Computer Science Department

Mount Holyoke College

CS 322
Operating Systems
Programming Assignment 3
Writing a shell
Due: October 19, 11:30 PM

Goals

- To get more experience with fork, wait and exec
- To learn how shells work
- To learn about the Unix search path

Background

The shell is a program where the user enters commands by typing and it parses and executes those commands. It is the program that runs inside the Terminal application on both the Mac and Linux. There are actually many different shell programs available, with bash being a very popular shell. You can find out which shell you are running by using this command:

```
-> echo $SHELL
/bin/bash
```

`SHELL` is an environment variable. `$SHELL` gives me the value of the variable. The `echo` command displays it on the screen.

Shells are surprisingly complicated. You can do "`man bash`" to get a sense of their complexity. (Don't worry; the shell we write will be very simple.)

It turns out that when we type a command line at a shell prompt, it is one of two things. It can be a built-in command that the shell itself knows how to execute. Alternatively, it can be an executable program that lives in its own file. For example, when we type `./textsort`, we are running the `textsort` program that we wrote. When we type `ls`, we are running the `ls` program that lives in `/bin/ls`. When we type `cd` to change directory, we are executing a built-in command that the shell knows how to execute itself. In general, it may not be obvious which commands are built-in and which are separate programs, but they are executed quite differently.

Built-in commands are executed by the shell itself. The code to implement these is built directly into the shell program. When we enter a command, the shell looks to see if it is a built-in command, and, if so, runs the corresponding code.

If the command we enter is not a built-in command, it will look for a file with the given name. So, when we type `./textsort`, it looks in the current directory (signified by the .) for the file named `textsort`. It then forks a process, uses exec to load and run the program, and waits for that child process to complete before displaying the prompt again.

What if the user types "ls"? This is a program in /bin/ls, but how does the shell know to look in /bin? The shell uses a search path. This is a list of directories. It starts at the beginning of the list and looks in each directory for a matching file until it finds it. You can find out what search path the shell is using by displaying the value of the PATH environment variable. Here is what I see in my Linux VM:

```
-> echo $PATH
/home/blerner/bin:/home/blerner/.local/bin:/usr/local/sbin:/usr/local/
bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

So, when I enter "ls", it first looks for the program in /home/blerner/bin, then in /home/blerner/.local/bin, etc. until it eventually finds it in /bin.

Another handy command is to just find out where a particular executable program is on your path, using the "which" command:

```
-> which ls
/bin/ls
```

Assignment

Basic Shell: Whoosh

Your 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 program should be whoosh. Here is how it should look when you start it:

```
blerner@blerner-VirtualBox:./whoosh
whoosh>
```

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, your shell will invoke your implementation of the built-in command.

Most Unix shells have many built-in commands such as cd, echo, pwd, etc. In this project, you should implement just 4 built-in commands: exit, cd, pwd, and path.

exit

The exit command should cause whoosh to terminate. To implement the exit built-in command, you should simply call exit(0); in your C program.

pwd

When a user types pwd, you simply call getcwd(), and show the result.

cd

When the user enters 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.

When the user enters `cd` (with arguments), your shell should change the working directory to the directory specified by the argument. To change directories, use C's `chdir` function.

You do not have to support tilde (~).

cd / pwd example

```
whoosh> cd
whoosh> pwd
/home/blerner
whoosh> cd Documents
whoosh> pwd
/home/blerner/Documents
```

Executing external programs

Most of the commands that a shell executes are not built-in commands but programs that exist in separate executable files, like `gcc`. If the user enters a command that is not one of the built-in commands, your shell should look in the `/bin` directory for a file with that name. If it finds a matching file, it should fork a process to run the program in, exec the program and wait for it to complete before giving another prompt.

Note that the command might contain arguments, like "`ls -la /tmp`". Your shell should run the program `/bin/ls` with all the given arguments and display the output on the screen.

When whoosh starts, it should look only in `/bin` for external programs.

One more built-in command: path

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 shell 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.

A typical usage would be like this:

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

Note that the directories on the path are separated by spaces.

By doing this, your shell will know to look in `/bin` first and then `/usr/bin` when a user types a command, to see if it can find the proper binary to execute. If the program is found in /bin, it should execute that program. If it is not found in /bin, it should look in /usr/bin and execute it, if it is found there.

If the user sets path to be empty, then the shell should not be able to run any programs (but all the built-in commands, such as `cd` and `path`, should still work).

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 tested with automated testing, you should print this one and only error message whenever you encounter an error of any type:

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

Do not modify the error message or add extra error messages.

For the following situation, your shell should call `reportError` and exit gracefully:

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

For the following situations, you should call `reportError` 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.

If you call a C library function or a C system call and it returns with an error, you should call `reportError` and continue if you can. If the error is one for which you can't recover (like running out of memory), you should exit whoosh.

You should be sure that you have no memory problems, including memory leaks. The easiest way to check for these problems is to run valgrind.

Assumptions

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

Use git to manage your code

We now have git set up so that both members of a team can use the same repository to share code. I recommend that you use git to store versions of your program whether you are working in a team or alone. If you are working with a partner, the instructions to get started are a little different than before.

If you are working alone and you created a git repository for it, you can continue to use that repository. If you are working alone but did not create a git repository last time, please refer to the instructions from the last assignment on how to do that.

The following instructions are for those students working in a group.

There is now a git server running on cs322.cs.mtholyoke.edu and the initial empty repositories have already been created.

1. If you have not already done this, install git in your Linux VM.

```
sudo apt install git
```

2. **Clone the repository in your Linux VM.** Substitute your user id for blerner and your group id for g01. Groups ids are given below with the team list.

```
git clone blerner@cs322.cs.mtholyoke.edu:/repos/g01
```

The remaining instructions to configure git, add, commit, push and pull changes are the same as in programming assignment 2.

Teams

You will work in groups on this assignment. Here are the groups. The g number, like g02, is the group id to use in the clone command.

g02: Tricia, Miriam, Caitlin
g03: Lisa, Hye Yoon
g07: Briar, Srishti
g09: Sanaa, Regina
g11: Deepshikha, Raessa
g12: Onji, Linh
g13: Olivia, Veneta
g14: CiCi, Jessica
g15: Tien, Kayla
g16: Elizabeth, Surabhi

If your name does not appear in the list above, it means that I received an email from you indicating that you wanted to work alone on this assignment so you are not assigned to a group.

Grading

Grading will be based on correctness, documentation, and coding style in these proportions:

60% Correctness (broken down as follows)

- 5% Starting up
- 10% Parsing the command line
- 5% exit
- 5% pwd
- 5% cd
- 10% path
- 15% Running executable programs
- 5% Miscellaneous

20% Correct memory management

10% Comments

10% Coding style

Turning in your solution

Place your C file(s) and makefile into a single tar.gz file, using your name and your partner's name rather than mine in the name of the file. (First names are enough.)

tar -cvzf Barbara_Lerner_Assign1.tar.gz *.c makefile

Upload your tar.gz file to Moodle. Only one of you should submit this.