

CS 302 Project 2: User Programs

Due: 06/17/2018 week 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	3
2	Deliverables	4
2.1	Design Document and Design Review	4
2.1.1	Design Document Guidelines	4
2.1.2	Design Document Additional Questions	5
2.1.3	GDB Questions	5
2.1.4	Design Review	6
2.2	Code (Due 06/17)	7
2.2.1	Hack Testing (Due 06/17)	7
2.3	Hack Testing Report(Due 06/17)	7
2.4	Final Report (Due 06/17) and Code Quality	8
3	Reference	9
3.1	Pintos	9
3.1.1	Getting Started	9
3.1.2	Source Files	9
3.1.3	Using the File System	10
3.1.4	Formatting and Using the Filesystem	10
3.1.5	How User Programs Work	11
3.1.6	Virtual Memory Layout	11
3.1.7	Accessing User Memory	12
3.1.8	80x86 Calling Convention	13
3.1.9	Program Startup Details	14
3.1.10	Adding new tests to Pintos	15
3.2	System Calls	16
3.2.1	System Call Overview	16
3.2.2	Process System Calls	16

3.2.3	File System Calls	17
3.3	FAQ	18
3.3.1	Argument Passing FAQ	20
3.3.2	System Calls FAQ	20

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 and start user programs, 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`.

All 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, none of the user programs will 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, `lib/user/syscall.c`, which prepares the syscall arguments and handles the transfer to kernel mode. The kernel's syscall handlers are located in `userprog/syscall.c`.

The `halt` syscall will shutdown 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. The `practice` syscall just adds 1 to its first argument, and returns the result (to give you practice writing a syscall handler).

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. If the stack pointer is invalid when the user program makes a syscall, the kernel cannot crash while trying to dereference an invalid or null pointer. 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: It may be the case that a 4-byte memory region (like a 32-bit integer) consists 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 3.1.7 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 filesystem. 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 filesystem is not thread-safe. You must make sure that your file operation syscalls do not call multiple filesystem functions concurrently. In Project 3, you will add more sophisticated synchronization to the Pintos filesystem, but for this project, you are permitted to use a global lock on filesystem operations, to ensure thread safety. We recommend that you avoid modifying the `filesystem/` 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.

Note: Your final code for Project 2 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. You should keep this in mind while designing your implementation for this project.

2 Deliverables

Your project grade will be made up of 3 components:

- 60% Code
- 10% Hack Testing – This is a new component of your project grade (see explanation below).
- 30% Final Report and Code Quality

2.1 Design Document and Design Review

Before you start writing any code for your project, you should create an implementation plan for each feature and convince yourself that your design is correct. For this project, you can write a **design document** and **attend a design review** with your project TA. This document is not necessary. You can either write it or not.

2.1.1 Design Document Guidelines

Write your design document inside the `doc/project2.md` file, which has already been placed in your group’s GitHub repository. You must use GitHub Flavored Markdown¹ 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/project2.md>

For each of the first 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, 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**. This section should be similar in style and format to the design document you submitted for Project 1. 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.

¹<https://help.github.com/articles/basic-writing-and-formatting-syntax/>

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 inside of the scheduler, in 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 parallelism of the kernel. 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.
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.

2.1.2 Design Document Additional Questions

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

1. Take a look at the Project 2 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.)
4. Answer the GDB questions below.

2.1.3 GDB Questions

To be successful at Pintos, a strong familiarity with GDB is extremely helpful. As such, in this design doc, we will ask some questions about Pintos which can be answered with GDB. Please make sure that each member of your group does this section of the design doc. (Feel free to work together of course!)

We will use GDB on the first test of project 2 – `args-none`. To get started please first build project 2 by running `make check` in the `src/userprog` directory. Then start Pintos with the `args-none` test by doing

```
pintos --gdb --fileysys-size=2 -p tests/userprog/args-none -a args-none \  
-- -q -f run args-none
```

Finally, start GDB (`pintos-gdb build/kernel.o`) and attach it to the Pintos process (`debugpintos`). If any of these instructions are not clear, please take another look at the GDB section of Project 1's spec.

The questions you should answer in your design doc are the following.

1. Set a break point at `process_execute` 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`. (Hint: for the last part `dumplist &all_list thread allelem` may be useful.)
2. 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.
3. 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`.
4. Where is the thread running `start_process` created? Copy down this line of code.
5. Continue one more time. The userprogram should cause a page fault and thus cause the page fault handler to be executed. It'll look something like

```
[Thread <main>] #1 stopped.  
pintos-debug: a page fault exception occurred in user mode  
pintos-debug: hit 'c' to continue, or 's' to step to intr_handler  
0xc0021ab7 in intr0e_stub ()
```

Please find out what line of our user program caused the page fault. Don't worry if it's just an hex address. (Hint: `btpagefault` may be useful)

6. The reason why `btpagefault` returns an hex address is because `pintos-gdb build/kernel.o` only loads in the symbols from the kernel. The instruction that caused the page fault is in our userprogram so we have to load these symbols into gdb. To do this use

```
loadusersymbols build/tests/userprog/args-none
```

. Now do `btpagefault` again and copy down the results.

7. Why did our user program page fault on this line?

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.

2.2 Code (Due 06/17)

We will use multiple test cases to test your project, and you will get some of them in advance. For each case, we will check whether your program can handle a certain situation. Finally, we will run your program on the full test set (our test set and hack test set). Your ranking will be determined by the number of test cases you passed and your code quality. The score distribution will be determined by our teacher.

2.2.1 Hack Testing (Due 06/17)

Pintos already contains a test suite for Project 2, 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 15 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.

2.3 Hack Testing Report (Due 06/17)

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/project2.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).
- 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.

2.4 Final Report (Due 06/17) and Code Quality

After you complete the code for your project, you will submit a final report. Write your final report inside the `reports/project2.pdf` file. Please include the following in your final report:

- A reflection on the project – what exactly did each member do? What went well, and what could be improved?
- 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)

3 Reference

3.1 Pintos

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).

3.1.1 Getting Started

Log in to the Vagrant Virtual Machine that you set up in hw0. You should already have your Pintos code from Project 1 in `~/code/group` on your VM. You may start Project 2 using your Project 1 code. But you may also start over from the skeleton code if you wish.

If you would like to start over from the skeleton code, please run these commands on your VM:

```
$ cd ~/code/group/
$ git checkout 81c04a231523daaa082204b53bd8d3bc5f4d534f -- pintos/
$ git commit -m "Revert changes to pintos/ from Project 1"
$ git push group master
```

Make sure **your design document and final report for Project 1 is still accessible** from the GitHub website. We will be looking for those documents, so if you do not have them, we might mistakenly think you didn't turn them in. Also, please **do not force push** and please do not delete your commits from Project 1. You should know that orphan commits are still accessible on GitHub, and we have a history of the commit hashes you've pushed to the autograder, but we will not enjoy digging up that information if we need it.

We recommend that you use Git to tag your final Project 1 code, for your own benefit.

Once you have made some progress on your project, you can push your code to the autograder by pushing to "group master". 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 2 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. By the end of the project, your code should pass all of the tests.

3.1.2 Source Files

For this project, you will need to modify some of the same files that you changed in Project 1. You may also need to modify these additional files:

process.c Loads ELF binaries and starts processes.

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 don't expect you to understand this assembly code, but 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.

exception.c Handle exceptions. Currently all exceptions simply print a message and terminate the process. Some, but not all, solutions to Project 2 involve modifying `page_fault()` in this file.

3.1.3 Using the File System

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

- 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.

3.1.4 Formatting and Using the Filesystem

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`

`--fileys-size=2`. This command creates a simulated disk named `fileys.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 --fileysys-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 --fileysys-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 `--fileysys-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 --fileysys-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.

3.1.5 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 Using the File System), 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 `fileysys.dsk` beyond a useful state, which may happen occasionally while debugging.

3.1.6 Virtual Memory Layout

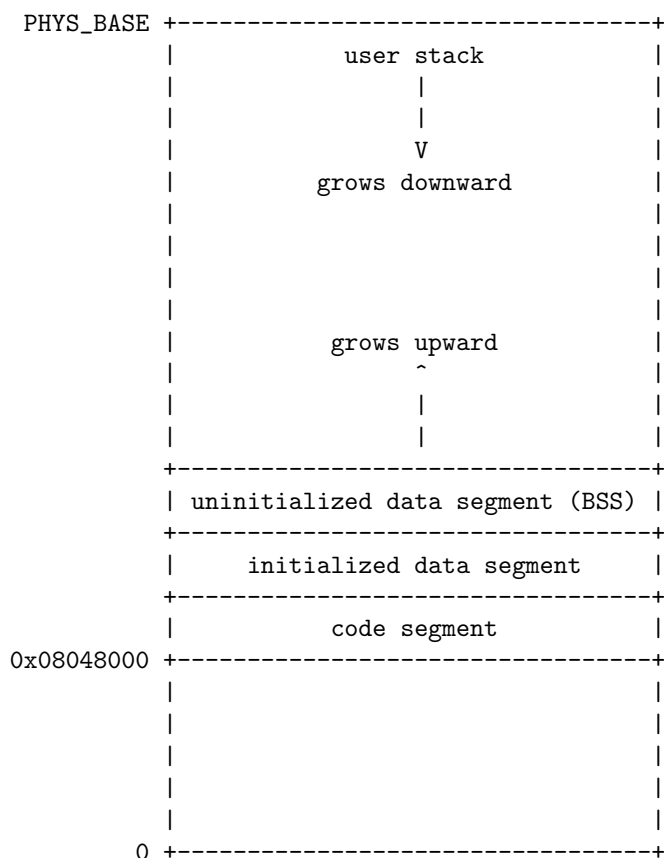
Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address 0 up to `PHYS_BASE`, which is defined in `threads/vaddr.h` and defaults to `0xC0000000` (3 GB). Kernel virtual memory occupies the rest of the virtual address 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:



3.1.7 Accessing User Memory

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

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

- verify the validity of a user-provided pointer, then dereference it. If you choose this route, you'll want to look at the functions in `userprog/pagedir.c` and in `threads/vaddr.h`. This is the simplest way to handle user memory access.
- check only that a user pointer points below `PHYS_BASE`, then dereference it. An invalid user pointer will cause a “page fault” that you can handle by modifying the code for `page_fault()` in

userprog/exception.c. This technique is normally faster because it takes advantage of the processor's MMU, so it tends to be used in real kernels (including Linux).

In either case, you need to make sure not to “leak” resources. For example, suppose that your system call has acquired a lock or allocated memory with `malloc()`. If you encounter an invalid user pointer afterward, you must still be sure to release the lock or free the page of memory. If you choose to verify user pointers before dereferencing them, this should be straightforward. It's more difficult to handle if an invalid pointer causes a page fault, because there's no way to return an error code from a memory access. Therefore, for those who want to try the latter technique, we'll provide a little bit of helpful code:

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

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

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

3.1.8 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
	+-----+

3.1.9 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 section 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. Word-aligned accesses are faster than unaligned accesses, so for best performance round the stack pointer down to a multiple of 4 before the first push.

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

The table below 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
0xbfffffc	argv[3][...]	bar\0	char[4]
0xbfffff8	argv[2][...]	foo\0	char[4]
0xbfffff5	argv[1][...]	-l\0	char[3]
0xbffffed	argv[0][...]	/bin/ls\0	char[8]
0xbffffec	word-align	0	uint8_t
0xbffffe8	argv[4]	0	char *
0xbffffe4	argv[3]	0xbfffffc	char *
0xbffffe0	argv[2]	0xbfffff8	char *
0xbffffdc	argv[1]	0xbfffff5	char *
0xbffffd8	argv[0]	0xbffffed	char *
0xbffffd4	argv	0xbffffd8	char **
0xbffffd0	argc	4	int
0xbffffcc	return address	0	void (*)()

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

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

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

```

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

```

3.1.10 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 filesystem 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.

3.2 System Calls

3.2.1 System Call Overview

The first project already dealt with one way that the operating system can regain control from a user program: interrupts from timers and I/O devices. These are "external" 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.

3.2.2 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.

3.2.3 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 an `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 4 is complete, so writes past end of file will return an error.) These semantics are implemented in the file system and do not require any special effort in system call implementation.

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.

3.3 FAQ

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

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

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

```

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 --`, 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`.

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

All my user programs die with system call! You'll have to implement system calls before you see anything else. Every reasonable program tries to make at least one system call (`exit()`) and most programs make more than that. Notably, `printf()` invokes the write system call. The default system call handler just prints system call! and terminates the program. Until then, you can use `hex_dump()` to convince yourself that argument passing is implemented correctly (see section 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.)

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.

3.3.1 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 `0xbfffffff`.

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.

3.3.2 System Calls FAQ

Can I just cast a struct `file *` to get a file descriptor?

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

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

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

How can I run user programs that need more than 4 kB stack space? You may modify the stack setup code to allocate more than one page of stack space for each process. 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.