Introduction

Work must be submitted in groups of two or three, with only a single hand-in containing both solo and collaborative work should be submitted with a group cover sheet showing both student IDs.

The second part of the assignment is to develop system calls for the educational Pinto OS. Pinto OS was developed at Stanford University to enable students to experience many of the features of a real OS, without some of the complexities of an operating system such as Linux or MS Windows.

You must submit:

- Architecture design document, detailing your design and how you implemented it.
- Documented source code. All changes and additions to the source code must be documented.

Part 1 - Argument Passing (25%)

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 (runman **strtok_r** at the prompt).

There will be an in lab session, where the details of Pintos, will be covered, including hints on this part of the assignment. This will take place on the week of 27/Nov/2017. The slides will be posted to BB, but it is strongly advised that you addend those sessions.

Part 2 System Calls (40%)

Implement the system call handler in userprog/syscall.c. The skeleton implementation we provide "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.

The more system calls you implement the more marks will be opened up. Note, that some are much harder than others and so you might what to choose and order of implementation! Marks will be assigned in alignment to how hard they are to implement.

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 halt (void)** Terminates Pintos by calling **shutdown_power_off()** (declared in **threads/init.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.
- System Call: **pid_t exec (const char *cmd_line)** Runs the executable whose name is given in cmd_line, passing any given arguments, and returns the new process's program id (pid). Must return pid -1, which otherwise should not be a valid pid, if the program cannot load or run for any reason. Thus, the parent process cannot return from the **exec** until it knows whether the child process successfully loaded its executable. You must use appropriate synchronization to ensure this.
- System Call: **int wait (pid_t pid)** Waits for a child process **pid** and retrieves the child's exit status.

  If **pid** is still alive, waits until it terminates. Then, returns the status that pid passed to exit. If **pid** did not call **exit()**, but was terminated by the kernel (e.g. killed due to an exception), **wait(pid)** must return -1. It is perfectly legal for a parent process to wait for child processes that have already terminated by the time the parent calls **wait**, but the kernel must still allow the parent to retrieve its child's exit status, or learn that the child was terminated by the kernel.

  wait must fail and return -1 immediately if any of the following conditions is true:

  - pid does not refer to a direct child of the calling process. **pid** is a direct child of the calling process if and only if the calling process received **pid** as a return value from a successful call to exec. Note that children are not inherited: if **A** spawns child **B** and **B** spawns child process **C**, then **A** cannot wait for **C**, even if **B** is dead. A call to **wait(C)** by process **A** must fail. Similarly, orphaned processes are not assigned to a new parent if their parent process exits before they do.
  - The process that calls **wait** has already called **wait** on **pid**. That is, a process may wait for any given child at most once.

  Processes may spawn any number of children, wait for them in any order, and may even exit without having waited for some or all of their children. Your design should consider all the ways in which waits can occur. All of a process's resources, including its struct thread, must be freed whether its parent ever waits for it or not, and regardless of whether the child exits before or after its parent.

You must ensure that Pintos does not terminate until the initial process exits. The supplied Pintos code tries to do this by calling **process_wait()** (in **userprog/process.c**) from **main()** (in **threads/init.c**). We suggest that you implement **process_wait()** according to the comment at the top of the function and then implement the **wait** system call in terms of **process_wait()**.

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

- 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.
- 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 openreturns 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 our version of Pintos files have a fixed length, 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.

The file defines other syscalls. You are can safetly ignore them and no additional marks will be given for implementing them, although much kudos will be given!

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. It is not recommended to modify the code in the **filesys** directory.

The source base provides 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.

When you're done with this part, and forevermore, Pintos should be bulletproof. Nothing that a user program can do should ever cause the OS to crash, panic, fail an assertion, or otherwise malfunction.

If a system call is passed an invalid argument, acceptable options include returning an error value (for those calls that return a value), returning an undefined value, or terminating the process.

There will be an in lab session, where the details of system calls, within Pintos, will be covered. This will take place on the week of 11/12/2017. The slides will be posted to BB, but it is strongly advised that you addend those sessions.