Programming Assignment 1

Basic Shell

Due: June 21, 2023

1 Introduction

This project aims to get you familiar with forking processes and engaging with inter-process communication by having you implement a `UNIX`-style shell with basic functionality.

2 Given Materials

On Brightspace under **Content/Assignments/Assignment 1** you'll see two files:

- shell.c
- makefile

`shell.c` contains code to help get you started on making a shell. It has a basic parser which will tokenize any input you give to it. `makefile` has a basic compilation line which will output the executable binary `shell`. You can run the makefile by typing `make` into the directory with both the makefile and c file. You may change the makefile if you so choose to, but **do not** change the executable name; I will expect the binary to be named `shell` when I go to test your shell.

3 Requirements

This is meant to be a rather basic introduction into the wonderful world of `UNIX` system calls. You will be expected to code a rudimentary shell in `C` capable of executing some of the basic commands and functionality associated with more complex shells (like `bash`).

3.1 Part A

Create a shell capable of executing simple commands. The shell should create a child process and transmogrify that process into the user-specified program. For example, if the user types in:

```
sh550>echo hello world
```

I expect the shell to look like this afterwards:

```
sh550>echo hello world
hello world
sh550>
```

Hint: Take a look at the man pages for `fork` and the various `exec` calls (`execl`, `execlp`, `execle`, `execv`, `execvp`, `execvpe`) to figure out which one would be right for a shell.

3.2 Part B

Extend the functionality of your shell to include two brand new history commands: **hist** and **!!**.

3.2.1 hist

hist should display the history of all commands sent in to the shell up to that point (including the **hist** command itself!). The history should be displayed in the proper order with its ordinal number and a period behind each one. For example, given the following shell state:

```
sh550> echo hello world
hello world
sh550> ls | wc -l
5
```

typing **hist** should display:

```
sh550> hist
1. echo hello world
2. ls | wc -l
3. hist
```

3.2.2 !!

!! will perform the last entered command again. For example, if you type in the following command:

```
sh550>echo hello world
hello world
```

and your next command is **!!**, your shell should look like the following:

```
sh550>echo hello world
hello world
sh550 >!!
hello world
```

<< Note: **!!** does not need to alter the shell's history. Meaning that our previous call to hello world will only show up once after a subsequent call to **hist**. >>

Hint: You may implement this function any way you like, but there's no need to get any fancier than a simple linked list!

3.3 Part C

Further extend the functionality of Part A and B by allowing for simple redirects. **You only need to support one redirect at a time. I do not expect long redirect chains to function properly.**

3.3.1 |

The pipe character should work just the same as it does in a standard `UNIX` shell. It will pipe the output of the first command into the input of the second command. For example, let us use the output of a call to `cat` and count the lines with a call to `wc -l`:

```
sh550> cat genesis.txt | wc -l
287
```

Hint: Take a special look at the man page for `dup2` and think about how you'll want to structure your pipes!

3.3.2 > and <

The > will take the output of a process and write it into a given file, whereas < will use the contents of a file as the input for a given command. For example, we can write an `echo` into a file `hi.txt` and then read from that file using `cat`:

```
ssh550>echo hello world > hi.txt
ssh550>ls
hi.txt  shell  shell.c
ssh550> cat < hi.txt
hello world
```

<< Note: You do not need to make redirects function for your Part B commands. >>

4 Evaluation

I will not be going out of my way to test every edge-case possible for your shell, but I do expect each part to work reasonably well with basic shell commands. I have a select handful of secret commands I will be running to ensure a base level of correctness; partial credit will be given if appropriate. If you are unsure what the correct output should be for a given command set, try it out on the Linux terminal!

I also expect to see comments throughout your source code. Your comments do not need to be overly detailed, but they should provide a high level explanation as to why certain design choices were made.

If I can't compile your code, you will receive a 0 for the project. Compilation will give you a baseline of 15 points. Part A is worth 25 points, Part B is worth 25 points, and Part C is worth 35 points.

5 Extra Credit

For those adventurous enough to go a bit beyond expectations, I will add a point onto your final class grade if you can chain an indefinite number of redirects. The following command should work flawlessly:

```
sh550> cat < genesis.txt | grep water | wc -l > water_mentions.txt
sh550> cat water_mentions.txt
11
```

6 Submission

As stated previously, make sure I can run your makefile and that the binary is named `shell`. Submit your code and makefile in a `tar.gz` zipped file with the following naming format <email_username>_shell.tar.gz. If I were submitting my file would be named as follows: jraskin3_shell.tar.gz

(You can compress a file by running `tar -czvf name-of-archive.tar.gz shell.c makefile`)