Project 2a: The Unix Shell

Objectives

There are three objectives to this assignment:

- To familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.

Updates

Read these updates to keep up with any small fixes in the specification.

When a command is not found in the path, do not print **command not found,** but rather the only error message.

By default, the path should be set to **/bin** so that the user can type a few commands without first setting the path. Type `ls /bin` to see what commands are available.

You do not have to support multiple commands being executed concurrently or being specified on the same command line; each command line will contain at most one command to run.

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 shells you implement will be similar to, but simpler than, the one you run every day in Unix. You can find out which shell you are running by typing `"echo $SHELL"` at a prompt. You may then wish to look at the man pages for the shell you are running (probably `bash`) to learn more about all of the functionality that can be present. For this project, you do not need to implement too much functionality.

Program Specifications

Basic Shell: WhooSH

Your basic shell, called `whoosh` , is basically an interactive loop: it repeatedly prints a prompt `"whoosh> "` (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 **whoosh:**

```
prompt> ./whoosh
whoosh>
```

You should structure your shell such that it creates a new process for each new command (note that there are a few exceptions to this, which we discuss below). There are two advantages of creating a new process. First, it protects the main shell process from any errors that occur in the new command. Second, it allows for concurrency; that is, multiple commands can be started and allowed to execute simultaneously. **However, in this project, you do not have to build any support for running multiple commands at once.**

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 all the given arguments and print the output on the screen.

You might be wondering how the shell knows to run `/bin/ls` (which means the program binary `ls` is found in the directory `/bin`) when you type `ls` . The shells knows this thanks to a **path** variable that the user sets. The path variable contains the list of all directories to search, in order, when the user types a command. We'll learn more about how to deal with the path below.

Important: Note that the shell itself does not "implement" `ls` or really many other commands at all. All it does is find those executables in one of the directories specified by `path` and create a new process to run them. More on this below.

The maximum length of a line of input to the shell is 128 bytes.

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 C program.

So far, you have added your own `exit` built-in command. Most Unix shells have many others such as `cd` , `echo` , `pwd` , etc. In this project, you should implement **exit, cd, pwd,** and **path.**

The formats for **exit, cd,** and **pwd** are:

```
[optionalSpace]exit[optionalSpace]
[optionalSpace]pwd[optionalSpace]
[optionalSpace]cd[optionalSpace]
[optionalSpace]cd[oneOrMoreSpace]dir[optionalSpace]
```

When you run `cd` (without arguments), your shell should change the working directory to the path stored in the $HOME environment variable. Use the call `getenv("HOME")` in your source code to obtain this value.

You do not have to support tilde (~). Although in a typical Unix shell you could go to a user's directory by typing `cd ~username,` in this project you do not have to deal with tilde. You should treat it like a common character, i.e., you should just pass the whole word (e.g.

"~username") to chdir(), and chdir will return an error.

Basically, when a user types `pwd`, you simply call getcwd(), and show the result. When a user changes the current working directory (e.g. "cd somepath"), you simply call chdir(). Hence, if you run your shell, and then run pwd, it should look like this:

```
% cd
% pwd
/afs/cs.wisc.edu/u/m/j/username
% echo $PWD
/u/m/j/username
% ./whoosh
whoosh> pwd
/afs/cs.wisc.edu/u/m/j/username
```

The format of the **path** built-in command is:

```
[optionalSpace]path[oneOrMoreSpace]dir[optionalSpace] (and possibly
more directories, space separated)
```

A typical usage would be like this:

```
whoosh> path /bin /usr/bin
```

By doing this, your shell will know to look in `/bin` and `/usr/bin` when a user types a command, to see if it can find the proper binary to execute. If the user sets path to be empty, then the shell should not be able to run any programs (but built-in commands, such as path, should still work).

Redirection

Many times, a shell user prefers to send the output of his/her 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 `output.out` file. In addition, the standard error output of the file should be rerouted to the file `output.err` (the twist is that this is a little different than standard redirection).

If the `output.out` or `output.err` files already exists before you run your program, you should simple overwrite them (after truncating). If the output file is not specified (e.g. the user types `ls >`), you should print an error message and not run the program `ls` .

Here are some redirections that should **not** work:

```
ls > out1 out2
ls > out1 out2 out3
ls > out1 > out2
```

Note: don't worry about redirection for built-in commands (e.g., we will not test what happens when you type `path /bin > file`).

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). Also, do not add whitespaces or tabs or extra error messages.

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 is any program-related errors (e.g. invalid arguments to `ls` when you run it, for example), let the program prints its specific error messages in any manner it desires (e.g. could be stdout or stderr).

White Spaces

The `">"` operator will be separated by spaces. Valid input may include the following:

```
whoosh> ls
whoosh> ls > a
whoosh> ls > a
```

But not this (it is ok if this works, it just doesn't have to):

```
whoosh> ls>a
```

Defensive Programming and Error Messages

Defensive programming is required. Your program should check all parameters, error-codes, etc. before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your

program must respond to all input in a reasonable manner; by "reasonable", we mean print the error message (as specified in the next paragraph) and either continue processing or exit, depending upon the situation.

Since your code will be graded with automated testing, 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));
```

For this project, the error message should be printed to *stderr.* Also, do not attempt to add whitespaces or tabs or extra error messages.

You should consider the following situations as errors; in each case, your shell should print the error message to stderr and **exit** gracefully:

- An incorrect number of command line arguments to your shell program.

For the following situation, you should print the error message to stderr and **continue** processing:

- A command does not exist or cannot be executed.
- A very long command line (over 128 bytes).

Your shell should also be able to handle the following scenarios below, which are **not errors.**

- An empty command line.
- Multiple white spaces on a command line.

All of these requirements will be tested extensively.

Hints

Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly. To simplify things for you in this assignment, we will suggest a few library routines you may want to use to make your coding easier. You are free to use these routines if you want or to disregard our suggestions. To find information on these library routines, look at the manual pages.

Basic Shell

Parsing: For reading lines of input, once again check out **fgets().** To open a file and get a handle with type **FILE *** , look into **fopen().** Be sure to check the return code of these routines for errors! (If you see an error, the routine **perror()** is useful for displaying the problem. *But do not print the error message from perror() to the screen. You should only print the one and only error message that we have specified above*). You may find the **strtok()** routine useful for parsing the command line (i.e., for extracting the arguments within a command separated by whitespaces).

Executing Commands: Look into **fork, execv,** and **wait/waitpid.** See the man pages for these functions, and also read book chapter here.

You will note that there are a variety of commands in the `exec` family; for this project, you must use **execv.** You should **not** use the **system()** 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. The first argument specifies the program that should be executed, with the full path specified; this is straight-forward. The second argument, `char *argv[]` matches those that the program sees in its function prototype:

```
int main(int argc, char *argv[]);
```

Note that this argument is an array of strings, or an array of pointers to characters. For example, if you invoke a program with:

```
foo 205 535
```

and assuming that you find `foo` in directory `/bin` , then argv[0] = "/bin/foo", argv[1] = "205" and argv[2] = "535".

Important: the list of arguments must be terminated with a NULL pointer; in our example, this means argv[3] = NULL. We strongly recommend that you carefully check that you are constructing this array correctly!

Built-in Commands

For the `exit` built-in command, you should simply call `exit()`. The corresponding process will exit, and the parent (i.e. your shell) will be notified.

For managing the current working directory, you should use **getenv, chdir,** and **getcwd.** The `getenv()` call is useful when you want to go to your HOME directory. **You do not have to manage the PWD environment variable.** getcwd() system call is useful to know the current working directory; i.e. if a user types pwd, you simply call getcwd(). And finally, chdir is useful for moving directories. Fore more information on these topics, read the man pages or the Advanced Unix Programming book **Chapters 4 and 7.**

Redirection

Redirection is relatively easy to implement. For example, to redirect standard output to a file, just use **close()** on stdout, and then **open()** on a file. More on this below.

With a file descriptor, you can perform read and write to a file. Maybe in your life so far, you have only used **fopen()** , **fread()** , and **fwrite()** for reading and writing to a file. Unfortunately, these functions work on **FILE*** , which is more of a C library support; the file descriptors are hidden.

To work on a file descriptor, you should use **open()** , **read()** , and **write()** system calls. These functions perform their work by using file descriptors. To understand more about file I/O and file descriptors you should read the Advanced Unix Programming book **Section 3** (specifically, 3.2 to 3.5, 3.7, 3.8, and 3.12), or just read the man pages. Before reading

forward, at this point, you should become more familiar file descriptors.

The idea of redirection is to make the stdout descriptor point to your output file descriptor. First of all, let's understand the STDOUT_FILENO file descriptor. When a command `"ls -la /tmp"` runs, the ls program prints its output to the screen. But obviously, the ls program does not know what a screen is. All it knows is that the screen is basically pointed by the STDOUT_FILENO file descriptor. In other words, you could rewrite `printf("hi")` in this way: `write(STDOUT_FILENO, "hi", 2)`.

To check if a particular file exists in a directory, use the `stat()` system call. For example, when the user types `ls`, and path is set to include both `/bin` and `/usr/bin`, try `stat("/bin/ls")`. If that fails, try `stat("/usr/bin/ls")`. If that fails too, print the **only error message.** ~~error message `Command not found.`~~

Miscellaneous Hints

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

~~Next, try working on multiple commands. Make sure that you are correctly handling all of the cases where there is miscellaneous white space around commands or missing commands.~~ Next, add built-in commands. Finally, add redirection support.

We strongly recommend that you 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. And, it's 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 junk at it and make sure the shell behaves well. Good code comes through testing -- you must run all sorts of different tests to make sure things work as desired. Don't be gentle -- other users certainly won't be. Break it now so we don't have to break it later.

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.

Handin

To ensure that we compile your C correctly for the demo, you will need to create a simple **makefile;** this way our scripts can just run `make` to compile your code with the right libraries and flags. If you don't know how to write a makefile, you might want to look at the man pages for `make` or better yet, read the tutorial.

The name of your final executable should be `whoosh`, i.e. your C program must be invoked exactly as follows:

```
emperor1% ./whoosh
```

Copy all of your .c source files into the appropriate subdirectory. Do **not** submit any .o files. Make sure that your code runs correctly on the linux machines in the galapagos (and similar) labs.

Contest

There is another contest for the shell. In your previous contest, you had to write the **fastest** sort. Now, you have to write the **shortest, completely working, readable** shell. What do we mean by **readable?** Well, the code should not be short because you have removed all white space and made it so we cannot understand what the code does. Rather, it should be short because you have removed all unnecessary redundancy. We will count the code by the number of (non-whitespace) lines; we will then examine the best few entrants to see whose code is actually readable but compact, and then choose a winner. Winner, as usual, gets a **FAMOUS 537 T-SHIRT.** Good luck!

Grading

We will run your program on a suite of test cases, some of which will exercise your programs ability to correctly execute commands and some of which will test your programs ability to catch error conditions.