# Assignment 4: Implementing a Linux *shell*

Date assigned: October 5, 2022
Date due: November 2, 2022

# Unix Shell

In this project, you will build a simple Unix shell to replace `sh` provided by the Linux kernel. The shell is the *command-line* interface, and thus central to any Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.

There are three specific objectives for this assignment:

1. To reinforce/expand your Unix programming skills
2. To create, destroy and manage Unix processes, and
3. To understand the roll and functioning of shell user interfaces

# Overview

In this assignment, you will implement a *command line interpreter (CLI)* or, as it is more commonly known, a *shell*. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shell you implement will be like, but simpler than, the one used every day in Unix systems. If you don't know what shell you are running, it's probably `bash`. One thing you should do on your own time is to learn more about the Linux (Ubuntu) shell by reading the man pages or other online materials.

# Program Specifications

## Basic Shell: `smash`

Your basic shell, referred to as "`smash`" (short for the Super Milwaukee Shell, naturally), is basically an interactive loop: that repeatedly prints a prompt `smash>` (including the space after the greater-than sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This loop is repeated until the user types `exit` or `ctrl-z`

The name of your shell source code should be `smash.c`.

The shell can be invoked with either no arguments or a single argument; anything else is an error. Here is the no-argument
version:

```
prompt>
./smash
smash>
```

At this point, `smash` is running and ready to accept commands. Type away!

The mode above is called *interactive* mode and allows the user to type commands directly. The shell also supports a *batch mode*, which will read input from a *batch file* and executes commands found there. Here is how you run the shell with a batch file named `batch.txt`:

```
prompt> ./smash batch.txt
```

Notice that in interactive mode, a prompt is printed (`smash>` ). In batch mode, no prompt should be printed.

You should structure your shell such that it creates a process for each new command (the exception is *built-in commands*, discussed below). Your basic shell should be able to parse a command and run the program corresponding to the command. For example, if the user types `ls -la /tmp` , your shell should run the program `/bin/ls` with the given arguments `-la` and `/ tmp`. (How does the shell know to run`/bin/ls`? It follows something called the shell **path**; more on this below).

# Structure

## Basic Shell

The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the shell's built-in command "`exit`", at which point it exits. That's it!

For reading lines of input, you use `getline()` library method. This allows you to obtain arbitrarily long input lines with ease. Generally, the shell will be run in *interactive mode*, where the user types a *command-line* (one at a time) and the shell acts on it. However, your shell will also support *batch mode*, in which the shell is given an input file of *command-lines*; in this case, the shell should not read user input (from `stdin`) but rather from this batch file to acquire the commands to execute.

In either mode, if you hit the end-of-file marker (EOF) in the batch file you should call `exit(0)`and exit gracefully.

To parse each command-line into its constituent pieces, consider using `strsep()`. Read the *man* page (carefully) for more details.

To execute commands, use `fork()`, `exec()`, and `wait()/waitpid(.)`. See those man pages for definition of these functions and read the relevant book chapter (http://www.ostep.org/cpu-api.pdf) for a brief overview.

You will note that there are a variety of commands in the `exec` family. For this project use `execv`. You should **not** use the `system()` library function call to run a command. Remember that if `execv()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part

is getting the arguments correctly specified.

**Note**: Generally `argv[0]` for programs is the program's name instead of a pathname.

# Paths

In our example above, the user typed `ls` but the shell knew to execute the program `/bin/ls`. How does your shell know this?

It turns out that the user must specify a **path** variable to describe the set of directories to search for executables; the set of directories that comprise the path are sometimes called the *search path* of the shell. The path variable contains the list of all directories to search, in order, when the user types a command.

**Important:** Note that the shell itself does not *implement* `ls` or other commands (except built-ins). All it does is find those executables in one of the directories specified by `path` and create a new process to run them.

To check if a particular file exists in a directory and is executable, consider the `access()` system call. For example, when the user types "`ls`", and path is set to include both `/usr/bin` and `/bin` (assuming empty path list at first, `/bin` is added, then `/usr/bin` is added), try `access("/usr/bin/ls", X_OK)`. If that fails, try `/bin/ls`. If that fails too, it is an error. Your initial shell path should contain one directory: `/bin`

Note: Most shells allow you to specify a binary specifically without using a search path, using either **absolute paths** or **relative paths**. For example, a user could type the **absolute path** `/bin/ls` and execute the `ls` binary without a search path being needed. A user could also specify a **relative path** which starts with the current working directory and specifies the executable directly, e.g., `./main`. In this project, you **do not** have to worry about these features.

# Built-in Commands

Whenever your shell accepts a command, it should check whether the command is a **built-in command** or not. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command. For example, to implement the `exit` built-in command, you simply call `exit(0);` in your smash source code, which then will exit the shell.

In this project, you should implement `exit`, `cd`, and `path` as built-in commands.

- `exit`: When the user types `exit`, your shell should simply call the `exit` system call with 0 as a parameter. It is an error to pass any arguments to `exit`.
- `cd`: `cd` should always take one argument (0 or >1 args should produce an error). To change directories, use the `chdir()` system call with the argument supplied by the user; if `chdir` fails, that is also an error.
- `path`: The `path` command takes 1 or more arguments, with each argument separated by whitespace from the others. Three options are supported: `add`, `remove`, and `clear`. **Clarification**: Invalid arguments should generate an error.

  `add` accepts 1 path. Your shell should append it to the *beginning* of the path list. For example,

`path add /usr/bin` results in the path list containing `/usr/bin` and `/bin` (notice the order here). Your shell should *not* report an error if an invalid path is added. It should kindly accept it.
`remove` accepts 1 path. It searches through the current path list and removes the corresponding one. If the path cannot be found, this is an error.
`clear` takes no additional argument. It simply removes everything from the path list. If the user sets path to be empty, then the shell should not be able to run any programs (except built-in commands).

# Redirection

Many times, a shell user prefers to send the output of a program to a file rather than to the screen. Usually, a shell provides this nice feature with the `>` character. Formally this is named as redirection of standard output. To make your shell users happy, your shell should also include this feature, but with a slight twist (explained below).

For example, if a user types "`ls -la /tmp > output`", nothing should be printed on the screen. Instead, the standard output of the `ls` program should be rerouted to the file `output`. In addition, the standard error output of the program should be rerouted to the file `output` (the twist is that this is a little different than standard redirection). However, if the program cannot be found (i.e., mistyped `pwd` as `pdd`), an error should be reported, but not to be redirected to `output`.

If the `output` file exists before you run your program, you should simply overwrite it (after truncating it).

The exact format of redirection is a command (and possibly some arguments) followed by the redirection symbol followed by a filename. Multiple redirection operators or multiple files to the right of the redirection sign are errors. Redirection without a command is also not allowed - an error should be printed out, instead of being redirected.

Note:

- Don't worry about redirection for built-in commands (e.g., we will **not** test what happens when you type `path /bin > file`).
- Don't worry about the order of `stdout` and `stderr`. In other words, if a process writes to both, the output could be jumbled up. (This is okay!)

# Parallel Commands

```
smash> cmd1 & cmd2 args1 args2 & cmd3 args1
```

Your shell will also allow the user to launch parallel commands. This is accomplished with the ampersand operator as follows:

In this case, instead of running `cmd1` and then waiting for it to finish, your shell should run `cmd1`, `cmd2`, and `cmd3` (each with whatever arguments the user has passed to it) in parallel, *before* waiting for any of them to complete.

Then, after starting all such processes, you must make sure to use `wait()` (or `waitpid`) to wait for them to complete. After all processes are done, return control to the user as usual (or, if in batch mode, move on to the next line).

Note:

- Don't worry about parallel built-in commands (e.g., we will **not** test `cd foo & ls -al`). Redirection should
- be supported (e.g., `cmd1 > output & cmd 2`).
- Empty commands are allowed (i.e., `cmd &` , `& cmd` ).

## Multiple Commands

What if a shell user would like to type in multiple commands in a single line? Sometimes, they might turn in a long list of commands, prepare some popcorns and, wait until all of them to finish. This is supported by semicolons:

```
smash> cmd1 & cmd2 args1 args2 ; cmd3 args1
```

Here, your shell runs `cmd 1` and `cmd 2` in parallel, like specified above, waits until they complete, and executes `cmd 3`
afterwards.

Note:

- Your shell should support multiple built-in commands, such as `ls ; cd foo ; ls`
- Redirection should be supported (e.g., `cmd1 > output ; cmd 2`). Empty commands are allowed (i.e., `cmd ;`
- `, ; cmd &` ).

## Program Errors

**The one and only error message.** You should print this one and only error message whenever you encounter an error of any type:

```
char error_message[30] = "An error has
occurred\n"; write(STDERR_FILENO,
error_message, strlen(error_message));
```

The error message should be printed to stderr (standard error), as shown above.

After most errors, your shell simply *continues processing* after printing the one and only error message. However, if the shell is invoked with more than one file, or if the shell is passed a file that doesn't exist, it should exit by calling `exit(1)`.

There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there are any program-related errors (e.g., invalid arguments to `ls` when you run it, for example), the shell does not have to worry about that (rather, the program will print its own error messages and exit).

For parallel and multiple commands, syntax errors (e.g., `ls; > output` ) or invalid programs names (e.g., a mistyped `ls` , like `lss` ) should prevent the entire line from executing.

# Miscellaneous Hints

Remember to get the **basic functionality** of your shell working before worrying about error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as `ls .`

Next, add built-in commands. Then, try working on redirection. Finally, think about parallel and multiple commands. Each of these requires a little more effort on parsing, but each should not be too hard to implement. It is recommended that you separate the process of parsing and execution - parse first, look for syntax errors (if any), and then finally execute the commands.

```
smash> ls& pwd>output; cd /usr
```

At some point, you should make sure your code is robust to white space of various kinds, including spaces ( ) and tabs ( `\t` ). In general, the user should be able to put variable amounts of white space before and after commands, arguments, and various operators; however, the operators (redirection, parallel commands, and multiple commands) do not require whitespace. For example, your shell should accept commands like:

Check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. It's also just good programming sense.

Beat up your own code! You are the best (and in this case, the only) tester of this code. Throw lots of different inputs at it and make sure the shell behaves well. Good code comes through testing; you must run many different tests to make sure things work as desired. Don't be gentle – other users certainly won't be.

Finally, keep versions of your code. More advanced programmers will use a source control system such as git. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.

# Submitting Your Implementation

It is possible to implement the shell in a single `.c` file, so your Canvas submission should contain three files: the compliable c source, a README to explain your solution and how to use it, and one or more screen shots showing it in use on Linux.