G726004 Project 1: User Programs

Design Document Due:	周四, 2019-12-26
Code Checkpoint #1:	周四, 2020-01-02
Code Checkpoint #2:	周四, 2020-01-09
Code Due:	周四, 2020-01-16
Final Report Due:	周四, 2020-01-16

Contents

1 Your Task **3**
- 1.1 Task 1: Argument Passing ... 3
- 1.2 Task 2: Process Control Syscalls ... 3
- 1.3 Task 3: File Operation Syscalls ... 4

2 Deliverables **4**
- 2.1 Design Document (Due 09/17) and Design Review ... 4
  - 2.1.1 Getting Acquainted with Pintos ... 5
  - 2.1.2 Design Overview ... 8
  - 2.1.3 Additional Questions ... 8
  - 2.1.4 Design Review ... 9
  - 2.1.5 Grading ... 9
- 2.2 Code (Due 09/23, 09/30, 10/04) ... 9
- 2.3 Checkpoint #1 (Due 09/23) ... 9
- 2.4 Checkpoint #2 (Due 09/30) ... 10
- 2.5 Final Code (Due 10/04) ... 10
  - 2.5.1 Student Testing Code (Due 10/04) ... 10
- 2.6 Student Testing Report (Due 10/07) ... 10
- 2.7 Final Report (Due 10/07) and Code Quality ... 12

3 Reference **13**
- 3.1 Pintos ... 13
  - 3.1.1 Getting Started ... 13
  - 3.1.2 Overview of the Pintos Source Tree ... 13
  - 3.1.3 Building Pintos ... 15
  - 3.1.4 Running Pintos ... 15
  - 3.1.5 Formatting and Using the File System from the Pintos Command Line ... 16
  - 3.1.6 Using the File System from the Pintos Kernel ... 16
  - 3.1.7 Debugging Pintos ... 17
  - 3.1.8 Debugging Pintos Tests ... 18
  - 3.1.9 Debugging Page Faults ... 19
  - 3.1.10 Debugging Kernel Panics ... 19
  - 3.1.11 Adding Source Files ... 20
  - 3.1.12 Why Pintos? ... 20

3.2 User Programs 20
3.2.1 Overview of Source Files for Project 1 20
3.2.2 How User Programs Work 21
3.2.3 Virtual Memory Layout 21
3.2.4 Accessing User Memory 22
3.2.5 80x86 Calling Convention 23
3.2.6 Program Startup Details 24
3.2.7 Adding New Tests to Pintos 25
3.3 System Calls 26
3.3.1 System Call Overview 26
3.3.2 Process System Calls 26
3.3.3 File System Calls 27
3.4 FAQ 28
3.4.1 Argument Passing FAQ 30
3.4.2 System Calls FAQ 30
3.5 Debugging Tips 31
3.5.1 printf 31
3.5.2 ASSERT 31
3.5.3 Function and Parameter Attributes 31
3.5.4 Backtraces 32
3.5.5 GDB 34
3.5.6 Triple Faults 36
3.5.7 General Tips 36
3.6 Advice 37

Welcome to the first project of G726004! Our projects in this class will use Pintos, an educational operating system. They 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 G726004 will be organized as follows. For clarity, the details of the assignment itself will be in the Your Task section ar 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 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.

1.1 Task 1: Argument Passing

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

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.

1.2 Task 2: Process Control Syscalls

Pintos currently only supports one syscall: exit. You will add support for the following new syscalls: halt, exec, wait, and practice. Each of these syscalls has a corresponding function inside the user-level library in pintos/src/lib/user/syscall.c|[1], which prepares the syscall arguments and handles the transfer to kernel mode. The kernel's syscall handler is located in pintos/src/userprog/syscall.c|[2].

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 child process to exit.

To implement syscalls, you first need a way to safely read and write memory that's in user process's virtual address space. The syscall arguments are located on the user process's stack, right above the user process stack pointer. 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 the 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 pointer could be invalid as well.

You will need to gracefully handle cases where a syscall cannot be completed, because of invalid memory access. These kinds of memory errors include null pointers, invalid pointers (which point to unmapped memory locations), or pointers to the kernel's virtual address space. Beware: a 4-byte memory

[1]https://github.com/Berkeley-CS162/group0/blob/master/pintos/src/lib/user/syscall.c
[2]https://github.com/Berkeley-CS162/group0/blob/master/pintos/src/userprog/syscall.c

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.

1.3 Task 3: File Operation Syscalls

In addition to the process control syscalls, you will also need to implement these file operation syscalls: create, remove, open, filesize, read, write, seek, tell, and close. Pintos already contains a basic 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 the entire 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 ensure that you deny writes to current-running program files. 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.

2 Deliverables

Your project grade will be made up of 4 components:

- 15% Design Document and Design Review
- 60% Code（必须有分析报告）
- 15% Student Testing
- 10% Final Report and Code Quality

2.1 Design Document (Due 12/26) 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.

Write your design document inside the doc/project1.md file, which has already been placed in your group's GitHub repository. You must use GitHub Flavored Markdown[3] to format your design document. You can preview your design document on GitHub's web interface by going to the following address: (replace group0 with your group number)

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

[3]https://help.github.com/articles/basic-writing-and-formatting-syntax/

There are three parts to the design document. The first part is the Getting Acquainted with Pintos exercise, where you should work through the exercise and include answers to all of the questions. The second part is a Design Overview where you will include an overview of your proposed design for completing Project 1. The third part is to answer some Additional Questions. We explain each part of the design document in detail in the sections below.

2.1.1 Getting Acquainted with Pintos

The nature of the Pintos projects is that one must have a good understanding of the existing codebase in order to design good solutions for the projects. The goal of this exercise is to help you develop some familiarity with the Pintos code.

First, run make and make check in the pintos/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 pintos/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 pintos/src/userprog/build/tests/userprog/do-nothing. The do-nothing test simply runs the do-nothing executable in Pintos using pintos run (see Running Pintos).

View the file pintos/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 (see Process System Calls). 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 in your design doc:

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 into the design doc, 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 pintos/src/lib or pintos/src/lib/user), and copy it into your design doc. 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.

Now that we understand why the do-nothing program crashes, we will step through the execution of the do-nothing test in Pintos, starting from when the kernel boots, in GDB. 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 pintos/src/userprog/ and run

```
pintos --gdb --filesys-size=2 -p ./build/tests/userprog/do-nothing -a do-nothing --
    -q -f run do-nothing
```

In another terminal window, change the working directory to pintos/src/userprog/build, start GDB (pintos-gdb ./kernel.o), and attach it to the Pintos process (debugpintos). If any of these instructions are not clear, please take another look at Debugging Pintos and Debugging Pintos Tests.

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 (pintos/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 (pintos/src/threads/start.S).
- The code at the kernel entrypoint switches to 32-bit protected mode[4] and then calls main() (pintos/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 pintos/src/userprog/process.c.

Now, answer the following questions.

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 threads. (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 threads.
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.

[4] https://en.wikipedia.org/wiki/Protected_mode

12. Once you've executed `iret`, type `info registers` to print out the contents of registers. Include the output of this command in your design doc. 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 `pintos-gdb ./kernel.o` 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`.

The faulting instruction you observed in GDB should match the one you found in #3. Now that you have determined the faulting instruction, understand 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 `pintos/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. 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 `pintos/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.

16. The `int $0x30` instruction switches to kernel mode and pushes an interrupt stack frame onto the kernel stack. 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?

17. Step into `thread_exit()`[5], and then step into `process_exit()`. What is the purpose of the `temporary` semaphore? As you can see, `process_exit()` calls `sema_up (&temporary);`; where is the corresponding call to `sema_down (&temporary);`? (Hint: semaphores are covered in Lecture 4[6] on September 10, 2019; feel free to review the slides.)

18. Set a breakpoint for the line just after the call to `sema_down` that you just identified, and continue. If you did this correctly, you should hit the breakpoint you just set. What is the name and address of the thread running this function? What other threads are present in Pintos at this time?

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.

[5]It's important that you step *into* this function call, not *over* it. Stepping over it will freeze your debug session because the function call never returns, but GDB will wait for it to do so.

[6]https://cs162.eecs.berkeley.edu/static/lectures/4.pdf

Be sure to push your code, with the small change you made in order to make the do-nothing test pass, to GitHub. You should now receive full credit for the proj1-do-nothing **assignment on the autograder.**

2.1.2 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, typedefs, or enumerations that you will be adding or modifying (if it already exists). These definitions should be written with the **C programming language**, not with pseudocode. Include a **brief explanation** the purpose of each modification. Your explanations should be as concise as possible. Leave the full explanation to the following sections.

2. **Algorithms** – This is where you tell us how your code will work. Your description should be at a level below the high level description of requirements given in the assignment. We have read the project spec too, so it is unnecessary to repeat or rephrase what is stated here. On the other hand, your description should be at a level above the code itself. Don't give a line-by-line run-down of what code you plan to write. Instead, you should try to convince us that your design satisfies all the requirements, **including any uncommon edge cases**. 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 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.

2.1.3 Additional Questions

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

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

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

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

2.1.4 Design Review

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

2.1.5 Grading

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

2.2 Code (Due 01/02, 01/09, 01/16)

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 design doc, each checkpoint, and the final code. This is how we will track your progress for each checkpoint.**

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. The requirements for these checkpoints are described below. If you're unable to meet the checkpoint deadlines, you can still receive full credit as long as you send an email to your TA explaining your progress so far and seeking help were you need it. 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.

2.3 Checkpoint #1 (Due 01/02)

You must have implemented the following:

- The write syscall for the STDOUT file descriptor only.
- The practice syscall.
- Task 1: 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 the Getting Acquainted with Pintos exercise 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[7] when the exit syscall is made from userspace, but not if it the process exits due to an invalid memory access.

2.4 Checkpoint #2 (Due 01/09)

In addition to the tasks in Checkpoint #1 (Due 09/23), you must have completed Task 2: Process Control Syscalls in its entirety.

2.5 Final Code (Due 01/16)

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

2.5.1 Student Testing Code (Due 01/16)

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 pintos/src/userprog/ directory.

2.6 Student Testing Report (Due 01/16)

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. Place your Student Testing Report inside reports/project1.md, alongside your final report.

Make sure your Student Testing Report contains the following:

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

[7] https://github.com/Berkeley-CS162/group0/blob/master/pintos/src/userprog/syscall.c#L32

- Tell us about your experience writing tests for Pintos. What can be improved about the Pintos testing system? (There's a lot of room for improvement.) What did you learn from writing test cases?

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

2.7 Final Report (Due 01/16) and Code Quality

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

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

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

- Does your code exhibit any major memory safety problems (especially regarding C strings), memory leaks, poor error handling, or race conditions?
- Did you use consistent code style? Your code should blend in with the existing Pintos code. Check your use of indentation, your spacing, and your naming conventions.
- Is your code simple and easy to understand?
- If you have very complex sections of code in your solution, did you add enough comments to explain them?
- Did you leave commented-out code in your final submission?
- Did you copy-paste code instead of creating reusable functions?
- Did you re-implement linked list algorithms instead of using the provided list manipulation
- Are your lines of source code excessively long? (more than 100 characters)
- Is your Git commit history full of binary files? (don't commit object files or log files for this project)

3 Reference

3.1 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 G726004 student a dedicated PC for use with Pintos. Therefore, we will run Pintos projects in a system simulator, that is, a program that simulates an x86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it. Simulators also give you the ability to inspect and debug an operating system while it runs. In class we will use the Bochs[8] and QEMU[9] simulators.

3.1.1 Getting Started

Log in to the Vagrant Virtual Machine that you set up in hw0. You should already have a copy of the Pintos skeleton code in ~/code/group on your VM. Since this is your first group project, you will need to link this local git repository to your group's GitHub repository. Go to the "Group" section of the autograder[10] and click the Octocat icon to go to your group's GitHub repository. Copy the **SSH Clone URL** (it should look like "git@github.com:Berkeley-CS162/groupX.git") and use it in the commands below:

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

Once you have made some progress on your project, you can push your code to the autograder by pushing to "group master". This will use the "group" remote that we just set up. You don't have to do this right now, because you haven't made any progress yet.

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

To compile Pintos and run the Project 1 tests:

```
$ cd ~/code/group/pintos/src/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.

3.1.2 Overview of the Pintos Source Tree

The Pintos source code in ~/code/group/pintos/src/ is organized into the following subdirectories:

threads/
The base Pintos kernel, including the bootloader, kernel entrypoint, base interrupt handler, page allo- cator, subpage memory allocator, and CPU scheduler. Most of your code for Project 2 will be in this

[8] https://en.wikipedia.org/wiki/Bochs
[9] https://en.wikipedia.org/wiki/QEMU
[10] https://cs162.eecs.berkeley.edu/autograder/dashboard/group/

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 both the Pintos kernel and user programs that run inside it. You can include header files from this directory using the `#include <...>` notation.[11] 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.

[11] 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.

3.1.3 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 back-trace 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.

build/loader.bin

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

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

3.1.4 Running Pintos

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

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

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

The number next to each action's name tells Pintos how many arguments there are, including the name of the action itself. There are also additional flags that the kernel accepts, which you can see in

the usage function. The preprocessor flags USERPROG and FILESYS, used in the threads directory, are enabled in Projects 1 and 3. In Project 2 we will disable these flags. We won't enable the VM macro in any of our projects in this class.

The run action accepts a program invocation, like echo hello, and executes it. This is implemented in the run_task function in test/init.c. Because run_task uses process_execute, the run action does initially support program invocations that involve command-line arguments. Only once you implement argument passing in process_execute (Task 1) will this behavior be supported.

Before you can use the run action to invoke a program, you need to have a file system with the executable you want to run (e.g., echo). The file-system-related actions (e.g., ls, cat, etc.) and other flags can be used to create such a file system, as described below.

3.1.5 Formatting and Using the File System from the Pintos Command Line

During the development process, you may need to be able to create a simulated disk with a file system partition. The pintos-mkdisk program provides this functionality. From the userprog/build directory, execute pintos-mkdisk filesys.dsk --filesys-size=2. This command creates a simulated disk named filesys.dsk that contains a 2 MB Pintos file system partition. Then format the file system partition by passing -f -q on the kernel's command line: pintos -f -q. The -f option causes the file system to be formatted, and -q causes Pintos to exit as soon as the format is done.

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

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

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

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

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

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

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

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

3.1.6 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.

3.1.7 Debugging Pintos

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

One of the most important options is `--gdb` which will allow you to use GDB to debug the code you've written. For example, to run the debugger for the `do-nothing` test we would perform the following steps:

- cd into `~/code/group/pintos/src/userprog`
- Run Pintos with the debugger option "`pintos --gdb -- run do-nothing`". At this point, Pintos should say that it is "Waiting for gdb connection on port 1234".
- Open a new terminal and SSH into the VM
- cd into `~/code/group/pintos/src/userprog/build`
- Open cgdb by running "`pintos-gdb kernel.o`". The `pintos-gdb` script loads `cgdb` with many useful GDB macros that will help you debug Pintos.
- In GDB, attach to Pintos by running "`target remote localhost:1234`".
  On your VM, you should be able to use "`debugpintos`" or "`deb`" as a shortcut to performing this step.
- Set a breakpoint at an interesting point in the program. Try "`break start_process`".
- Use "`continue`" or "`c`" to start Pintos.

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

3.1.8 Debugging Pintos Tests

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

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

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

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

```
pintos -v -k -T 60 --qemu --filesys-size=2 -p tests/userprog/do-nothing -a do-nothing
    -- -q -f run do-nothing < /dev/null 2> tests/userprog/do-nothing.errors >
    tests/userprog/do-nothing.output
perl -I../.. ../../tests/userprog/do-nothing.ck tests/userprog/do-nothing
    tests/userprog/do-nothing.result
FAIL tests/userprog/do-nothing
```

Here is an explanation:

- The first line runs the pintos script to start the Pintos kernel in a simulator, like we did before. The pintos script consumes the -v -k -T 60 --qemu arguments and passes the -q –f run do-nothing arguments to the Pintos kernel. Then, we use the <, 2>, and > symbols to redirect standard input, standard error, and standard output to files.
- The second line uses the Perl programming language to run ../../tests/threads/do-nothing.ck to verify the output of Pintos kernel.
- Using the Perl script from line 2, the build system can tell if this test passed or failed. If the test passed, we will see "pass", followed by the test name. Otherwise, we will see "FAIL", followed by the test name, and then more details about the test failure.

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

Your final command should look like:

```
$ pintos --gdb -v -k -T 60 --qemu --filesys-size=2 -p tests/userprog/do-nothing -a
    do-nothing -- -q -f run do-nothing
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

You can transform it into this:

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

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

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

3.1.11 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.

3.1.12 Why Pintos?

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

3.2 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).

3.2.1 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.

3.2.2 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 Formatting and Using the File System from the Pintos Command Line), 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.

3.2.3 Virtual Memory Layout

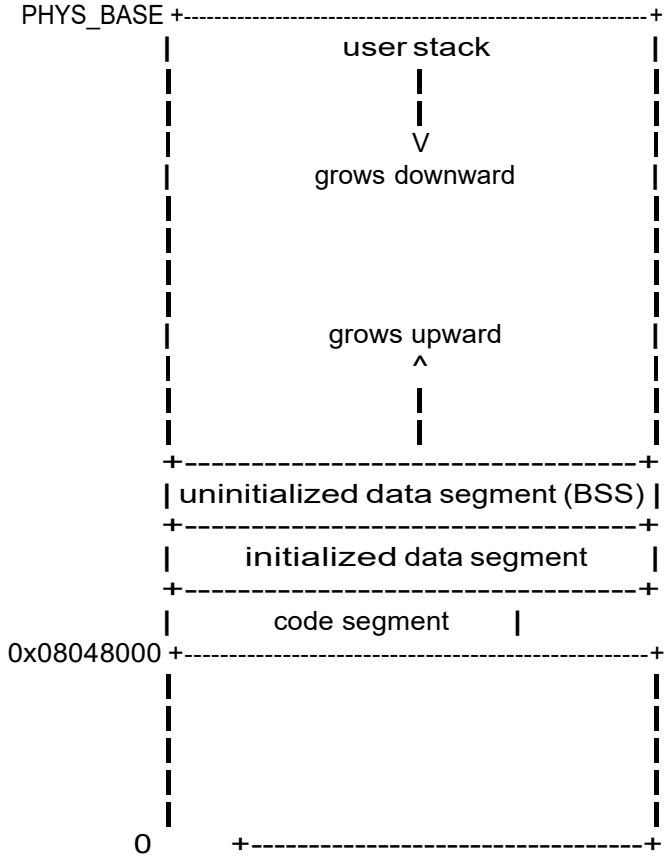
Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address 0 up to PHYS_BASE, which is defined in threads/vaddr.h and defaults to 0xc0000000 (3 GB). Kernel virtual memory occupies the rest of the virtual address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



3.2.4 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\verb. 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 would be possible than using get_user and put_user to handle each byte.

3.2.5 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*) on the stack and jumps to the first instruction of the callee. A single 80x86 instruction, CALL, does both.

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

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

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

6. The caller pops the arguments off the stack.

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

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

3.2.6 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:

```
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 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.|
```

3.2.7 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. 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 4MB of memory and the file system block device is 2MB by default. Don't use data structures or files that exceed these sizes.
- Your test should use msg() instead of printf() (they have the same function signature).

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

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

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

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

3.3 System Calls

3.3.1 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).

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

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

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

3.3.2 Process System Calls

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.3 File System Calls

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

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

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

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

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

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

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

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

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

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

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

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

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

A seek past the current end of a file is not an error. A later read obtains 0 bytes, indicating end of file. A later write extends the file, filling any unwritten gap with zeros. (However, in Pintos files have a fixed length until Project 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.

3.4 FAQ

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

The reference solution represents just one possible solution. Many other solutions are also possible and many of those differ greatly from the reference solution. Some excellent solutions may not

modify all the files modified by the reference solution, and some may modify files not modified by the reference solution.

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

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

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

Is the file system full?

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

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

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

You can list the files in your file system with `pintos -q ls`. The base Pintos file system does not support directories.

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

All my user programs die with system call! You'll have to implement system calls before you see anything else. Every reasonable program tries to make at least one system call (`exit()`) and most programs make more than that. Notably, `printf()` invokes the `write()` system call. The default system call handler just prints system call and 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 this spec on GDB macros.

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

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

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

3.4.1 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 where kernel virtual memory starts. But before the processor pushes data on the stack, it decrements the stack pointer. Thus, the first (4-byte) value pushed on the stack will be at address 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.

3.4.2 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 stack space? You may modify the stack setup code to allocate more than one page of stack space for each process. This is not required in this project.

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

3.5 Debugging Tips

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

3.5.1 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 twopoints, and so on. Based on what you learn, you can then insert more printf calls in the new, smaller region of code you suspect. Eventually you can narrow the problem down to a single statement. See section Triple Faults, for a related technique.

3.5.2 ASSERT

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

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

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

3.5.3 Function and Parameter Attributes

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

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

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

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

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

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

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

3.5.5 GDB

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

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

and issue the following GDB command:

```
target remote localhost:1234
```

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

Using GDB

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

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

break function

break file:line

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

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

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

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

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

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

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.

3.5.6 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.

3.5.7 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.

3.6 Advice

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/spot subtle bugs.

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

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

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