CS 162 Project 1: User Programs

Design Document:	Friday, September 17, 2021, 11:59 PM PDT
[Ungraded] Code Checkpoint #1:	Friday, September 24, 2021, 11:59 PM PDT
[Ungraded] Code Checkpoint #2:	Friday, October 1, 2021, 11:59 PM PDT
Code:	Wednesday, October 06, 2021, 11:59 PM PDT
Final Report:	Wednesday, October 06, 2021, 11:59 PM PDT

Contents

1 Introduction

Welcome to the first full project of CS 162! Our projects in this class will use Pintos, an educational operating system. They are 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: by the end of the course, you will have implemented the majority of a fully functional, if somewhat spartan, bona fide operating system!

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.

1.1 Setup

1. Run the following commands to create the `group` folder and attach it to the GitHub repository containing the skeleton code:

   ```
   rm -rf ~/code/group
   git clone -o staff https://github.com/Berkeley-CS162/group0.git ~/code/group
   cd ~/code/group/
   ```

2. Then visit your group repo on GitHub and find the SSH clone URL. It should have the form "`git@github.com:Berkeley-CS`

3. Now run the following command to add the `group` remote:

   ```
   git remote add group YOUR_GITHUB_CLONE_URL
   ```

4. You can get information about the remote you just added by running the following commands:

   ```
   git remote -v
   git remote show group
   ```

5. Install the pre-commit hook by running the following command:

   ```
   ln -s -f ../../.pre-commit.sh .git/hooks/pre-commit
   ```

If you have completed all of the above, your group repository will then exist at `~/code/group`, and will have two remotes:

- `staff` – points to the group0 (i.e. skeleton-code) repository
- `group` – points to your group's GitHub repository

2 Your Task

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

2.1 Task 1: Data Transfer

2.1.1 Argument Passing

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

Many of our Pintos test programs start by printing out their own name (e.g. `argv[0]`). Since argument passing has not yet been implemented, all of these programs will crash when they access `argv[0]`. Until you implement argument passing, these user programs will not work.

2.1.2 Floating Point Instructions

The Pintos kernel does not currently support floating point operations. You must update the OS so that both user programs and the kernel can use floating point instructions.

This may seem like a daunting task, but rest assurred, the OS doesn't have to do much when it comes to implementing floating point instructions: the compiler does the hard work of generating floating point instructions, and the hardware does the hard work of executing floating point instructions. However, because there is only one floating-point unit[1] (FPU) on the CPU, all threads on must share it. This is where the OS comes into play. The OS must:

1. Save the state of the FPU on context switches
2. Save the state of the FPU on interrupts & system calls

Pintos already saves general purpose registers (GPRs) on the stack during context switches and interrupts. See `src/threads/switch.S` and `src/threads/intr-stubs.S` assembly routines for the implementation. Saving floating point registers can be done in a similar way. You must also:

- Update one line of `src/threads/start.S` to allow FPU instructions to be executed on the hardware (see 4.2.2 Enabling the FPU)
- Ensure floating-point registers are initialized correctly when a new thread or process is created (see 4.2.3 FPU Initialization), unlike with GPRs
- Implement a system call to compute the value e (see 4.2.4 FPU Syscall)

2.2 Task 2: Process Control Syscalls

Pintos currently only supports one syscall – `exit` – which terminates the calling thread. You will add support for the following new syscalls: `practice`, `halt`, `exec`, and `wait`. The `practice` syscall just adds 1 to its first argument, and returns the result (to give you practice writing a syscall handler). The `halt` syscall will shut down the system. The `exec` syscall will start a new program with `process_execute()`. (There is no `fork` syscall in Pintos. The Pintos `exec` syscall is similar to calling Linux's `fork` syscall and then Linux's `execve` syscall in the child process immediately afterward.) The `wait` syscall will wait for a specific

[1] https://en.wikipedia.org/wiki/Floating-point_unit

child process to exit. Each of these syscalls has a corresponding function inside the user-level library in `src/lib/user/syscall.c`, which prepares the syscall arguments and handles the transfer to kernel mode. The kernel's syscall handler is located in `src/userprog/syscall.c`. See Process System Calls for more details.

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

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

2.3 Task 3: File Operation Syscalls

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

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

While a user process is running, you must ensure that nobody can modify its executable on disk. The "rox" tests check that this has been implemented correctly. The functions `file_deny_write()` and `file_allow_write()` can assist with this feature. Denying writes to executables backing live processes is important because an operating system may load code pages from the file lazily, or may page out some code pages and reload them from the file later. In Pintos, this is technically not a concern because the file is loaded into memory in its entirety before execution begins, and Pintos does not implement demand paging of any sort. However, you are still required to implement this, as it is good practice.

Note: Your final code for Project 1 will be used as a starting point for Project 3. The tests for Project 3 depend on some of the same syscalls that you are implementing for this project, and you may have to modify your implementations of some of these syscalls to support additional features required for Project 3. You should keep this in mind while designing your implementation for this project.

3 Deliverables

Your project grade will be made up of 4 components:

- 15% Design Document and Design Review
- 70% Code
- 10% Final Report, Code Quality
- 5% Student Testing

3.1 Design Document and Design Review

Before you start writing any code for your project, you should create an implementation plan for each feature and convince yourself that your design is correct. For this project, you must **submit a design document** and **attend a design review** with your project TA. You will submit your design document as a PDF to the Project 1 Design Document assignment on Gradescope.

Please ensure your design document is well-formatted, as this makes it much easier for your TA to read it and therefore give you helpful feedback. In particular, **please do NOT use Google Docs**, as it lacks the capability to natively and easily format code-blocks. Dropbox Paper[2] is our recommended free platform for creating your design doc, as it has real-time collaborative editing very similar to Google Docs, but has excellent support for code formatting.

There are two parts to the design document. The first part is a Design Overview where you will include an overview of your proposed design for completing Project 1. The second part is to answer some Additional Questions. We explain each part of the design document in detail in the sections below.

3.1.1 Design Overview

For each of the 3 tasks of this project, you must explain the following 4 aspects of your proposed design. We suggest you create a section for each of the 3 project parts. Then, in each section, create subsections for each of these 4 aspects.

1. **Data structures and functions** – Write down any `struct` definitions, global (or static) variables, `typedef`s, or enumerations that you will be adding or modifying (if it already exists). These definitions should be written with the **C programming language**, not with pseudocode. Include a **brief explanation** of the purpose of each modification. Your explanations should be as concise as possible. Leave the full explanation to the following sections.
2. **Algorithms** – This is where you tell us how your code will work. Your description should be at a level below the high level description of requirements given in the assignment. We have read the project spec too, so it is unnecessary to repeat or rephrase what is stated here. On the other hand, your description should be at a level above the code itself. Don't give a line-by-line run-down of what code you plan to write. Instead, you should try to convince us that your design satisfies all the requirements, **including any uncommon edge cases**. We expect you to read through the Pintos source code when preparing your design document, and your design document should refer to the Pintos source when necessary to clarify your implementation.
3. **Synchronization** – This section should list all resources that are shared across threads. For each case, enumerate how the resources are accessed (e.g., from an interrupt context, etc), and describe the strategy you plan to use to ensure that these resources are shared and modified safely. For each resource, demonstrate that your design ensures correct behavior and avoids deadlock. In general, the

[2] https://paper.dropbox.com

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

4. **Rationale** – Tell us why your design is better than the alternatives that you considered, or point out any shortcomings it may have. You should think about whether your design is easy to conceptualize, how much coding it will require, the time/space complexity of your algorithms, and how easy/difficult it would be to extend your design to accommodate additional features.

3.1.2 Additional Questions

You must also answer these additional questions in your design document:

1. Take a look at the Project 1 test suite in `src/tests/userprog`. Some of the test cases will intentionally provide invalid pointers as syscall arguments, in order to test whether your implementation safely handles the reading and writing of user process memory. Please identify a test case that uses an **invalid** stack pointer (`%esp`) when making a syscall. Provide the name of the test and explain how the test works. Your explanation should be very specific: use line numbers and the actual names of variables when explaining the test case.

2. Please identify a test case that uses a **valid** stack pointer when making a syscall, but the stack pointer is too close to a page boundary, so some of the syscall arguments are located in invalid memory (your implementation should kill the user process in this case). Provide the name of the test and explain how the test works. Your explanation should be very specific: use line numbers and the actual names of variables when explaining the test case.

3. Identify **one** part of the project requirements which is **not fully tested by the existing test suite**. Explain what kind of test needs to be added to the test suite, in order to provide coverage for that part of the project. There are multiple good answers for this question.

3.1.3 Design Review

You will schedule a 20-30 minute design review with your project TA. During the design review, your TA will ask you questions about your design for the project. You should be prepared to defend your design and answer any clarifying questions your TA may have about your design document. The design review is also a good opportunity to get to know your TA for those participation points.

3.1.4 Grading

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

3.2 Code

The code section of your grade will be determined by your autograder score. Pintos comes with a test suite that you can run locally on your VM. We run the same tests on the autograder. The results of these tests will determine your code score. **Be sure to push your code to GitHub for the checkpoints, and the final code. This is how we will track your progress for the checkpoints.**

You can check your current grade for the code portion at any time by logging in to the course autograder. Autograder results will also be emailed to you.

We will check your progress on Project 1 at two intermediate checkpoints. **These checkpoints will not be counted towards the final grade for your project.** However, it is in your best interest to complete them to ensure that your group is on pace to finish the assignment. Our goal is not to grade your in-progress implementations, but to ensure that you're making satisfactory progress and encourage you to ask for help early and often.

3.3 Checkpoint #1 [Ungraded]

You must have implemented the following:

- The `write` syscall for the `STDOUT` file descriptor only.
- The `practice` syscall.
- Argument Passing in its entirety.

We recommend that you begin the project by implementing the `write` syscall for the `STDOUT` file descriptor. Once you've done this, the `stack-align-1` test should pass, assuming you build on the code you have at the end of Project 0, and you properly align the stack when no command line arguments are passed to the user program.

After you've completed the above task and `printf()` works from userspace, implement the `practice` syscall and argument passing.

As a final step, make sure that the `exit(-1)` message is printed even if a process exits due to a fault. Currently the exit code is printed [3] when the `exit` syscall is made from userspace, but not if it the process exits due to an invalid memory access.

3.4 Checkpoint #2 [Ungraded]

In addition to the tasks in Checkpoint #1 [Ungraded], you must have completed Floating Point Instructions and Task 2: Process Control Syscalls in its entirety.

3.5 Final Code

You must have completed the entire project: Task 1: Data Transfer, Task 2: Process Control Syscalls, and Task 3: File Operation Syscalls.

3.5.1 Student Testing Code

Pintos already contains a test suite for Project 1, but not all of the parts of this project have complete test coverage. **Your task is to submit 2 new test cases, which exercise functionality that is not covered by existing tests.** We will not tell you what features to write tests for. You will be responsible for identifying which features of this project would benefit most from additional tests. Make sure your own project implementation passes the tests that you write. You can pick any appropriate name for your test, but beware that test names should be no longer than 14 characters. Once you finish writing your test cases, make sure that they get executed when you run "`make check`" in the `src/userprog/` directory.

For this project, you can either write tests in user-space or in the kernel. User-space tests interact with the kernel through system calls, whereas kernel tests have direct access to the kernel. We encourage you

[3] see line 32 of `/src/userprog/syscall.c`

to write user-space tests, though it may be more convenient to write kernel-based tests if you want to test the FPU. All user-space tests are located in `src/tests/userprog/` and all kernel tests are located in `src/tests/userprog/kernel/`.

3.6 Student Testing Report

While the tests themselves must be submitted with the rest of your code, you will also need to prepare a Student Testing Report, which will help us grade your test cases. Include your student testing report as a section in the same PDF as the rest of your final report (described in the next section).

Make sure your Student Testing Report contains the following:

- For each of the 2 test cases you write:
  - Provide a description of the feature your test case is supposed to test.
  - Provide an overview of how the mechanics of your test case work, as well as a qualitative description of the expected output.
  - Provide the output of your own Pintos kernel when you run the test case. Please copy the full raw output file from `userprog/build/tests/userprog/your-test-1.output` as well as the raw results from `userprog/build/tests/userprog/your-test-1.result`, or `userprog/build/tests/userprog/kern`
    and
    `userprog/build/tests/userprog/kernel/your-test-1.result` if you write a kernel test
  - Identify two non-trivial potential kernel bugs, and explain how they would have affected the output of this test case. You should express these in this form: "If your kernel did X instead of Y, then the test case would output Z instead.". You should identify two different bugs per test case, but you can use the same bug for both of your two test cases. These bugs should be related to your test case (e.g. "If your kernel had a syntax error, then this test case would not run." does not count).
- Tell us about your experience writing tests for Pintos. What can be improved about the Pintos testing system (there's room for improvement)? What did you learn from writing test cases?

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

3.7 Final Report and Code Quality

After you complete the code for your project, your group will submit a final report in the form of a PDF to the Project 1 Final Report assignment on Gradescope. Please include the following in your final report:

- The changes you made since your initial design document and why you made them (feel free to re-iterate what you discussed with your TA in the design review)
- A reflection on the project – what exactly did each member do? What went well, and what could be improved?
- Your Student Testing Report (see the previous section for more details).

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

- Does your code exhibit any major memory safety problems (especially regarding C strings), memory leaks, poor error handling, or race conditions?

- Is your code simple and easy to understand? Does it follow a consistent naming convention?
- If you have very complex sections of code in your solution, did you add enough comments to explain them?
- Did you leave commented-out code in your final submission?
- Did you copy-paste code instead of creating reusable functions?
- Did you re-implement linked list algorithms instead of using the provided list manipulation functions?
- Is your Git commit history full of binary files? (don't commit object files or log files for this project)

Note that you don't have to worry about manually enforcing code formatting (e.g. indentation, spacing, wrapping long lines, consistent placement of braces, etc.) as the make rule `make format` (which is run automatically each time you commit provided you followed the setup instructions correctly) takes care of this.

4 Reference

The majority of the Pintos reference is now located in the specification for Project 0, "Introduction to Pintos"[4] in an effort to keep this project specification from being intimidatingly long. Please be sure to go through it.

We have copied here the User Programs section because it is useful for both Project 0 and Project 1. Additionally, you will need to carefully read through the Floating Point and System Calls sections (which are *not* included in the Project 0 Reference) as it details the required behavior of FPU saving and system calls you will implement for this project. In addition, we highly recommend you read the advice section included at the end of this specification.

4.1 User Programs

User programs are written under the illusion that they have the entire machine to themselves, 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).

4.1.1 Overview of Source Files for Project 1

threads/start.S Contains assembly code that the Pintos bootloader jumps to. This is the first kernel code that starts executing. You will have to modify this file to implement 4.2.2 Enabling the FPU.

threads/init.c The kernel code in `threads/start.S` does some basic setup then jumps to `int main(void)` in `threads/init.c`, which does more setup in C, then executes the actions requested of Pintos (for example, running a test or running a user program).

threads/thread.c Contains code for thread creation and thread scheduling. You will have to modify this file to implement 4.2.3 FPU Initialization.

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.

threads/intr-stubs.S Contains the code for storing registers and executing an interrupt handler. The corresponding interrupt stack frame structure is defined in `threads/interrupt.h`. You should modify this code to save FPU state.

threads/switch.S Contains the code for storing registers on a context switch between two kernel threads. The corresponding switch-threads stack frame structure is defined in `switch.h`. You should modify this code to save FPU state.

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.

[4]https://cs162.org/static/projects/project0.pdf

lib/float.h This file has important floating point functions for implementing this project, and can be used in both userspace for testing and in the kernel

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.

4.1.2 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.

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, Pintos will be unable to do useful work. The provided test framework and `pintos-test` take care of running user programs in tests for you.

4.1.3 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:

```
 PHYS_BASE +----------------------------------+
           |            user stack            |
           |                 |                |
           |                 |                |
           |                 V                |
           |          grows downward          |
           |                                  |
           |                                  |
           |                                  |
           |                                  |
           |           grows upward           |
           |                 ^                |
           |                 |                |
           |                 |                |
           +----------------------------------+
           | uninitialized data segment (BSS) |
           +----------------------------------+
           |     initialized data segment     |
           +----------------------------------+
           |           code segment           |
0x08048000 +----------------------------------+
           |                                  |
           |                                  |
           |                                  |
           |                                  |
           |                                  |
         0 +----------------------------------+
```

4.1.4 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.

4.1.5 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*) onto 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 |
                                +----------------+
```

4.1.6 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:

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

4.1.7 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. Most 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`[5] 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).

[5] https://man7.org/linux/man-pages/man2/brk.2.html

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 the Pintos test driver that your test passed.

Some userprog tests can directly access the kernel. These are located in tests/userprog/kernel/. To add a kernel test, you can roughly follow the above steps, except in the tests/userprog/kernel/ directory. There are some differences in making a kernel test, however. To isolate the differences, follow the examples in the tests/userprog/kernel/ directory.

4.2 Floating Point

4.2.1 Introduction to the FPU

Technically, the operating system does not *need* to do anything special for user programs to execute floating point code, as long as that code computes the relevant FPU operations 'manually' — that is to say, by performing the bit-level manipulations necessary for perform addition, subtraction, multiplication, etc. on floating point values within software, rather than calling special hardware instructions to do so. Of course, this method, while not infeasible (many embedded devices use this approach), is very slow; as such, most modern operating systems provide support for the hardware to expose floating point instructions to the user directly. The skeleton code you were given was written without consideration for the floating-point components of the system, and as such user code with floating-point operations will not function correctly when run within Pintos as-is. You will have to determine what changes will need to be made to Pintos to enable this functionality in order to complete this portion of the project.

Unlike the other components of the x86 instruction set architecture that you've encountered thus far, support for the floating point unit was originally included as a separate co-processor with its own ISA (known as x87). As such, it behaves in a very different way than the rest of the system. The FPI doesn't have specific floating point registers that you can address independently: instead, it has a *stack* of registers, allowing programmers to interact with the FPU by pushing to and popping from the stack. For example, to compute the square root of π, you would first push π onto the stack, and then run the fsqrt instruction. This instruction will pop the top of the stack, compute the square root of that value, and push it onto the top of the stack. Now, $\sqrt{\pi}$ is at the top of the stack. Other operations work similarly: fadd removes the top two operands from the stack and adds them together, placing the result back on the top of the stack.

For information on what each of the instructions does, as well as other details about the FPU architecture, Intel's x86 Guide[6] has more information on exactly what each instruction does; while the document might seem intimidating at first glance, most of its length comes from explaining the individual instructions in detail: the component describing the overall layout, structure, and requirements of the architecture is much shorter,

[6] https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html

and may prove helpful for your implementation (plus, getting comfortable with hardware specs is a good habit to get in to when working on OS-level code).

4.2.2 Enabling the FPU

When you first try to issue floating point instructions on the processor, you may run into a "Device Not Found" Exception. This is because the Pintos starter code intializes the CR0[7] register to indicate that there is no FPU by setting the `CR0_EM` bit. You must update this as follows:

```
- orl $CR0_PE | CR0_PG | CR0_WP | CR0_EM, %eax
+ orl $CR0_PE | CR0_PG | CR0_WP, %eax
```

in `src/threads/start.S` to remove the flag and indicate to the processor that the FPU is present.

4.2.3 FPU Initialization

Another difference Newly-created threads and processes generally cannot assume anything about the contents of GPRs, meaning GPRs do not have to be initialized to any particular values when a new thread or process is created. However, newly-created threads and processes in general should be able to assume that the FPU is clean (it has been initialized/reset properly). So, you must ensure to also:

1. Initialize the FPU at OS startup
2. Re-initialize the FPU when a new thread is created
3. Re-initialize the FPU when a new process is created

When initializing the FPU for a new thread/process, you must make sure the current thread (the thread that is creating the new thread/process) does not lose it's own floating point data.

4.2.4 FPU Syscall

As a chance for you to see floating point operations in practice and for us to test your code, you will also need to implement the `double compute_e(int n)` syscall. This system call computes an approximation of e as $e \approx \sum_{i=0}^{n} \frac{1}{n!}$ using the Taylor series for the constant e. For information on how to implement a system call, see 2.2 Task 2: Process Control Syscalls and 2.3 Task 3: File Operation Syscalls. Most of the work for implementing this system call has been done for you in `sys_sum_to_e` in `lib/float.c`. Once you validate the argument `int n` from the system call, you can simply store the result of `sys_sum_to_e(n)` in the `%eax` register for return. This system call is not intended to be difficult to implement correctly, but rather as a check to confirm that your kernel correctly supports FPU operations in general.

4.2.5 Writing Assembly

Just like with operations concerning the general purpose registers, you must use assembly code to interact with the FPU directly. You will not be able to complete this portion of the project without understanding some of the existing assembly code in the skeleton, or without writing assembly code of your own.

When you are writing assembly, we highly recommend that you do so directly in the relevant `.S` files: it may seem tempting to instead write inline assembly within the C code via `asm`, but it requires additional work to annotate which registers are in use within the block so that the compiler knows how to handle them correctly, and is very easy to mess up accidentally. You can also see examples of both `asm` as well as direct assembly in `.S` files in the starter code — most editors can be made to support syntax highlighting for GNU assembly, but you may need to dig around a little bit to tell it to do so for .S files in particular.

[7] https://en.wikipedia.org/wiki/Control_register#CR0

For Pintos, **use AT&T syntax** when writing floating point assembly. There are unfortuantely not many great guides on the instruction formats (e.g. orders and types of operands) for floating point instructions easily available online. However, you may find that these two[8] guides[9] may provide some information. You can also look at the starter code for examples of floating point instructions and their operand types and orders, and even generate your own with `gcc -S`, which outputs assembly code from C source code.

4.2.6 FPU Tips

Tips: First, we recommend that you complete the code for this section on a separate branch from the rest of the project, and merge in when it's complete. To complete this section, it's important to thoroughly read and understand the switching code in Pintos. Track where floating point state needs to be saved or re-initialized along the way: note that, since a lot of this code happens 'behind the scenes' from the perspective of a normal C program, it's handled directly in assembly. For example, you might consider when FPU state needs to be saved/re-initialized in the following circumstance:

- The Pintos kernel starts up, and creates the first user process
- How the Pintos kernel jumps to the first user process that it creates
- How the Pintos kernel regains control on syscalls/interrupts/exceptions from user processes
- How Pintos user processes create other Pintos processes
- How Pintos user processes context switch between each other

All of these actions have analogs in the existing non-floating-point code: that's how the kernel is able to run code at all, of course! It may prove helpful to identify where each of the above items are handled in the existing codebase, after which point you can consider what needs to be done to extend each of those locations to support floating-point operations as well.

4.2.7 FPU Testing

Our testing code uses an extremely simplified version of `<stdio.h>` to print floating point values. You can see how it is implemented in `lib/stdio.c`. As a result, printing very small or very large floating point numbers will cause errors. Our code only allows floating point values of the form `<whole>.<decimal>`, where `<whole>` and `<decimal>` each fit into a standard 32-bit integer. If you intend to write your own tests, please keep this in mind.

4.3 System Calls

4.3.1 System Call Overview

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

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

[8]https://docs.oracle.com/cd/E18752_01/html/817-5477/eoizy.html#epmnw
[9]http://ref.x86asm.net/coder32.html

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

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

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

4.3.2 FPU System Calls

For Task 1, you will need to implement the following system calls:

System Call: double compute_e(int n) A "fake" system call that doesn't exist in any modern OS. You will implement this to get familiar with the system call interface and to see an example of floating point instructions in use. This system call simply computes n terms of the Taylor summation of e. Most of the work of this system call has been implemented for you; it shouldn't be much harder than the `practice` system call below. See 4.2.4 FPU Syscall for more details.

4.3.3 Process System Calls

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

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

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

System Call: void exit (int status) Terminates the current user program, returning status to the kernel. If the process's parent waits for it (see below), this is the status that will be returned. Conventionally, a status of 0 indicates success and nonzero values indicate errors. Every user program that finishes in the normal way calls `exit`—even a program that returns from `main()` calls `exit` indirectly (see `start()` in `lib/user/entry.c`). In order to make the test suite pass, you need to print out the exit status of each user program when it exits. The code should look like: "`printf("%s: exit(%d)\n", thread_current()->name, exit_`

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

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

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

- pid does not refer to a direct child of the calling process. pid is a direct child of the calling process if and only if the calling process received pid as a return value from a successful call to exec. Note that children are not inherited: if A spawns child B and B spawns child process C, then A cannot wait for C, even if B is dead. A call to wait(C) by process A must fail. Similarly, orphaned processes are not assigned to a new parent if their parent process exits before they do.
- The process that calls wait has already called wait on pid. That is, a process may wait for any given child at most once.

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

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

Warning: Implementing this system call will likely require considerably more work than any of the rest.

4.3.4 File System Calls

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

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

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

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

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

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

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

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

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

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

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

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

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

A seek past the current end of a file is not an error. A later read obtains 0 bytes, indicating end of file. A later write extends the file, filling any unwritten gap with zeros. (However, in Pintos files have a fixed length until Project 3 is complete, so writes past end-of-file will return an error.) These semantics are implemented in the file system and do not require any special effort in system call implementation.

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

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

4.4 FAQ

How much code will I need to write? Here's a summary of our reference solution, produced by `git diff --stat`.

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

```
src/lib/float.h          |   6 +
src/threads/init.c       |   9 +
src/threads/interrupt.h  |   1 +
src/threads/intr-stubs.S |   6 +-
src/threads/start.S      |   5 +-
src/threads/switch.S     |   4 +
src/threads/switch.h     |   5 +-
src/threads/thread.c     |  12 +-
src/threads/thread.h     |  23 ++
src/userprog/exception.c |   7 +
src/userprog/process.c   | 242 +++++++++++++++---
src/userprog/process.h   |   1 +
src/userprog/syscall.c   | 418 ++++++++++++++++++++++++++++++-
src/userprog/syscall.h   |   1 +
14 files changed, 681 insertions(+), 59 deletions(-)
```

The kernel always panics when I run the test case I wrote for "Student Testing Code" Is your file name too long? The file system limits file names to 14 characters. Is the file system full? Does the file system already contain 16 files? The base Pintos file system has a 16-file limit.

The kernel always panics with "assertion `is_thread(t)` failed" This happens when you overflow your kernel stack. If you're allocating large structures or buffers on the stack, try moving them to static memory or the heap instead. It's also possible that you've made your `struct thread` too large. See the comment underneath the kernel stack diagram in `src/threads/thread.h` about the importance of keeping your `struct thread` small.

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

All my user programs die upon making a system call! You'll have to implement system calls before you see anything else. Every reasonable program tries to make at least one system call (`exit()`) and most programs make more than that. Notably, `printf()` invokes the `write()` system call. The default system call handler just handles `exit()`. Until you have implemented system calls sufficiently, you can use `hex_dump()` to check your argument passing implementation (see Program Startup Details).

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

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

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

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

How do I compile new user programs? Modify `src/examples/Makefile`, then run `make`.

Can I run user programs under a debugger? Yes, with some limitations. See the section of the project 0 specification[10] on debugging.

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

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

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

[10] https://cs162.org/static/projects/project0.pdf#page=10

4.4.1 Argument Passing FAQ

Isn't the top of stack in kernel virtual memory? The top of stack is at PHYS_BASE, typically 0xc0000000, which is also the point whence kernel virtual memory begins, continuing upwards to the top of the address space. But before the processor pushes data on the stack, it decrements the stack pointer. Thus, the first (4-byte) value pushed on the stack will be at address 0xbffffffc.

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

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

Can I enforce a maximum size on the arguments list? You can set a reasonable limit on the size of the arguments — as long as it has enough to pass all of our tests, it should be fine, but note that both Project 2 and Project 3 rely on this code as well, so make sure you aren't cutting things too close, or you may have to come back and revisit that limit later on in the semester.

4.4.2 System Calls FAQ

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

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

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

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

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

5 Advice

5.1 General advice

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

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

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

5.2 Group work

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

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

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

5.3 Development advice

5.3.1 Compiler Warnings

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

The skeleton code we have provided you triggers only one warning (which cannot be disabled, regarding the `main` function) which you may safely ignore. **If your code is buggy, the first thing you should do is check to see if the compiler is emitting any warnings.** While sometimes warnings might be emitted for code that is perfectly fine, in general it's best to remedy your code to fix warnings whenever you see them.

5.3.2 Faster Compilation

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

Please be warned however that **you should only pass the -j flag to make when compiling your code**. You should *not* use it when running your tests with make check. While it is actually safe to do so for Project 1, this is not the case for future projects, so it's best not to get into the habit of it.

5.3.3 When All Else Fails

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

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

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