CS 537 Spring 2019, Project 2a: Unix Shell

Change Log:

- **2/9/19 -** Defined behavior of `history` command when executed as first argument.
- **2/11/19 -** Clarified piping for built-in and error behaviors.
- **2/15/19 -** Added FAQ.

Important Note: Due to concerns raised on Piazza additional clarifying requirements and examples have been added for the `history` built-in command.

FAQ

- **Piping when first command is empty** Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=487)
- **Redirection when there is no command,** `> out.txt` Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=531)
- **History Arguments** Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=413) Q2 (https://piazza.com/class/jql5yyu0x8a12k?cid=591) Q3 (https://piazza.com/class/jql5yyu0x8a12k?cid=404) Q4 (https://piazza.com/class/jql5yyu0x8a12k?cid=313) Q5 (https://piazza.com/class/jql5yyu0x8a12k?cid=313)
- **'cd ' , cd with zero argument** Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=540)
- **'wish> ' apears late** Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=452)
- **History skips empty lines** Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=580)
- **Redirection errors** Q1 (https://piazza.com/class/jql5yyu0x8a12k?cid=527)

In this project, you'll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the 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 shell that you implement will be similar to, but simpler than, the one you run every day in Unix. 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.

Program Specifications

Basic Shell: `wish`

Your basic shell, called `wish` (short for Wisconsin Shell, naturally), is basically an interactive loop: it repeatedly prints a prompt `wish>` (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 `wish`.

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

```
prompt> ./wish
wish>
```

At this point, `wish` 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 from therein. Here is how you run the shell with a batch file named `batch.txt`:

```
prompt> ./wish batch.txt
```

One difference between batch and interactive modes: in interactive mode, a prompt is printed (`wish>`). In batch mode, no prompt should be printed.

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).

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 (http://www.ostep.org/cpu-api.pdf) 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'

Note: Most shells allow you to specify a binary specifically without using a search path, using either **absolute paths** or **relative paths**. For example, a user could type the **absolute path** `/bin/ls` and execute the `ls` binary without a search path being needed. A user could also specify a **relative path** which starts with the current working directory and specifies the executable directly, e.g., `./main`. In this project, you **do not** have to worry about these features.

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 wish source code, which then will exit the shell.

In this project, you should implement `exit`, `cd`, `history`, 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.
- `history`: When the user enters the `history` command with no arguments, then the shell should print a list of all the lines of input which the user has entered since the shell was started **including the `history` command**. The user may also supply a single argument to the `history` command `n` where `n` is a number. When this happens the shell should print the previous `n` lines of input *(if `n` is not an integer, then round its value up to the next*

greatest integer). If `n` is greater than than the total number of lines input by the user, then all previous lines of input should be printed. If the first arguement to the history command is not an integer or if the number of arguments is greater than one, then an error should be printed and the input ignored. Examples of using the history command are shown below:

Example #1

```
prompt> ./wish
wish> ls
my_dir my_file.txt
wish> cd my_dir
wish> ls
my_other_file.txt
wish> cat my_other_file.txt
Hello, world!
wish> history
ls
cd my_dir
ls
cat my_other_file.txt
history
wish> history 3
cat my_other_file.txt
history
history 3
wish> history 10
ls
cd my_dir
ls
cat my_other_file.txt
history
history 3
history 10
wish>
```

Example #2

```
prompt> ./wish
wish> history
history
wish>
```

- `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: `wish> 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. Also, notice in the previous example that one path ends with a `/` while the other does not. The shell should be robust enough to run programs in both directories.

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.

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

Piping

Another popular feature which shells offer is the ability to send the output of one command onto a second command in a single line of input. This allows users to chain commands together without using intermediate files. This ability is commonly known as *piping* because the "pipe" character `|` is often used to represent the command.

For example, if a user types `ls -la | more`, then the standard output and standard error output of `ls -la` should be given as the standard input of `more`. Then the output of `more` should be printed to the screen. Note, this is to be done without the use of redirection through an intermediate file.

Note: don't worry about piping for built-in commands (e.g., we will not test what happens when you type `history | grep ls`).

Important Note 1: In our shell, no more than two commands will be piped in a single line of input from the user. So, if the user enters a command with more than one `|`, then an error should be printed and the input ignored.

Important Note 2: The use of a pipe and output redirection in a single line of input is not supported by our shell. So, if the user submits input which contains both a `|` and `>`, an error should be printed to the screen and the input ignored.

Important Note 3: If a line of input contains `|` with no following command (ex. `ls |`), then the input should be ignored and an error be printed.

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 ~~*any*~~ most errors, your shell should simply *continue 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. 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 (for example, the redirection command) 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. 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.

Test

To test your Unix Shell: 1) Navigate to the directory that contains your wish.c. 2) Run the script "runtests" that can be found in /u/c/s/cs537-1/tests/p2a. You may copy it to another location if you prefer.

Due Date

This project is due on February, 15th, 2019 at 11:59PM.

Submiting Your Implementation

It is possible to implement the shell in a single `.c` file, so your handin directory should contain at minimum:

- 1 make file named `Makefile`

- 1 README file named `README.md`
- 1 `.c` file containing the source code for your implementation of the shell

The directory may contain more files than these three, but extra files should be either more source files `.c` and `.h`. If you use multiple source files, then please use `.h` files appropriately to maitain a clean and well-formed code base.

Use the README file for any explanation you feel would be helpful to someone trying to understand the structure of your code and files (if you have multiple source files).

The make file included in your submission should be behave in the following mannner. When the `make` command is executed in your handin directory an executeable file named `wish` should be created such that when `./wish` is executed in the handin directory your implementation of the shell begins running in interactive mode.

When the make file compiles source code it should use the `gcc` compiler followed by both the `-Wall` and `-Werror` flags. So, any build targets in your make file should match the following regular expression: `[A-z0-9]+:\n\tgcc -Wall -Werror`. If you would like to confirm that the build targets in your make file are correct, then you may use RegExr (https://regexr.com/) to verify that the build targets match the expression.

Below is an example of a make file for an implementation which is contained in a single `.c` file:

```
default:
    gcc -Wall -Werror wish.c -o wish

clean:
    rm wish
```

If you are unfamiliar with make files, then you can learn more about on the GNU website (https://www.gnu.org/software/make/manual/html_node/Introduction.html).