Project 2: Developing a Linux Shell

In this project, you'll build a simple Unix/Linux shell. The shell is the heart of the command-line interface, and thus is central to the Linux/Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.

There are three specific objectives to this assignment:

- To further 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 (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 Linux. If you don't know what shell you are running, it's probably `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 (and the associated Shell Introduction file).

Program Specifications

Basic Shell: `myshell`

Your basic shell, called `myshell` is fundamentally an interactive loop: it repeatedly prints a prompt `myshell>` (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 `myshell`.

The shell can be invoked with either no arguments (interactive) or a single argument (batch0; anything else is an error. Here is the no-argument way:

```
prompt> ./myshell

myshell>
```

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

The mode above is called *interactive* mode, and allows the user to type commands directly. The shell also supports a *batch mode*, which instead reads input from a batch file and executes commands found in the file. Here is how you run the shell with a batch file named `batch.txt`:

```
prompt> ./myshell batch.txt
```

There is a difference between batch and interactive modes: in interactive mode, a prompt is printed (`myshell>`). In batch mode, no prompt should be printed during execution of commands.

You should structure your shell such that it creates a process for each new command (the exceptions are *built-in commands*, discussed below). 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 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).

Your project is to develop/write a simple shell - `myshell` - that has the following properties:

1. The shell must support the following internal commands:

   a. cd <directory> - Change the current default directory to <directory>. If the <directory> argument is not present, report the current directory. If the directory does not exist an appropriate error should be reported. This command should also change the PWD environment variable.
   b. clr - Clear the screen.
   c. dir <directory> - List the contents of directory <directory>.
   d. environ - List all the environment strings.
   e. echo <comment> - Display <comment> on the display followed by a new line (multiple spaces/tabs may be reduced to a single space).
   f. help - Display the user manual using the more filter.
   g. pause - Pause operation of the shell until 'Enter' is pressed.
   h. quit - Quit the shell.
   i. The shell environment should contain shell=<pathname>/myshell where <pathname>/myshell is the full path for the shell executable(not a hardwired path back to your directory, but the one from which it was executed).

2. All other command line input is interpreted as program invocation, which should be done by the shell **fork**ing and **exec**ing the programs as its own child processes. The programs should be executed with an environment that contains the entry: `parent=<pathname>/myshell` where `<pathname>/myshell` is as described in 1.i. above.

3. The shell must be able to take its command line input from a file. That is, if the shell is invoked with a command line argument:

   ```
   myshell batchfile
   ```

   then `batchfile` is assumed to contain a set of command lines for the shell to process. When the end-of-file is reached, the shell should exit. Obviously, if the shell is invoked without a command line argument, it solicits input from the user via a prompt on the display.

4. The shell must support I/O - redirection on either or both ***stdin*** and/or ***stdout***. That is, the command line

   ```
   programname arg1 arg2 < inputfile > outputfile
   ```

   will execute the program *programname* with arguments *arg1* and *arg2*, the stdin FILE stream replaced by *inputfile* and the stdout FILE stream replaced by *outputfile*.

   stdout redirection should also be possible for the internal commands

   ```
   dir,  environ, echo, & help.
   ```

   With output redirection, if the redirection character is > then the outputfile is created if it does not exist and truncated if it does. If the redirection token is >> then outputfile is created if it does not exist and appended to if it does.

5. The shell must support background execution of programs. An ampersand (**&**) at the end of the command line indicates that the shell should return to the command line prompt immediately after launching that program.

6. You are to include an implementation of command line pipes. (essentially an extension of redirection) so that commands can be strung together. An example is

   ```
   cat out.txt | wc –l
   ```

7. The command line prompt must contain the pathname of the current directory.

Note: You can assume that all command line arguments (including the redirection symbols, <, > & >> and the background execution symbol, &) will be delimited from other command line arguments by white space - one or more spaces and/or tabs (see the command line in 4. above).

Project Requirements

1. Design a simple command line shell that satisfies the above criteria and implements it on the specified Linux platform.

2. Write a simple manual describing how to use the shell. The manual should contain enough detail for a beginner to Linux to use it. For example, you should explain the concepts of I/O redirection, the program environment, and background program execution. The manual MUST be named readme_doc and must be a simple text document capable of being read by a standard Text Editor.

   For an example of the sort of depth and type of description required, you should have a look at the online manuals for csh and tcsh (man csh, man tcsh). These shells obviously have much more functionality than yours and thus, your manuals don't have to be quite so large or with such detail.

   You should NOT include building instructions, included file lists or source code - we can find that out from the other files you submit. This should be an Operator's manual not a Developer's manual.

3. The source code MUST be extensively commented and appropriately structured to allow your peers to understand and easily maintain the code. Properly commented and laid out code is much easier to

interpret, and it is in your interests to ensure that the person grading your project is able to understand your coding without having to perform mental gymnastics!

4. Your solution to the project will be submitted through the Project 2 assignment in Canvas. You will also develop the solution to the project through a GitHub repo. A link to the GitHub repo is to be included as a Canvas project comment.

5. The Canvas submission should contain only source code file(s), include file(s), a `makefile` (all lower case please), and the `readme_doc` file (all lowercase, please). No executable program should be included. The TA will be automatically rebuilding your shell program from the source code provided in Canvas. If the submitted code does not compile it cannot be graded and will lose significant points!

6. The `makefile` (all lowercase, please) **MUST** generate the binary file `myshell` (all lower case please). A sample `makefile` would be

```
# Joe Citizen, s1234567 - Operating Systems Project 1
# CompLab1/01 tutor: Fred Bloggs
myshell: myshell.c utility.c myshell.h
gcc -Wall myshell.c utility.c -o myshell
```

   The program `myshell` is then generated by just typing `make` at the command line prompt.

   Note: The fourth line in the above `makefile` **MUST** begin with a tab.

7. In the instance shown above, the files in the submitted directory would be:

```
makefile
myshell.c
utility.c
myshell.h
readme_doc
```

Submission

A `makefile` is required. All files in your submission will be copied to the same directory, therefore, do not include any paths in your `makefile`. The `makefile` should include all dependencies that build your program. If a library is included, your `makefile` should also build the library.

Do not hand in any binary or object code files. All that is required is your source code, a `makefile` and `readme_doc` file. Test your project by copying the source code only into an empty directory and then compile it by entering the command `make`.

Required Documentation

Your source code will be assessed and graded as well as the `readme_doc` manual. Commenting is required in your source code. The user manual can be presented in a format of your choice (within the limitations of being displayable by a simple Text Editor). Again, the manual should contain enough detail

for a beginner to Linux to use the shell. For example, you should explain the concepts of I/O redirection, the program environment and background execution. The manual **MUST** be named `readme_doc` (all lowercase, please, **NO** `.txt` extension).

You are to create a document describing your problem solution, and program implementation and program documentation. This document should also include your descriptions of testing methods and results of applying the tests. Submit this document along with your assignment submission.

REMEMBER: This is to be **your work**. This is not a unique project; that is, in the general sense it is a project given to previous students and at other Universities. You learn nothing from using code developed by others. **We will not tolerate code or documentation that is not your own**.

Notes on Shell Structure

Basic Shell

The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command `exit`, at which point it exits. That's it!

For reading lines of input, you should use `getline()`. This 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 this case, the shell should not read user input (from `stdin`) but rather from this 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, you might want to use strtok() (or, if doing nested tokenization, use strtok_r()). Read the man page (carefully) for more details.

To execute commands, look into `fork()`, `exec()`, and `wait()/waitpid()`. See the man pages for these functions, and also read the relevant book chapter for a brief overview.

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()` library function 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.

Paths

In our example above, the user typed ls but the shell knew to execute the program `/bin/ls`. How does your shell know this?

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 (except built-ins). All it does is find those executables in one of the directories specified by `path` and create a new process to run 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' ``

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 shell source code, which then will exit the shell.

In this project, you should implement `exit`, `cd`, 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 take one argument (0 or >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. A typical usage would be like this: `myshell> path /bin /usr/bin`, which would add `/bin and /usr/bin` to the search path of the shell. If the user sets path to be empty, then the shell should not be able to run any programs (except built-in commands). The `path` command always overwrites the old path with the newly specified path.

Redirection

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

If the `output` file exists before you run your program, you should simple overwrite it (after truncating it).

The exact 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.

Parallel Commands

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

```
myshell> cmd1 & cmd2 args1 args2 & cmd3 args1
```

In this case, instead of running `cmd1` and then waiting for it to finish, your shell should run `cmd1`, `cmd2`, and `cmd3` (each with whatever arguments the user has passed to it) in parallel, *before* waiting for any of them to complete.

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).

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`).

Next, add built-in commands. Then, try working on redirection. Finally, think about parallel commands. Each of these requires a little more effort on parsing, but each should not be too hard to implement.

At some point, you should make sure your code is robust to white space of various kinds, including spaces ( ) and tabs (`\t`). In general, the user should be able to put variable amounts of white space before and after commands, arguments, and various operators; however, the operators (redirection and parallel commands) do not require whitespace.

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. It's also 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 different inputs at it and make sure the shell behaves well. Good code comes through testing; you must run many different tests to make sure things work as desired. Don't be gentle -- other users certainly won't be.

Finally, keep versions of your code. Take advantage of the features of GitHub. Use branches to create options for trying options in your program. Perform commits often to ensure keeping version information for your changes.