CS 537 - Introduction to Operating Systems

Project 1: Command Line Interpreter (Shell)

Aman Rakesh Chadha
(9068354597)

1 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.

2 Overview

In this assignment, you will implement a **command line interpreter** 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 'csh' or the shell you are running (more likely tcsh, or bash, or for those few wacky ones in the crowd, zsh or ksh) to learn more about all of the functionality that can be present. For this project, you do not need to implement too much functionality.

3 Program Specifications

Basic Shell

Your basic shell is basically an interactive loop: it repeatedly prints a prompt "537sh% " (note the space after the percent 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 **537sh** :

```
prompt%./537sh
537sh%
```

You should structure your shell such that it creates a new process for each new command. 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.

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

Note that the shell itself does not "implement" ls or really many other commands at all. All it does is find those executables and create a new process to run them. More on this below.

The maximum length of a command line your shell can take is 512 bytes (excluding the carriage return).

Multiple Commands

After you get your basic shell running, your shell is not too fun if you cannot run multiple jobs on a single command line. To do that, we use the ";" character to separate multiple jobs on a single command line.

For example, if the user types "ls ; ps ; who" , the jobs should be run all one-by-one in sequential mode. Hence, in our previous example ("ls ; ps ; who"), all three jobs should run: first ls, then ps, then who. After they are done, you should finally get a prompt back.

Your shell also has a neat parallel mode using + as a separator. For example, if the user types "ls + ps + who" , the jobs should be run all at the same time. As before, after they are all done, you should finally get a prompt back. Note that this is different from standard Unix/Linux shells.

Mixing ; and + on a line causes the sequences of programs separated by ; to be run in parallel with other sequences. For example, ls ; echo + cat ; sort causes the sequence ls ; echo to run in parallel with the sequence cat ; sort, and the shell prints a prompt after both sequences complete.

Hint: to handle a command line with a mix of ';' and '+' , first split the command line into sequences separate by a '+' , and fork a new shell for each sequence that ends in a '+'. If the last sequence of commands does not end in a '+', you don't need a new process for it. Then, split the sequence of commands separated by ';' and then them one-at-a-time in a shell process with fork(), exec(), and wait.

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.

Note that exit could be among the multiple jobs specified on one command line (e.g., "ls ; exit ; who"). In those cases, your shell should stop after the exit and discard the remaining jobs on that line. Otherwise, it should treat the command like the others -- if

the programs are run in parallel, then the built-in commands should also run in parallel. The shell should not exit until parallel processes complete.

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 **cd** and **pwd**. Your shell users will be happy with this feature because they can change their working directory. Without this feature, your user is stuck in a single directory.

- **exit, cd, and pwd formats.** 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 getenv("HOME") to obtain this.

When you run "cd" in parallel mode, it should only affect the programs in the same sequence and not other parallel sequences nor the next command executed by the shell. For example, cd bin ; ls + ls causes the first ls to print the contents of the bin directory, but the second one does not and the next command in the shell does not use the bindirectory. However, if cd were executed in isolation or in sequential mode (in conjunction with no commands in parallel), it will affect the next command to the shell.

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 error.

Basically, when a user types pwd, you simply call getcwd(). 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:

```
prompt% cd;./537sh

537sh% pwd
/afs/cs.wisc.edu/u/m/j/username
537sh% cd /tmp
537sh% pwd
/tmp
```

- **File Output Redirection**

Many times, a shell user prefers to send the output of his/her program to a file rather than to the screen. The UNIX 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.

For example, if a user types "ls -la /tmp > file-list.txt" , the output of the ls program should be stored in the file file-list.txt.

- **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). Your shell must catch and report all errors to builtin commands.

- **White Spaces**

Zero or more spaces can exist between a command and the shell special characters (i.e. ";", "+" and ">"). All of these examples are correct.

```
537sh% ls+ls+ls
537sh% ls + ls + ls
537sh% ls > a+ ls > b+ ls> c+ ls >d
```

- **Batch Mode**

So far, you have run the shell in interactive mode. Most of the time, testing your shell in interactive mode is time-consuming. To make testing much faster, your shell should support **batch mode**.

In interactive mode, you display a prompt and the user of the shell will type in one or more commands at the prompt. In batch mode, your shell is started by specifying a batch file on its command line; the batch file contains the same list of commands as you would have typed in the interactive mode.

In batch mode, you should **not** display a prompt. You should print each line you read from the batch file back to the user before executing it; this will help you when you debug your shells (and us when we test your programs). To print the command line, **do not use printf** because printf will buffer the string in the C library and will not work as expected when you perform automated testing. To print the command line, use write(STDOUT_FILENO,...) this way:

```
write(STDOUT_FILENO, cmdline, strlen(cmdline));
```

In both interactive and batch mode, your shell should terminates when it sees the exit command on a line or reaches the end of the input stream (i.e., the end of the batch file).

To run the batch mode, your C program must be invoked exactly as follows:

```
537sh [batchFile]
```

The command line arguments to your shell are to be interpreted as follows.

- batchFile: an optional argument (often indicated by square brackets as above). If present, your shell will read each line of the batchFile for commands to be executed. If not present or readable, you should print the one and only error message (see **Program Errors** section above).

Implementing the batch mode should be very straightforward if your shell code is nicely structured. The batch file basically contains the same exact lines that you would have typed interactively in your shell. For example, if in the interactive mode, you test your program with these inputs:

```
emperor1%./537sh
537sh% ls + who + ps
some output printed here
537sh% ls > file.txt++++ ps > /non-existing-dir/file
"some output and error printed here"
537sh% ls-who-ps
some error printed here
```

then you could cut your testing time by putting the same input lines to a batch file (for example myBatchFile):

```
ls + who + ps
ls > file.txt++++ ps > /non-existing-dir/file
ls-who-ps
```

and run your shell in batch mode:

```
emperor1%./537sh myBatchFile
```

In this example, the output of the batch mode should look like this:

```
ls + who + ps
some output printed here
```

ls > file.txt++++ ps > /non-existing-dir/file+
some output and error printed here
ls-who-ps
some error printed here

- **Important Note: To automate grading, we will heavily use the batch mode**. If you do everything correctly except the batch mode, you could be in trouble. Hence, make sure you can read and run the commands in the batch file. Soon, we will provide some batch files for you to test your program.
- **Defensive Programming**. 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 the *one and only error message* whenever you encounter an error of any type.

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.
- The failure to open a batch file
- Failure to allocate memory during program initialization

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 512 characters, excluding the carriage return)
- Failure to allocate memory while executing a command

Your shell should also be able to handle the following scenarios below, which are **not errors**. The best way to check if something is not an error is to run the command line in the real Unix shell.

- An empty command line.
- An empty command between two or more '+' characters.
- Multiple white spaces on a command line.
- White space before or after the '+' character or extra white space in general.

All of these requirements will be tested extensively.

4 Miscellaneous Details

- Built-in commands exit and pwd should not be followed by any argument. Otherwise, the input is considered as invalid.
- You do **not** need to worry about redirection for built-in commands.
- When your program see an `exit' in the command line, the whole shell program, not just that one line's processing, should terminate.
- A command like `ls + exit1234' should **not** cause your shell to terminate.
- You do **not** need to worry about tab-character ('\t') in the command.

5 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. (Do not expect this detailed of advice for future assignments!) 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 (using the Unix command **man**).

Basic Shell

Parsing: For reading lines of input, you may want to look at **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 whitespace or a tab or...). Please check here for a brief tutorial of strtok.

Executing Commands: Look into **fork** , **execvp**, and **wait/waitpid**. See the UNIX man pages or the above links for these functions. Before starting this project, you should definitely play around with these functions.
You will note that there are a variety of commands in the exec family; for this project, you must use **execvp**. You should **not** use the **system()** call to run a command. Remember that if execvp() 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. You do not need to specify the full path for a program here. The path prefix will be filled according to the environment configuration. 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
```

then argv[0] = "foo", argv[1] = "205" and argv[2] = "535".

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

Multiple Commands

If you get your basic shell running, supporting multiple commands should be straightforward. The only difference here is that you need to wait for all commands (processes) to finish before return to the shell, print the prompt, and read in the next line. To do that, you simply use waitpid() again.

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. If you run built-in commands in parallel mode with '+', you still need to fork a new shell process for the command.

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. Please read the UNIX man pages, or the above links.

Output Redirection

For output redirection, you actually **only** need **dup2()**, and **open()**.

The idea of using dup2() is to intercept the byte stream going to the standard output (i.e. your screen), and redirect the stream to your designated file. dup2 uses file descriptors, which implies that you need to understand what a file descriptor is. You should read the UNIX man page, or here to get an initial understanding of what a file descriptor is.

With 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. Hence, it is impossible for you to use dup2 with these particular functions.

To work on file descriptors, you should use **open()**,**read()**, and **write()** system calls. These functions perform their works by using file descriptors. To understand more about file I/O and file descriptors you should read this man page. Before reading forward, at this point, you should get yourself familiar with file descriptor.

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 give yourself a practice, create a simple program where you create an output file, intercept stdout, and call printf("hello"). When you creat your output file, you should get the corresponding file descriptor. To intercept stdout, you should call "dup2(output_fd, STDOUT_FILENO);". If you run your program, you should not see "hello" printed on the screen. Instead, the word has been redirected to your output file.

In short, to intercept your ls output, you should redirect stdout before you execute ls , i.e. make the dup2() call before the exec('ls') call. Alternately, you can close(STDOUT_FILENO) and open() a new file; that file will be assigned the lowest available file descriptor and hence will be assigned to standard output.