Unix Shell
Programming Assignment

In this project, you will build a simple Unix shell (pseudo-shell). The shell is the primary user interface to modern Linux/UNIX operating systems. Essentially, the shell is a user-space process that allows users to interact with modern operating system. Developing a *pseudo-shell* that runs in user-space is the focus of this project.

There are three specific objectives to this assignment:

- To further familiarize yourself with the Linux programming environment.
- To gain non-trivial experience in **process management**, **IPC** and **process redirection**.
- To gain exposure in developing **service-oriented** apps.

Overview

In this assignment, you will implement a *command line interpreter (CLI)* or, as it is more commonly known, a *pseudo-shell*. The shell should operate in the following manner:

- The shell prints out a prompt and waits for the user to enter a command
- when the user enters a command (in response to the shell prompt)
    - the shell creates a child process that executes the command entered
    - waits until the child process terminates
    - and then prompts for more user input.

The shells you implement will be similar to, but simpler than, the one provided in UNIX. If you don't know what shell you are running, it's probably **zsh** or **bash**. One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials.

Program Specifications

Basic Pseudo-Shell: gush

Your basic shell, called gush (short for Georgetown University Shell, naturally), is basically an interactive loop: gush repeatedly prints a prompt gush> (note 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 is repeated until the user types exit. The name of your final executable should be gush.

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

```
prompt> ./gush
gush>
```

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

The mode above is called *interactive* mode, and allows the user to enter commands directly into **gush**. gush also supports a *batch mode*, which reads input from a batch file and executes commands from therein. Here is how you run gush with a batch file named `batch.txt`:

```
prompt> ./gush batch.txt
```

One difference between batch and interactive modes is:

- In interactive mode, a prompt is printed (gush>).
- In batch mode, no prompt should be printed.

You should structure your shell such that it creates a process for each new command entered by the user. An exception to this rule is for *built-in commands*. Your gush must parse data entered at the prompt to correctly spawn and execute commands entered. 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's something called the shell **path**; more on this below).

Structure

Basic Shell (**gush**)

The shell is very simple (conceptually): it runs in a while loop, repeatedly waiting for input to indicate what command to execute. The shell, then executes that command. The loop continues indefinitely, until the user types the built-in command **exit**, at which point, the shell exits.

When reading lines of input, consider using **getline()**. getline() 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 (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 commands. In batch mode, the shell does not read user input (from **stdin**) but rather from the supplied file to get the commands to execute.

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

To parse the input line into constituent pieces, consider using strtok() (or, if doing nested tokenization, strtok_r()). Read the man page (carefully) for more details.

To manage processes you must use fork(), execve(), and wait()/waitpid(). See the man pages for these functions, and also read the relevant [chapter(s) in text] for a brief overview.

You will note that there are a variety of commands in the **exec** family; for this project, you must use **execve**. You should **not** use the system() library function call to run a command. Remember that if execve() 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.

Paths

In our example above, the user typed ls, which is located in the /bin directory (i.e., /bin/ls). How did the shell find the program ls?

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 (the exception are built-ins commands). All the shell does is find those executables in one of the directories specified by path and create a new process to execute 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 /bin and /usr/bin, try access("/bin/ls", X_OK). If that fails, try "/usr/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 (executable file) 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 using the shell search path. A user could also specify a **relative path** which starts with the current working directory and specifies the executable directly, e.g., ./main

Built-in Commands

Whenever your shell accepts a command, it should check whether the command is a **built-in command** or not. If the command is a **built-in**, then the shell will (itself) process the command. Built-in commands are implemented as functions in the shell. For example, to implement the exit built-in command, you simply call exit(0); in your gush source code, which then will exit the shell.

In this project, you should implement exit, cd, kill, history, pwd, 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 always takes **zero (0)** or **one argument** (>1 args should be signaled as 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 **0** or **more arguments**, with each argument separated by whitespace from the others. An example of typical usage might be:

  gush> path /bin /usr/bin

  which adds /bin and /usr/bin to the search path of the shell. If the user sets path to be empty, then the shell should only execute commands containing **absolute paths**, **relative paths** or **build-in commands**. The path command always overwrites the old path with the newly specified path.

- **pwd**: The pwd command takes zero (0) arguments and prints the full path name of the current working directory to **stdout**.

- **history**: The history command provides users with the ability recall and execute commands entered in the shell. Your shell will keep a **circular list** of the last **ten (10)** commands entered. If the number of commands exceeds ten, your shell will overwrite previous commands entered (in FIFO order)

  - typing **history** at the command prompt will list the contents of the shell's history list e.g. (assume user typed **ls –l**, **gcc –g test.c** and **pwd**)
    gush> history
    - 1 ls –l
    - 2 gcc –g test.c
    - 3 pwd
      note: history is not stored in the shell's history list

  - To execute a previously entered command type !n (n is a number in the history list) at the command prompt and the shell will re-execute the command in the shell's history list with that number (.e.g. assuming the above listing)
  - gush>!2 *# this will re-execute gcc-g test.c*

  - It is an error to try and re-execute a non-existing command

Redirection

A shell allows users to redirect input/output of a process. For example, redirection allows a process to send output that normally would be sent to the screen, be sent to a file instead. The following symbols are used to indicate redirection **<** , **>** .

- **>** is used to redirect standard out
- **<** is used to redirect standard in

For example, if a user types ls -la /tmp > output, the output from /bin/ls will be sent to the file **output**. Hence, standard out is redirected to the file output.

Redirecting standard out

- Creates the output file if it did not exist
- Truncates and overwrites the output file, if the file already existed.

The 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.

Pipes

Pipes are a form of Inter Process Communication. Pipes allow process to communicate through the use of file descriptor (stdin, stdout). Your gush will implement the pipe feature found in traditional shells. An example is below

gush> ls –l | nl

The above command use the pipe symbol "|" to create a one-direction communication channel where ls –l writes to the **pipe** and nl reads from the **pipe**. There may be up to 4 pipes on a command line.

Parallel Commands (background processes)

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

gush> cmd1 & cmd2 args1 args2 &

In this case, instead of running cmd1 and then waiting for it to finish, your shell will run cmd1 and cmd2 (each with whatever arguments the user has passed to it) in parallel. The shell returns immediately after spawning cmd1 and cmd2 (the shell does not wait on parallel commands to exit, before continuing).

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).

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 bad batch file, 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).

What to submit:

- Well designed source files
- Makefile
- Readme.txt (including list of features implemented and not implemented)

To receive A- or A requires:

- General implementation
- Processing commands with **two** or more **pipes**
- Redirecting **standard in**
- Processing multiple **background processes**

General Implementation

- Shell interactive mode
    - Managing processes fork()/execve()/wait()/waitpid()
- Built-in commands
- Redirecting standard out
- Processing a single pipe (this requires processing two commands)
    - For example, ls -l | wc -l
- Shell batch mode
- Error message and handling