5.1 Page Allocator

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

The page allocator divides the memory it allocates into two pools, called the kernel and user pools. By default, each pool gets half of system memory above 1 MiB, but the division can be changed with the `-ul` kernel command line option. An allocation request draws from one pool or the other. If one pool becomes empty, the other may still have free pages. The user pool should be used for allocating memory for user processes and the kernel pool for all other allocations. This distinction is not very relevant in this project, since all threads you will be dealing with are kernel threads (unlike in Project 1). For Project 2, all allocations should be made from the kernel pool.

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

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

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

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

Page allocator types and functions are described below.

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

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

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

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

4.5.2 Block Allocator

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

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

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

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

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

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

The block allocator may not be called from interrupt context.

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

- **Function:** `void * malloc (size_t size)`
  Obtains and returns a new block, from the kernel pool, at least `size` bytes long. Returns a null pointer if `size` is zero or if memory is not available.

- **Function:** `void * calloc (size_t a, size_t b)`
  Obtains a returns a new block, from the kernel pool, at least `a * b` bytes long. The block's contents will be cleared to zeros. Returns a null pointer if `a` or `b` is zero or if insufficient memory is available.

- **Function:** `void * realloc (void *block, size_t new_size)`
  Attempts to resize `block` to `new_size` bytes, possibly moving it in the process. If successful, returns the new block, in which case the old block must no longer be accessed. On failure, returns a null pointer, and the old block remains valid.

  A call with `block` null is equivalent to `malloc`. A call with `new_size` zero is equivalent to `free`.

- **Function:** `void free (void *block)`
  Frees `block`, which must have been previously returned by `malloc`, `calloc`, or `realloc` (and not yet freed).

4.6 Linked Lists

Pintos contains a linked list data structure in `lib/kernel/list.h` that is used for many different purposes. This linked list implementation is different from most other linked list implementations you may

have encountered, because **it does not use any dynamic memory allocation**.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.7 Debugging Tips

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

4.7.1 printf

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

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

4.7.2 ASSERT

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

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

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

4.7.3 Function and Parameter Attributes

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

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

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

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

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

4.7.4 Backtraces

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

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

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

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

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

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

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

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

The backtrace output would then look something like this:

```
0xc0106eff: debug_panic (lib/debug.c:86)
0xc01102fb: file_seek (filesys/file.c:405)
0xc010dc22: seek (userprog/syscall.c:744)
0xc010cf67: syscall_handler (userprog/syscall.c:444)
```

```
0xc0102319: intr_handler (threads/interrupt.c:334)
0xc010325a: intr_entry (threads/intr-stubs.s:38)
0x0804812c: (unknown)
0x08048a96: (unknown)
0x08048ac8: (unknown)
```

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

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

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

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

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

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

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

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

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

The results look like this:

```
0xc0106eff: (unknown)
0xc01102fb: (unknown)
0xc010dc22: (unknown)
0xc010cf67: (unknown)
0xc0102319: (unknown)
0xc010325a: (unknown)
0x0804812c: test_main (...xtended/grow-too-big.c:20)
0x08048a96: main (tests/main.c:10)
0x08048ac8: _start (lib/user/entry.c:9)
```

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

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

The result is a combined backtrace:
in `kernel.o`:

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

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

```
0x0804812c: test_main (...xtended/grow-too-big.c:20)|
0x08048a96: main (tests/main.c:10)|
0x08048ac8: _start (lib/user/entry.c:9)|
```

Here's an extra tip: `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.
```

4.7.5 GDB

Using GDB

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

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

break function

break file:line

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

Use `break main` to make GDB stop when Pintos starts running.

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

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

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

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

diassemble function disassembles `function`.

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

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

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

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

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

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

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

btthreadall short-hand for `btthreadlist all_list allelem`.

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

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

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

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

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

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

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

In Project 1, a page fault in a user process leads to the termination of the process. You should expect those page faults to occur in the robustness tests where we test that your kernel properly terminates processes that try to access invalid addresses. To debug those, set a breakpoint in `page_fault` in `exception.c`, which you will need to modify accordingly.

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

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

Followed by the output of the `btpagefault` command.

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

```
loadusersymbol program
```

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

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

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

4.7.6 Triple Faults

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

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

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

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

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

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

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

4.7.7 General Tips

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

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