Project 2

Andrew Quinn

Due: Monday May 6th, 2024 at 11:59 PM.
Learning Objectives:
1. Learn how system calls work from an PintOS perspective.
2. Practice "defensive programming".
3. Practice explaining and justifying computer system designs.

OVERVIEW

The goal of this project is to build support for user programs in PintOS. In the last assignment, all of the code that you used (e.g., all of the `alarm` tests) ran as part of the operating system. In this assignment, you will instead support userspace processes. Your job will be to implement the core features required to make this possible.

Your system will need to support more than one process at a time, but, each process will only have a single thread. User processes in your PintOS will execute with "limited direct execution": the illusion that they have access to the entire machine. Your PintOS will need to provide the illusion. The provided code supports loading and running user programs to help get you started. It does not support system calls; adding that support is the main goal of this assignment.

You will be working out of the `src/userprog` directory, meaning that you should build your system from there. However, in this assignment, you will use code from almost every directory in PintOS. It will seem unnervingly complex at first; this is all part of learning.

REQUIREMENTS

Your submission should include a design document, the ability to parse command-line arguments for a user program, the ability to execute system calls made by a user program, the ability to produce a process termination message, and the ability to validate a user's memory input.

This project will be completed in partnerships. However, you will each submit individually on canvas, and will each need to have your group's source code in your individual CSE 134 repository. We suggest that you and your partner choose one of your repositories as "the working repository" (the course staff will ensure that partners have access to each other's CSE134 repositories). You and your partner would do your project development using this repo, pushing at regular intervals as you make progress. Once you are finished, you would push your local repository to the other person's CSE 134 repository. The easiest way to do this is to setup multiple `remotes` in `git`.

One note: we expect to see code commits produced by each partner. It is probably a good idea for you and your partner to "pair program" the assignments. Nonetheless, we will ask questions if the code commits indicate that one person did all of the work.

DESIGN DOCUMENT

Your submission should include a design document that describes key design decisions that you made in your system. The document should be located at `docs/p02.md`. Your design document should include a separate section for each of the four other tasks (argument parsing, system calls, process termination message, and user memory). For each section, you should outline (1) any data structures that you created or extended in your design; (2) any algorithms that you created for your design; (3) any synchronization used in your design; and (4) a justification of your design (Why is it correct? Why is it fast? etc.). You should aim to have enough detail in your design that a fellow 134 student would be capable of re-implementing your system by following it. Note: if your design does not require any of these features (e.g., you do not have data structures for the process termination message), specify "N/A".

COMMAND LINE PARSING

From this assignment onwards, PintOS will call `process_execute()` with the arguments passed to it on the command line (see `run_task()` from `src/threads/init.c`). The argument to `process_execute()` is a c-string including all of arguments to the program. Your first task is to extend this functionality so that `process_execute()` divides its input into words at spaces. The first word is the program name, the second word is the first argument, and so on. Thus `process_execute("grep foo bar")` should run `grep` passing two arguments `foo` and `bar`.

To support command line arguments, your PintOS will need to change the way that it sets up a user process's stack. Peruse through the code in `src/userprog/process.c` and figure out how it currently sets up the stack; you'll want to extend that. Section 3.5, especially the example in Section 3.5.1 at this webpage will probably prove a useful resource for what a stack should look like.

A few other notes:

1. Your PintOS should treat multiple spaces as equivalent to a single space, so that means that `process_execute("grep foo  bar")` is the same as `process_execute("grep foo bar")`
2. Your PintOS should impose a limit on the length of the command line arguments. One reasonable one is to limit arguments to those that fit in a single page (4 kB).
3. Your PintOS can parse argument strings however you would like. There's a few functions in `lib/string.h` that you might find helpful.

SYSTEM CALLS

Implement the system call handler in `userprog/syscall.c`. The current implementation terminates processes on every system call—yours should instead retrieve the system call number and any system call arguments from the user process and carry out the appropriate actions. After this assignment, and forevermore, your PintOS should be bulletproof. Nothing that a user program can do should ever cause it to crash, panic, fail an assertion, or otherwise malfunction. So, program *defensively*.

The system calls can be broadly split into those pertaining to processes and those pertaining to files. PintOS provides a basic file system implementation in `src/filesys` for you to use for the file-based system calls. While you *can* edit any file in PintOS for this assignment, we suggest that you *do not* modify the code in `src/filesys`.

Your system call implementation will need to be defensive with user process provided inputs. It should not assume that a user provided the right number of arguments, that the memory addresses they provide are valid, etc. Your implementation should use your User Memory solution to make this work. If a system call is passed an invalid argument, your PintOS should either return an error value (for system calls that return values), or terminate the offending process. The provided tests check for each of these cases, so you'll know that your PintOS does the right thing if it passes all of the tests.

Your system call implementation will need to be synchronized so that any number of user processes can execute a system call at once. The current implementation in `src/filesys` is not currently thread safe, so you should treat all functions from that directory as a global critical section. Note[1] there are calls to the file system from within `process_execute()`.

In the 80x86 architecture, the `int` is the mechanism for invoking a system call; in PintOS, user programs invoke `int    $0x30`. PintOS's system call calling convention is to push all arguments onto the stack before invoking the interrupt, in exactly the same manner as the 80x86's function calling conventions. PintOS already provides userspace implementations in `lib/usr/syscall.c`, so you will not need to worry about implementing anything in userspace.

However, you will need to implement the OS code to support system calls. When the system call handler, `syscall_handler()` from `sys/userprog/syscall.c`, gets control, the system call number will be in the 32-bit word at the caller's stack pointer, the first argument will be in the 32-bit word at the next higher address, and so on. Your implementation can access the caller's stack pointer from within `syscall_handler()` as the `esp` member of `f`, the `struct intr_frame` that is passed to the function. The 80x86 convention for function return values is to place them in eax. Your `syscall_handler()` can provide such functionality by modifying the `eax` member of `f`.

You should implement the following system calls in your PintOS. Note that there are other system calls that the PintOS userspace supports; no need to implement them. The System call numbers for each system call are defined in `lib/syscall-nr.h`, your PintOS will probably need to reference them:

```
void halt (void)
```

Terminate PintOS by calling `shutdown_power_off()` from `"devices/shutdown.h"`.

```
void exit (int status)
```

Terminates the current user program. If the process's parent waits for the child(see below), then the PintOS should return the `status` as it was passed to this function. Conventionally, a status of 0 indicates success and nonzero values indicate errors.

[1]this one tricked your professor

`pid_t exec (const char *cmd_line)`

Runs the executable whose arguments are given in `cmd_line`. Returns the new process's program id (pid), or, -1 if there is an error loading the child's executable. Note: this design means that the `exec` cannot return to the parent process until it determines whether the child process successfully loaded its executable.

`int wait (pid_t pid)`

Waits for a child process pid and retrieves the child's exit status. If child process pointed to by `pid` is still alive, `wait` should block until the child terminates. After the child process terminates, `wait` should return the status that the child passed to exit. Note that a parent process can call `wait` on a child processes that has already terminated; your PintOS should still ensure that the parent retrieves the correct status. `wait(pid)` should return -1 in 3 cases:

1. `pid` does not refer to a direct child of the calling process, i.e., a process that was created by calling process due to a call to `exec`. In PintOS, 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. Similarly, orphaned processes (processes whose parents are terminated) are not assigned to a new parent if their parent process exits before they do.
2. The process that calls `wait` has already called `wait` on `pid`. That is, a process may wait for any given child at most once.
3. The child process referenced by `pid` was terminated by the kernel (e.g., due to an exception).

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()`.

`bool create (const char *file, unsigned initial\_size)`

Creates a new file, called `file`, that initially has `initial_size` bytes in size. The system call should return `true` if successful and `false` otherwise. Creating a new file does not open it: opening the new file is a separate operation requiring the process call the system call `open`.

`bool remove (const char *file)`

Deletes the file, `file`. Returns `true` if successful and `false` otherwise. A file may be removed regardless of whether it is open or closed, and removing an open file does not close it.

`int open (const char *file)`

Opens the file, `file`. Returns a non-negative integer handle, a *file descriptor* (fd), or -1 if the file could not be opened. In PintOS, the 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 (there is no `STDERR_FILENO` in PintOS). The open system call should 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.

`int filesize (int fd)`

Returns the size, in bytes, of the file open as fd.

`int read (int fd, void *buffer, unsigned size)`

Reads `size` bytes from the file descriptor, `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). An input in which `fd` is 0 should read from the keyboard using `input_getc()`.

`int write (int fd, const void *buffer, unsigned size)`

Writes `size` bytes from `buffer` to file descriptor, `fd`. Should return the number of bytes actually written, which may be less than size if some bytes could not be written. While writing past end-of-file would normally extend the file, the basic file system does not implement file growth. So, your implementation only needs 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. A call in which `fd` is 1 should write to the console. You should try to write all of `buffer` to the console in a single call; otherwise lines of text from different processes can easily become interleaved.

`void seek (int fd, unsigned position)`

Changes the next byte to be read or written in file descriptor `fd` to be `position`, expressed in bytes from the beginning of the file. Your implementation should not treat a seek past the current end of a file as not an error. Instead, a later call to `read` or `write` on `fd` should return 0, indicating end of file.

`unsigned tell (int fd)`

Returns the position of the next byte to be read or written in the file descriptor, `fd`, calculated from the beginning of the file.

`void close (int fd)`

Closes file descriptor, `fd`. When a process terminates, whether from `exit` or through abnormally, your PintOS should implicitly close all of the process's open file descriptors.

PROCESS TERMINATION MESSAGE

Your PintOS should print a message indicating each time a user process has terminated. In particular, it should print the name and exit code, formatted as though the following were executed

```
printf("%s: exit(%d)\n", name, status);
```

where name is the name of the executable as passed to `process_execute()`, without any command-line arguments, and `status` is the status code that the process passed to `exit`, or -1 if the process terminated abnormally. PintOS should only produce the message for user processes, not for any kernel threads that execute, nor in the case that a user process invokes the `halt` system call.

Aside from this message, your PintOS should not print any other messages that the default PintOS does not already print. Extra messages will confuse the grading scripts, so be sure to remove any before submitting your assignment.

USER MEMORY

Many system calls ask that the operating system reference a user process's memory (e.g., `read`). A buggy or malicious process may provide an invalid memory address; your PintOS should prevent this from causing problems.

A Primer on PintOS Virtual Memory

To elaborate more, let's start by describing virtual memory in PintOS.

PintOS's virtual memory is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address 0 up to `PHYS_BASE`, defined as `0xc0000000` (3 GB) in `threads/vaddr.h`. Each process has its own virtual memory (or, address space). Each context switch exchanges virtual address spaces by changing the processor's `page directory base register` (see `page_activate()` in `userprog/pagedir.c`). Note that, if the current kernel thread executing on PintOS has an associated user process, then PintOS will be able to access that process's virtual memory.

Kernel virtual memory occupies the rest of the virtual address space, from `PHYS_BASE` up to 4 GB. Kernel virtual memory is global and always mapped the same way, regardless of what user process or kernel thread is running. PintOS maps kernel virtual memory one-to-one with physical memory, starting at `PHYS_BASE`. So, physical memory address 0x123 maps to kernel virtual address `PHYS_BASE + 0x123`

The hardware ensures that a process running in userspace can only access its memory—an attempt to access any memory outside of its address space (including accesses to kernel virtual memory) causes a page fault, handled by `page_fault()`in `userprog/exception.c`. In PintOS, the kernel cannot directly access physical memory—a kernel thread must access memory either in kernel virtual memory or through its associated user process's virtual memory. If a kernel thread attempts to access memory at an unmapped location, the hardware will trigger a page fault.

Safe User Memory Access

OK, with that background, now your task. You will need to provide mechanisms to read and write data in user virtual address spaces. Since all system call inputs come from user memory, even the system call number, you will need to use these facilities quite a bit!

There are basically two ways to do this correctly. First, you can verify that user-provided pointers are correct before dereferencing them. This way is a bit easier to get working correctly. If you choose this route, look at the functions in `userprog/pagedir.c` and in `threads/vaddr.h`.

Second, you can first check if user-provided pointers are in userspace, by checking that they are less than `PHYS_BASE`, and dereference them. An invalid user pointer will cause a page fault; you can modify the code in `page_fault()` to handle the fault correctly. This approach is a bit faster, typically, and thus is typically how modern kernels work. But, its a lot more complex. Section 3.1.5 in the pintos website has some helper code for you to use to implement this second approach. But, be warned: there are tricky bits left to implement.

RUBRIC

We will use the following rubric for this assignment:

Category	Percentage
Testing	60%
Design	40%

TESTING. We will use the provided tests for testing you assignment. You should hae 77 tests. If you see 80, then you should update your repository from the course upstream (by clicking "update fork" on gitUCSC). This will account for 60% of your grade. You can run these tests by executing the following from the `src/userprog/build` directory (after you have run `make` from the `src/userprog` directory):

```
make check
```

To see the tests as they will be weighted for the final score, execute:

```
make grade
```

DESIGN. We will evaluate your design document based upon the following criteria:

1. **Sufficient:** (30%) Does your design document describe the system with sufficient detail as to be re-creatable by an engineer?
2. **Accurate:** (30%) Does your design document accurately describe the design that you implemented?
3. **Correctness:** (30%) Would your proposed design, assuming it were implemented correctly, satisfy the requirements of the assignment?
4. **Simplicity:** (10%) Is your design simple, rather than overly complex?

HINTS

The pintos website provides a number of tips and hints for this assignment. The FAQ is pretty good—you do not need to "deny writes to executables", so no need to bother with those tips.

GETTING STARTED. This assignment is difficult because there are many moving parts. No one task is *too* difficult, but you have to get a little bit of *everything* working before you can test *anything*. Our suggestion is start your implementation by getting a small working prototype of each of the components until you can run a simple program. Only then should you start building up to implement full working components. For this project, here's what that entails:

1. Rather than getting argument parsing working off the bat, simply get the system to work for argument free programs. You can do this by changing `*esp = PHYS_BASE;` to be `*esp = PHYS_BASE - 12;` in the function `setup_stack()` in `userprog/process.c`. This will actually give you the wrong program name (a value of `null`) and so you won't pass any tests.
2. Rather than implementing the safe user memory, start by assuming non-malicious users and dereference user pointers directly.
3. Rather than implementing `process_wait`, simply replace it with an infinite loop. PintOS calls this function on startup, so if you don't replace this, the PintOS will never terminate.
4. Implement enough of the system call handler to get system call numbers from the stack. Implement `write` when it is called on `STDOUT_FILENO`.

With this in place, you should be able to start a program and see it print something to the console. You won't pass any tests, though. The fastest route to start passing tests is to next implement argument parsing and the process termination message.

CALLING CONVENTIONS. We highly recommend looking at Section 3.5, especially the example in Section 3.5.1, from this webpage to understand calling conventions. This will be helpful for both argument parsing and for handling system call inputs

RUNNING PROGRAMS. Running a user program requires having a file system image setup correctly. PintOS provides a number of helpers for this task; see Section 3.1.2 from the pintos webpage . Note that the testing scripts do all of this for you. You can copy and paste the command that the testing scripts executes on each test rather than setting up the file system manually each time.