Simple Shell Design

Seifeldin Ashraf
900183864

Data Structures Used:

I used only C-Strings (char*) for all the buffers, and commands parsing tokens. Multidimensional arrays of characters (char**) were used to store the arguments (the different parts of the command). The maximum buffer length is 1024 bytes.

Functions Description:

size_t read_command(char *cmd)
This function takes one input (char*) and output a size_t number. It returns the command length, and handles the **CTRL+D** input to make the shell terminate.

int build_args(char * cmd, char ** argv)
This function takes char* as first input, and char** as second input. It splits the command to a number of tokens or arguments using strtok and stores in char** argv to be used later by the program, and returns their number as output.

void set_program_path (char * path, char * bin, char * prog)
This function takes three char* inputs, one for the path which is the program path that gonna be needed to execute the command (the path depends on the command), one is bin which is the bin path "/bin", and the last one is the program path which is actually the first argument of the command. The function builds the program path by concatenating bin, and prog using strcat to the path argument to be returned and used by execve to execute the command.

int commandType (char* cmd)
This function basically takes one input which is the command, and using strchr it searches for the character that each command being identified with, like | for pipes, and < > for redirections. Then if a special character was found it returns a number that indicates the command and consequently the program will behave accordingly.

int countCMD (char* cmd, char simpleCommands)**
This function was implemented specifically for pipes. It takes the command as a char* and then tokenize each sub command using strtok that are being piped together and stores in it char** simpleCommands to be used later to execute each command individually.

void resolveRedirections (char* line)
This function takes the input command whose type may be mix between input and output redirection, pipes, mix between pipes and redirection. It saves the stdin, and stdout using dup(0), and dup(1) in order to get it back after processing the redirection commands. Then, it checks if there is any input redirection using strchr checking for '<' and if found, it parses the file name and then redirects the input from stdin to be taken from the file, if there is no input redirection we take the input from the stdin. The function then invokes the countCMD function to get the number of

commands in case of pipes, and returns the commands in char** simpleCommands. Then it iterates over each command, executes it by forking and invoking execve, and opens a pipe to pass its out to the next command. If it reaches the final command, the function checks if there is output redirection, and it redirects the output from stdout if so. At the end, the function redirects the input and output streams back to stdin and stdout.

void handleTicks (char* var, char* cmd)
This function receives two inputs, char* which is the environment variable to set, and the command whose output should be stored in this variable like (x=`cat /etc/hosts`). The function opens a pipe to redirects the output from stdout to fd[0]. By invoking the read function we read the output of this command to a buffer, which then is used in setenv to set the variable to the output of the command and then it is done.

void handleRedirection (char *filename, int input)
This function handles the redirection commands. It receives the file name that be redirected the output to or the input from. It also receives a signal that indicates whether the redirection is input or output. Using dup2 function it redirects the input/output to the filename is passed.

THE BONUS PART (globbing)
The globbing functionality executes commands like (ls -l *.txt). Once the shell identifies that the command needs globbing, it parses the pattern from the command. The pattern in globbing is like *.txt in the previous example. Then using the glob function which takes the pattern and a buffer to store the output we get all the files with the pattern needed. We can get the number of results using <buffer_var_name>.gl_pathc. Then we can using a simple for loop print all the results, and then freeing the buffer usign globfree() function.

Logical Flow of The Program:
The program starts by taking the command input from the user, and then:

1. invoking **build_args**
2. checking for the simple commands like cd, pwd, ls, exit, whoami, and echo.
3. It executes each one of the simple commands using execve, and some of them using special functions like chdir(), getenv().
4. The program then calls **commandType** function and get the number that indicates the type.
5. A set of if conditions check for the types, and calls the corresponding function to execute the command.