南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS 302

Operating System

Spring 2019

Project 2: User Programs

Code Due: April 28, 2019

Final Report Due: April 28, 2019

DBGroup@SUSTech

May 15, 2019

Contents

1 Your task

Now that you've worked with Pintos and are becoming familiar with its infrastructure and thread package, it's time to start working on the parts of the system that allow running user programs. The base code already supports loading and running user programs, but no I/O or interactivity is possible. In this project, you will enable programs to interact with the OS via system calls.

You will be working out of the userprog directory for this assignment, but you will also be interacting with almost every other part of Pintos. We will describe the relevant parts below.

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.

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 Grading Policy

Your project grade will be made up of 2 components, with 100% in total:

- **60%** — Code
- **40%** — Final Report and Code Quality

Project 2 is team project, at most two students per group, but you must have different team members in project 2 and 3.

2.1 Code (Due 4/28)

The code section of your grade will be determined by your score in our final test. Pintos comes with a test suite that you can run locally on your VM and we run the same tests in the final test. Your marks will be a weighted one, after considering your rank in your class.

2.2 Final Report (Due 4/28) and Code Quality

After you complete the code for your project, you will submit a final report. 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?
- Are your lines of source code excessively long? (more than 100 characters)

- Did you re-implement linked list algorithms instead of using the provided list manipulation

3 Submission

3.1 how to register team

Since project 2 is a team homework. We are setting a porject 2 team info at BB, please register your team info at it before **April 7, 18:00**
You can find it by "小组 —> Project 2:groups —> click 注册".

3.2 What to submit

A team submits only one zip file, which can be submitted by any member of the team.

The zip file should:

- Create a folder named as **{student1number-student2number}-project2**.
  Example: 12345678-87654321-project2
  - A folder named **{student1number-student2number}-pintos** for your pintos source code.
    Example: 12345678-87654321-pintos
  - A pdf file named as **{student1number-student2number}-report.pdf** for your final report.
    Example: 12345678-87654321-report.pdf
- Zip the folder and name your zip: The zip should be named as **{student1number-student2number}-project2.zip**.
  Example: 12345678-87654321-project1.zip

3.3 Where to submit and ddl

Upload your pdf and zip file via Blackboard.

The pdf file should be submitted no later than **23:00 (Beijing time) 28 April, 2019**.
The zip file should be submitted no later than **23:00 (Beijing time) 28 April, 2019**.

3.4 Remarks

You should read carefully all requirements of this project.

Prohibition
You will get 0 as score for this project if you break any of the prohibitions:

- You should not Plagiarism.
- You should give clear citation when you use others ideas.
- You should follow the naming policy.
- You should give a clear report for others to read.
- Your code should be well documented for others to understand.

Contact
For any question regarding this project, please email to *11849558@mail.sustc.edu.cn (Jian Zeng)* or *11749127@mail.sustc.edu.cn (Long Xiang)*. The subject of the email should respect the format: **[OS] Project 2 (LastName/FirstName-StudentNumber)**.
Example: [OS] Project 2 (Jian/Zeng-12345678)

4 Reference

4.1 Pintos

Up to now, all of the code you have run under Pintos has been part of the operating system kernel. This means, for example, that all the test code from the last assignment ran as part of the kernel, with full access to privileged parts of the system. Once we start running user programs on top of the operating system, this is no longer true. This project deals with the consequences.

We allow more than one process to run at a time. Each process has one thread (multithreaded processes are not supported). User programs are written under the illusion that they have the entire machine. This means that when you load and run multiple processes

at a time, you must manage memory, scheduling, and other state correctly to maintain this illusion.

In the previous project, we compiled our test code directly into your kernel, so we had to require certain specific function interfaces within the kernel. From now on, we will test your operating system by running user programs. This gives you much greater freedom. You must make sure that the user program interface meets the specifications described here, but given that constraint you are free to restructure or rewrite kernel code however you wish.

4.1.1 Source Files

The easiest way to get an overview of the programming you will be doing is to simply go over each part you'll be working with. In userprog, you'll find a small number of files, but here is where the bulk of your work will be:

`process.c`

`process.h`

Loads ELF binaries and starts processes.

`pagedir.c`

`pagedir.h`

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`

`exception.h`

When a user process performs a privileged or prohibited operation, it traps into the kernel as an "exception" or "fault." These files handle exceptions. Currently all exceptions simply print a message and terminate the process. Some, but not all, solutions to project 2 require

modifying page_fault() in this file.

`gdt.c`

`gdt.h`

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

`tss.c`

`tss.h`

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

4.1.2 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.
- 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.
- 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.
- No subdirectories.

- File names are limited to 14 characters.
- A system crash mid-operation may corrupt the disk in a way that cannot be repaired automatically. There is no file system repair tool anyway.
- When a file is removed (deleted), its blocks are not deallocated until all processes have closed all file descriptors pointing to it. Therefore, a deleted file may still be accessible by processes that have it open.

4.1.3 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.

4.1.4 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.

4.1.5 Virtual Memory Layout

Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address 0 up to PHYS_BASE, which is defined in threads/vaddr.h and defaults to 0xC0000000 (3 GB). Kernel virtual memory occupies the rest of the virtual address space, from PHYS_BASE up to 4 GB.

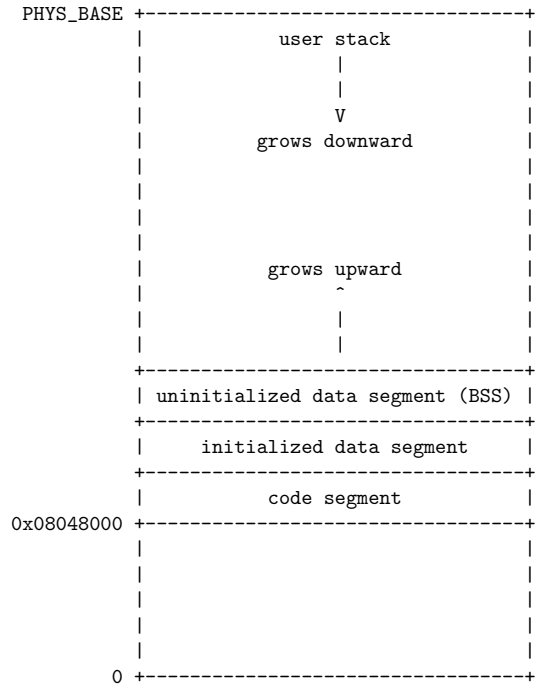
User virtual memory is per-process. When the kernel switches from one process to another, it also switches user virtual address spaces by changing the processor' s page directory base register (see pagedir_activate() in userprog/pagedir.c). struct thread contains a pointer to a process' s page table.

Kernel virtual memory is global. It is always mapped the same way, regardless of what

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

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

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



4.1.6 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).
- 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.
- 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.

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 to 0xffffffff and copies its former value into eip.

4.1.7 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.

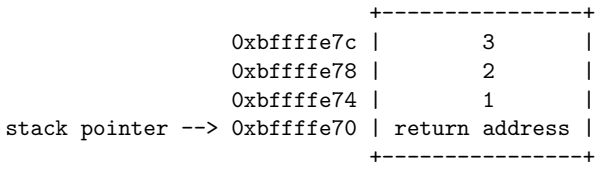
3. The callee executes. When it takes control, the stack pointer points to the return address, the first argument is just above it, the second argument is just above the first argument, and so on.

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:



4.1.8 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. Wordaligned 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.|
```

4.1.9 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.

4.2 System Calls

4.2.1 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.

4.2.2 Process System Calls

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

System Call: in 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 fcon process_wait().

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

4.2.3 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.

4.3 FAQ

How much code will I need to write? Here's a summary of a reference solution. 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.

4.3.1 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.

4.3.2 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.

5 Acknowledgment

This project is originally designed by Stanford for their operating System course. Our project materials are based on the released guide for UC Berkeley CS 162 Operating System. We check the project materials from both schools and combine useful information together to form this one. You can find the original version on internet.