Operating Systems Laboratory (CS39002)

Spring Semester 2020-2021

Assignment 6: Extending PintOS to run user programs with arguments (by changing program stack structure) and implementing (mostly file system-related) system calls.

Assignment given on:	March 14, 2020
Assignment deadline for part 1 (section 3):	March 25, 2020, 1:00 PM
Assignment deadline for part 2 (section 4):	April 8, 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.

We divided this project into two parts, so that you can demonstrate intermediate results for this project. In the part 1 you just need to make sure that user programs can run with arguments and implement two syscalls.

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 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 excerpt of the design document.

1. Motivation

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

- How to access the user memory
- How to implement syscalls

2. Background (without following these steps you cannot do the assignment)

We understand that file-system is not yet covered in class. However, to enable running of user programs in pintos you need file system support (say to store and load user written programs using disk). So, we are providing you the exact commands to run for creating a basic file system below. Running these commands should be sufficient to create a filesystem for this project.

2.1. Setting up a basic file system in pintos

You will need to interface to the file system code for this project, because user programs are loaded from the file system and many of the system calls you will implement deal with the file system. There is a simple but complete file system in the filesys directory. You may want to look over the filesys.h and file.h interfaces to understand how to use the file system, and especially its many limitations.

Current file system has the following limitations, but it should be enough for this project. So, you only need knowledge about memory and NOT file system for this first part of the project:

- 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, that is, 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. You need to restart pintos and recreate the file system.

One important feature is included:

- Unix-like semantics for filesys_remove() are implemented in this file system. That is, if a file is open when it is removed, its blocks are not deallocated and it may still be accessed by any threads that have it open, until the last one closes it.

You need to create a simulated disk with a file system partition for this project. The pintos-mkdisk program enables you do so.

- 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.

Incidentally, these commands work by passing special commands extract and append on the kernel's command line and copying to and from a special simulated "scratch" partition. If you're very curious, you can look at the pintos script as well as filesys/fsutil.c to learn the implementation details (not necessary for this project).

Summary of commands-to-run for creating filesystem: 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. (however, argument passing won't work until you implemented it in first part of this project.) 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'
```

These three final steps can actually be combined into a single command, so you can just run:

```
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.

2.2. User program memory stack structure

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 compile your own

programs as well. Some of the example programs will only work once projects 3 or 4 have been implemented.

Pintos can load ELF executables with the loader provided for you in userprog/process.c. ELF is a file format used by Linux, Solaris, and many other operating systems for object files, shared libraries, and executables. You can actually use any compiler and linker that output 80x86 ELF executables to produce programs for Pintos.

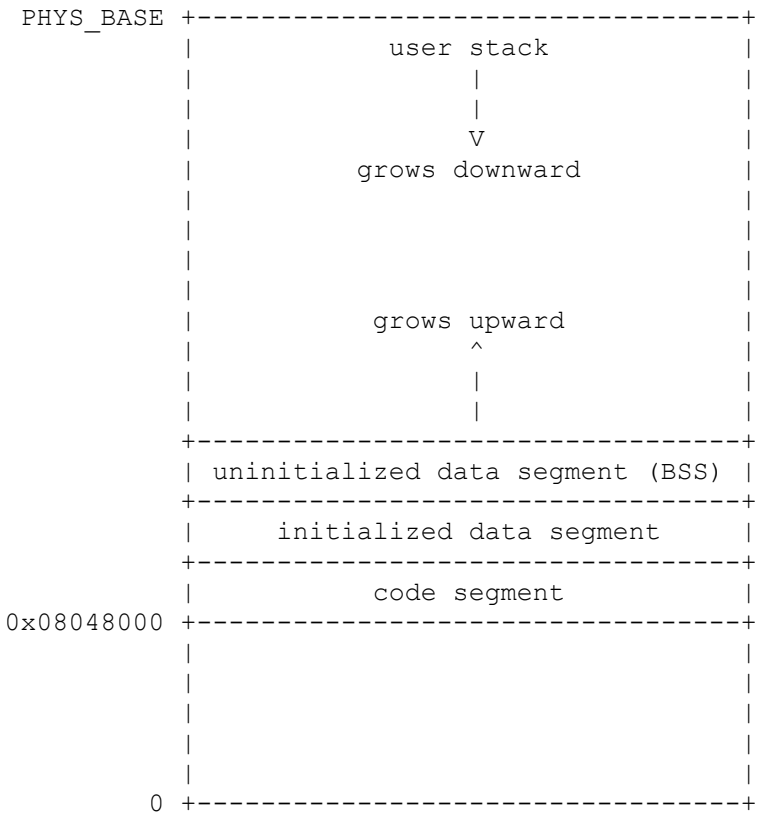
You should realize immediately that, until you copy a test program to the simulated file system, Pintos will be unable to do useful work. You won't be able to do interesting things until you copy a variety of programs to the file system. You might want to create a clean reference file system disk and copy that over whenever you trash your filesys.dsk beyond a useful state, which may happen occasionally while debugging.

2.3. User program's memory stack structure

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. 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:



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.

2.4. Enable minimal running of user programs (for starting your assignment)

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 after creating file system (see section 2.1 for the commands to run), first set the following up:

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

   ```
   *esp = PHYS_BASE;
   ```

   to

   ```
   *esp = PHYS_BASE - 12;
   ```

   within 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 + 2 syscalls [30 + 2x10 = 50 marks]

Deadline 25th March 2021

3.1. Argument passing

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). Once you tokenize the arguments, you need to setup your stack for storing and retrieving these arguments.

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.

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:

```
bffffc0                                          00 00 00 00 |            ....|
bffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |................|
bffffe0  f8 ff ff bf fc ff ff bf-00 00 00 00 2f 62 69 |............/bi|
bfffffe0  6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|
```

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

3.2. Implement two (2) system calls

Now that your program can take user arguments as inputs, let's give them more functionalities. In a modern machine that is done via enabling user programs to avail OS service. A user program can avail service from system via system calls or syscalls.

Implement the system call handler in userprog/syscall.c. The skeleton implementation "handles" system calls by terminating the process using the code below:

```
static void
syscall_handler (struct intr_frame *f UNUSED)
{
  printf ("system call!\n");
  thread_exit ();
}
```

Of course, you need to extend it and handle different system calls differently. In pintos, each system will create an interrupt with the interrupt number (check out `intr_frame`). Within the syscall handler pintos should retrieve the system call number (using the input argument), 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: 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.

To implement system calls (i.e., 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_part1_<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 (the argument passing and system call part) and include it in your zip file.

Late submission policy for part 1

- You need to show argument passing for the first part of this assignment.
- However, if you cannot finish demo-ing both the system calls, you can have another chance to demo them with the second part with a 30% penalty for the incomplete system call part of the first assignment.

4. Task 2: 7 syscalls [7 x 10 = 70 marks]

Deadline 8th April 2021

Now implement the following six more system calls (see **3.2.** above) related to file system. You should complete the system calls in first part first before attempting this one.

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.

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 "**Assignment_6_part2_<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 (the system call part) and include it in your zip file.