Operating Systems Laboratory (CS39002)

Spring Semester 2019-2020

Assignment 6: Modifying PintOS to run user programs with arguments and system calls.

Assignment given on: March 17, 2020

Assignment deadline: March 31, 2020, 1:00 PM

In this assignment we will improve pintos further. Specifically, so far in pintOS all the code is written in the kernel space. That is a severe problem for an OS. Imagine if you need to always run your browser, media player or literally any other program as part of the kernel and access to all the privileged part of your OS and the hardware without any check. It is indeed problematic. To that end, in this assignment we will implement basic functioning of user programs. We will implement argument passing to these user programs and implement a few system calls. 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. You can build project 6 on top of your project 5 submission or you can start fresh.

You should read the primary document for this part of the project from the following link: https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html. However, we will ask you to implement only part of what is mentioned in this link (e.g., for example you don't need to implement all system calls). Below we are providing a background information which is basically summary and exerpt of the design document.

1. Background

Your pintOS implementation should allow more than one process to run at a time. Each process has one thread (multithreaded processes are not supported). User programs are generally written under the abstraction that they have access to the resources of 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 abstraction.

Please read the whole of section 3.1. of pintOS doc **very** carefully from the link: https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html. You will need this knowledge to even start this assignment. Specifically, this section will tell you about

- The key system files which will support your project on running user programs
- How to use the rudimentary file system provided in pintOS while running those user programs
- How user programs work in pintOS
- The virtual memory layout leveraged by user programs in pintOS and
- How to access the user memory

2. Enable minimal running of user programs

Even before implementing running of user programs with syscalls and arguments first you need to make sure you can minimally run the user programs (which does not use any arguments and even when syscalls are not doing what they suppose to). So first set the following up:

1. Every user program will page fault immediately until argument passing is implemented. For the setup phase, you may simply wish to change

   `*esp = PHYS_BASE;`

   to

   `*esp = PHYS_BASE - 12;`

   in setup_stack() function. That will work for any test program that doesn't examine its arguments, although its name will be printed as (null).

   Until you implement argument passing, you can only run programs without passing command-line arguments. Attempting to pass arguments to a program will include those arguments in the name of the program, which is likely to fail.

2. All system calls need to read user memory. Few system calls need to write to user memory. Use knowledge from the section marked as "Accessing User Memory" (from the documentation mentioned in Section 1) to make sure you understand you kernel can read user memory.

3. In the system call infrastructure implement enough code to read the system call number from the user stack and dispatch to a handler based on it.

4. 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). So implement the exit() system call to minimally run your user programs.

5. You also need to implement the write() system call for writing to fd 1, i.e., the system console, as part of set up. All of our test programs write to the console (the user process version of printf() is implemented this way), so they will all malfunction until write() is available.

6. For now, change process_wait() to an infinite loop (one that waits forever). The current implementation in pintOS returns immediately, so pintOS will power off before any processes actually get to run. You will eventually need to provide a correct implementation.

After the above are implemented, user programs should work minimally. At the very least, they can write to the console and exit correctly. Once you reach this point, you can actually start to run limited user programs. However, your OS is still limited in terms of the functionality it provides to run user programs.

3. Task 1: Argument passing [30 marks]

Currently, process_execute() does not support passing arguments to new processes. Implement this functionality, by extending process_execute() so that instead of simply

taking a program file name as its argument, it divides it into words at spaces. The first word is the program name, the second word is the first argument, and so on. That is, process_execute("grep foo bar") should run grep passing two arguments foo and bar.

Within a command line, multiple spaces are equivalent to a single space, so that process_execute("grep foo bar") is equivalent to our original example. You can impose a reasonable limit on the length of the command line arguments. For example, you could limit the arguments to those that will fit in a single page (4 kB). (Do not base your limit on the maximum 128-byte command-line arguments that the pintOS utility can pass to the kernel.)

You can parse argument strings any way you like. If you're lost, look at strtok_r(), prototyped in lib/string.h and implemented with thorough comments in lib/string.c. You can find more about it by looking at the man page (run man strtok_r at the prompt).

For information on exactly how you need to set up the stack see the link: https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html#SEC51 .

3. Task 2: Implement System calls [70 marks]

Implement the system call handler in userprog/syscall.c. The skeleton implementation "handles" system calls by terminating the process. It will need to retrieve the system call number, then any system call arguments, and carry out appropriate actions.

Implement the following system calls. The prototypes listed are those seen by a user program that includes lib/user/syscall.h. (This header, and all others in lib/user, are for use by user programs only.) System call numbers for each system call are defined in lib/syscall-nr.h:

.

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.

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 **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.

The file defines other syscalls. Ignore them for now. You will implement some of them in later projects, so be sure to design your system with extensibility in mind.

To implement syscalls, you need to provide ways to read and write data in user virtual address space. You need this ability before you can even obtain the system call number, because the system call number is on the user's stack in the user's virtual address space. This can be a bit tricky: what if the user provides an invalid pointer, a pointer into kernel memory, or a block partially in one of those regions? You should handle these cases by terminating the user process. We recommend writing and testing this code before implementing any other system call functionality.

You must synchronize system calls so that any number of user processes can make them at once. In particular, it is not safe to call into the file system code provided in the filesys directory from multiple threads at once. Your system call implementation must treat the file system code as a critical section. Don't forget that process_execute() also accesses files. For now, we recommend against modifying code in the filesys directory.

PintOS provides you a user-level function for each system call in lib/user/syscall.c. These provide a way for user processes to invoke each system call from a C program. Each uses a little inline assembly code to invoke the system call and (if appropriate) returns the system call's return value.

Testing your implementation: You can test if your implementation is actually performing as intended by compiling and running a few user programs. Some user programs are in src/examples directory. We suggest compiling and running "cat.c, cp.c, echo.c, hex-dump.c, rm.c" files. You can run them together by modifying the "PROGS =" line in src/examples/Makefile.

Submission Guideline:

You need to upload a zip containing the files you changed along with a design document in Moodle. There should be one submission from each group. Name your zip file as "**Ass6_<groupno>.zip**". The zip file should contain:

- The files that you changed.
- A design document. You can find the template for design document here: http://cse.iitkgp.ac.in/~mainack/OS/assignments/06/userprog.tmpl.txt Fill it up according to your implementation and include it in your zip file.

If you are outside campus and cannot access moodle please follow the instructions that was included in the mail sent regarding submission of OS assignments.

Grading scheme:

The total marks for this part of the assignment (100 marks) is divided as follows.

Demo the implementation to your assigned TAs and show that your code works as intended	**40%**
During demo you need to demonstrate that you have indeed implemented the algorithm promised in the design document	**60%**