Clifton Dixon-Ferrell
4/22/19

WISH Shell Design Document

Capabilities:
This shell is capable of processing any command that does not require pipe() system calls. Therefore, this shell will be able to process commands using *ls* or *grep* but not commands like *ls -l | grep -v "foo" | more* . This shell is, however, able to do input/output redirection, commands with additional arguments and no-argument commands. It also terminates upon calling exit

Prompt:
The shell prompt for a wish shell is relatively simple. Upon loading the shell, the user will see a prompt reading "USER@WISH" with a # sign one line below. "USER" is written in blue, the @ symbol is in white, "WISH" and the # sign is in yellow, and the user text is written in cyan. The c code for each of these colors can be found further below in the 'Colors' section of the design document

Code:
The shell is run through a while loop that loops permanently unless broken out of by an exit command. A char** array named 'args' holds whatever text is written in by the user. A for loop is then made that loops through each argument in the array. If the 1st argument is an exit command, the program exits and the shell ends. Otherwise, if the loop has looped to the end of the array, the program calls runcommand(char** args).

runcommand() does the majority of command management and holds the majority of the code. It uses if-else statements as a way of checking for 3 cases.

- Case 1: the command contains '>>', meaning it requires file appending.
- Case 2: the command contains '>', meaning the command requires output redirection.
- Case 3: the command contains '>', meaning the command requires input redirection.
- Default: command can be run immediately.

We check if Cases 2 & 3 are satisfied by running a function int *contains(char* string, char** arr)*. The *contains()* command checks loops through the array and returns the element of the array thestring is found, or 0 if it doesn't find it at all. Therefore, if contains('>>',args) returns 0, there is no call for file appendage by the command. Case 1 uses a similar function called int *appendcontains(char** arr)* which loops through *arr* to find if both elements i and i+1 are the '>' symbol. If so, return i. Else, return 0.

File Appending:
If running the appendcontains() function didn't return 0, you will want to declare a new int *i* and set it equal to whatever number appendcontains() returns. You will also declare two new ints *f* and *p*. Then set p = fork(). If p == 0 (meaning the process is the child), you open the file listed in the char array using the open() system call and set the cursor to the end of the file using

the lseek() system call. Then, dup the file(using dup()), close the file(using close()), and execute the command(using execvp()). If the process is the parent, it just waits until the child process is finished.

Output Redirection:

If running contains('>',args) doesn't return 0, you will want to declare a new int *i* and set it equal to whatever number contains() returns. That integer i will provide the index of the '>' symbol in the character array so that we may set that element to NULL. We then fork the program as above. If the process is the parent, it waits until the child process has finished. If the process is the child, it opens the file listed in the command and sets it to be written to instead of stdout(using dup() and close()). Once closed, the command is then executed using execvp().

Input Redirection:

If running contains('<',args) doesn't return 0, you will want to declare a new int *i* and set it equal to whatever number contains() returns. hat integer i will provide the index of the '<' symbol in the character array so that we may set that element to NULL. We then fork the program as above. If the process is the parent, it waits until the child process has finished. If the process is the child, it opens the file from the command line using open() while also making sure that file is set to read-only. Then, run close() on the stdin and dup the open file. Finally, execute the command using execvp().

Default:

In this case, we fork the program like above. If the process is the parent, it waits for the child process to finish. If the process is the child, it executes the command using execvp().

End of Shell Code:

__**Additionals**__

Colors:

For the shell prompt, I used color text to add style to the prompt. To make things easier for myself when changing the color of the text, I made functions for each color I planned on using.Below are the following functions I wrote with the ones that I used being written in bold:

```
void red() { printf("\033[0;31m"); }
void blue() { printf("\033[0;34m"); }
void green() { printf("\033[2;32m"); }
void cyan() { printf("\033[0;36m"); }
void yellow() { printf("\033[0;33m"); }
void mustard() { printf("\033[2;33m"); }
void ocean() { printf("\033[2;34m"); }
void white() { printf("\033[0m"); }
```