CS 162 Project 2: `User Programs`

Initial Design Document Due:
Monday, March 28, 2016
Code Due:
Monday, April 11, 2016
Final Report Due:
Monday, April 11, 2016

Contents

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 an 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 `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.

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 4 components:

- 20% Design Document and Design Review
- 55% Code
- **15% Student Testing** – This is a new component of your project grade (see explanation below).
- 10% Final Report and Code Quality

2.1 Design Document (Due 3/28) 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.

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 Markdown[1] 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.)

2.1.3 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.

2.1.4 Grading

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

2.2 Code (Due 4/11)

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.

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.

2.3 Student Testing (Due 4/11)

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. You will 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 4/11) 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.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, unless you actually intend to)

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 448edfde949b4ce35f30cc49af350e7a44401815 -- 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 filesys.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 --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.

3.1.5 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 filesys.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:

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

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
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	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

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

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

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

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

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 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 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 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.

3.3.2 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.

3.4 Debugging Tips

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

3.4.1 printf

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

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

3.4.2 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.

3.4.3 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.

3.4.4 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.
```

3.4.5 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 2, 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 break point 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.

Sample GDB section This section narrates a sample gdb session, provided by Godmar Back. This example illustrates how one might debug a project 1 solution in which occasionally a thread that calls `timer_sleep` is not woken up. With this bug, tests such as `mlfqs_load_1` get stuck.

This session was captured with a slightly older version of Bochs and the gdb macros for Pintos, so it looks slightly different than it would now.

First, I start Pintos:

```
 $ pintos -v --gdb -- -q -mlfqs run mlfqs-load-1

writing command line to /tmp/gdalqtb5uf.dsk...
Bochs -q
========================================================================
Bochs x86 emulator 2.2.5
build from cvs snapshot on december 30, 2005
========================================================================
00000000000i[     ] reading configuration from Bochsrc.txt
00000000000i[     ] enabled gdbstub
00000000000i[     ] installing nogui module as the Bochs gui
00000000000i[     ] using log file Bochsout.txt
waiting for gdb connection on localhost:1234
```

Then, I open a second window on the same machine and start gdb:

```
$ pintos-gdb kernel.o

gnu gdb red hat linux (6.3.0.0-1.84rh)
copyright 2004 free software foundation, inc.
gdb is free software, covered by the gnu general public license, and you are
welcome to change it and/or distribute copies of it under certain conditions.
type "show copying" to see the conditions.
there is absolutely no warranty for gdb.  type "show warranty" for details.
this gdb was configured as "i386-redhat-linux-gnu"...
using host libthread_db library "/lib/libthread_db.so.1".
```

Then, I tell gdb to attach to the waiting Pintos emulator:

```
(gdb) debugpintos
remote debugging using localhost:1234
0x0000fff0 in ?? ()
reply contains invalid hex digit 78
```

Now I tell Pintos to run by executing `c` (short for `continue`):

Now Pintos will continue and output:

```
pintos booting with 4,096 kb ram...
kernel command line: -q -mlfqs run mlfqs-load-1
374 pages available in kernel pool.
373 pages available in user pool.
calibrating timer...  102,400 loops/s.
boot complete.
executing 'mlfqs-load-1':
```

```
(mlfqs-load-1) begin
(mlfqs-load-1) spinning for up to 45 seconds, please wait...
(mlfqs-load-1) load average rose to 0.5 after 42 seconds
(mlfqs-load-1) sleeping for another 10 seconds, please wait...
```

Until it gets stuck because of the bug I had introduced. I hit `ctrl+c` in the debugger window:

```
program received signal 0, signal 0.
0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
649    while (i <= pri_max && list_empty (&ready_list[i]))
(gdb)
```

The thread that was running when I interrupted Pintos was the idle thread. If i run `backtrace`, it shows this backtrace:

```
(gdb) bt
#0  0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
#1  0xc0101778 in schedule () at ../../threads/thread.c:714
#2  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#3  0xc0101419 in idle (aux=0x0) at ../../threads/thread.c:551
#4  0xc010145a in kernel_thread (function=0xc01013ff , aux=0x0)
at ../../threads/thread.c:575
#5  0x00000000 in ?? ()
```

Not terribly useful. What I really like to know is what's up with the other thread (or threads). Since I keep all threads in a linked list called `all_list`, linked together by a `struct list_elem` member named `allelem`, I can use the `btthreadlist` macro from the macro library I wrote. `btthreadlist` iterates through the list of threads and prints the backtrace for each thread:

```
(gdb) btthreadlist &all_list allelem
pintos-debug: dumping backtrace of thread 'main' @@0xc002f000
#0  0xc0101820 in schedule () at ../../threads/thread.c:722
#1  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#2  0xc0104755 in timer_sleep (ticks=1000) at ../../devices/timer.c:141
#3  0xc010bf7c in test_mlfqs_load_1 () at ../../tests/threads/mlfqs-load-1.c:49
#4  0xc010aabb in run_test (name=0xc0007d8c "mlfqs-load-1")
at ../../tests/threads/tests.c:50
#5  0xc0100647 in run_task (argv=0xc0110d28) at ../../threads/init.c:281
#6  0xc0100721 in run_actions (argv=0xc0110d28) at ../../threads/init.c:331
#7  0xc01000c7 in main () at ../../threads/init.c:140

pintos-debug: dumping backtrace of thread 'idle' @@0xc0116000
#0  0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
#1  0xc0101778 in schedule () at ../../threads/thread.c:714
#2  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#3  0xc0101419 in idle (aux=0x0) at ../../threads/thread.c:551
#4  0xc010145a in kernel_thread (function=0xc01013ff , aux=0x0)
at ../../threads/thread.c:575
#5  0x00000000 in ?? ()
```

In this case, there are only two threads, the idle thread and the main thread. The kernel stack pages (to which the `struct thread` points) are at `0xc0116000` and verb—0xc002f000—, respectively. The main thread is stuck in `timer_sleep`, called from `test_mlfqs_load_1`.

Knowing where threads are stuck can be tremendously useful, for instance when diagnosing deadlocks or unexplained hangs.

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.

3.4.6 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 what I call "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.

3.4.7 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.