Project 1a: A Unix Shell

Updates

- here are some test cases for your 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.

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 bash or the shell you are running (more likely tcsh or csh , or for those few wacky ones in the crowd, zsh or ksh or even fish) to learn more about all of the functionality that can be present. For this project, you do not need to implement too much functionality.

Readings

OSTEP Chapter 5

Program Specifications

Basic Shell

Your basic shell is basically an interactive loop: it repeatedly prints a prompt "mysh> " (note the space after >), 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 **mysh** :

```
% ./mysh
mysh>
```

You should structure your shell such that it creates a new process for each new command (there are a few exceptions to this, which we will 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.

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 line of input to the shell is 512 bytes (excluding the carriage return).

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 `wait` (the latter of which is described further below, under background jobs).

The formats for these build-in command are:

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

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:

```
% cd
% pwd
/home/username
% echo $PWD
/home/username
% ./mysh
mysh> pwd
/home/username
```

Redirection

Many times, a shell user prefers to send the output of his/her program to a file rather than to the screen. The 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 > output`, nothing should be printed on the screen. Instead, the output of the ls program should be rerouted to the output file.

If the "output" file already exists before you run your program, you should simple overwrite the file (after truncating it). If the output file is not specified (e.g. `ls >`), you should also print an error message.

Here are some redirections that should **not** work in `mysh` (however, you can play them in bash to see how bash handle these cases):

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

Background Jobs

Sometimes, when using a shell, you want to be able to run multiple jobs concurrently. In most shells, this is implemented by letting you put a job in the "background". This is done as follows:

```
mysh> ls &
```

By typing a trailing ampersand, the shell knows you just want to launch the job, but not to wait for it to complete. Thus, you can start more than one job by repeatedly using the trailing ampersand.

```
mysh> ls &
mysh> ps &
mysh> find . -name *.c -print &
```

Of course, sometimes you will want to wait for jobs to complete. To do this, in your shell you will simply type wait. The built-in command `wait` should not return until **all** background jobs are complete. In this example, wait will not return until the three jobs (ls, ps, find) all are finished.

```
mysh> ls &
mysh> ps &
mysh> find . -name *.c -print &
mysh> wait
```

Program Errors

The one and only error message. A section about the Error Message has been added. In summary, 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

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

```
mysh> ls
mysh> ls > a
mysh> ls>a
```

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, 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:

```
mysh [batch_file]
```

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

> batch_file: an optional argument (often indicated by square brackets as above). If present, your shell will read each line of the batch_file for commands to be executed. If batch_file does not exist, or is not readable, you should print the one and only error message (see Error Message section below).

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:

```
% ./mysh
mysh> ls
some output printed here
mysh> ls > /tmp/ls-out
some output printed here
mysh> notACommand
some error printed here
```

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

```
ls
ls > /tmp/ls-out
notACommand
```

and run your shell in batch mode:

```
prompt> ./mysh my_batch_file
```

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

```
ls
some output printed here
ls > /tmp/ls-out
some output printed here
notACommand
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 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 stdout 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 512 characters, excluding the carriage return)

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

- 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. (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 ...). Some have found `strchr()` useful as well.

Executing Commands: Look into `fork` , `execvp` , and `wait` / `waitpid` . See the UNIX man pages for these functions, and also read the Advance Programming in the UNIX Environment, Chapter 8 (specifically, 8.1, 8.2, 8.3, 8.6, 8.10). 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, 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
```

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!

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. For more details, read the Advanced UNIX Programming book Chapter 4.22 and 7.9 .

Redirection

Redirection is relatively easy to implement: just use `close()` on stdout and then `open()` on a file. You can also use `dup2()` system call which is safe for concurrency.

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.

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). 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)`.

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. Finally, you add built-in commands and redirection suppors.

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.

Bonus

Pipe

Another important feature of a real world shell is **pipe** (represented by "|"), which connects the output of one process to the input of another process. For example, the following will calculate how many times the string "hello" occurs in file "tmp":

```
% grep -o "hello" tmp | wc -l
```

There are two (connected) commands:

- `grep -o "hello" tmp` finds all occurrences of "hello" in file "tmp", and outputs with format "each occurrence per line" to the `STDOUT_FILEND`.
- `wc -l [FILE]` calculates the number of lines in FILE. If FILE is not given, `wc` reads from `STDIN_FILENO`.

Here, by the pipe, we connect the output of `grep` to the input of `wc`, and finally output the number of "hello" in "tmp". Similarly, the following longer pipe finds the number of unique lines which contain string "hello".

```
% grep "hello" tmp | sort | uniq | wc -l
```

If "tmp" looks like

```
hello 1 world
love 2 story
hello 2 world
hello 1 world
social 3 network
```

Then the output is 2. Can you figure out how it works?

To integrate pipe in your mysh, the idea is quite similar to redirection: you need to bind file descriptors `STDIN_FILENO` and `STDOUT_FILENO` properly before running a command. One key system call is `pipe()` which kindly creates the pipe for you (happy, yeah!). Again, details of `pipe` can be found in the man page, and also section 15.2 of Advanced UNIX Programming book. Read them, call `pipe()`, revisit redirection, make your pipe rock.

Hand In

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 mysh , i.e. your C program must be invoked exactly as follows:

```
% ./mysh
% ./mysh input_test_tile
```

The files you submit should include:

- .c souce files
- A makefile which builds `mysh`

If you want to get the bonus, you also need to hand in

- A README file explaining your implementation of pipe

Do not submit any .o files. Make sure that your code runs correctly on Linux machines.

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. Be sure that you thoroughly exercise your program's capabilities on a wide range of test suites, so that you will not be unpleasantly surprised when we run our tests.

Adapted from WISC CS537 by Remzi Arpaci-Dusseau