Pranav Angiya Janarthanan NetID: pa446
Dhruvita Patel NetID: dhp123

For this project, we were tasks with creating our own shell. Our shell will have two main modes allowing us to interact with it in specific ways. Interactive Mode allows us to input in any command we like through the Standard Input. Often times, this will end up being a terminal prompt. When entering a command, we will be able to see if the command produces an error by noticing the ! right next to the prompt.

In order to achieve this interface, we had to properly format our initial data structure to hold all the information. Our group had chosen to parse the string using a double Char pointer, or a variable string array. Eventually, information like the path and command was stored in a struct, allowing easy convenience and debugging to occur. We have two structs, one to represent the main/front prompt that is typed and the second struct to represent information about the pipe. This allowed us to separate the commands and perform actions like exec much easier. The struct contains the following Parameters:

```
struct commandInfo{
    char* path;                 //The path to the file
    char* command;              //The first token will always be the command to run
    char** arguments;           //List of all arguments stored in each element of a char array,
                                the last value of the array will be set to NULL
    int numOfArguments;         //Number of arguments stored inside the array
    char* inputRedirection;     //File that will be used for input redirection. There can't be
                                    multiple IO redirections
    char* outputRedirection;    //File that will be used for output redirection. There can't be
                                    multiple IO redirections.       NO OUTPUT
                                REDIRECTION IF THERES PIPE
    int isPipe;                 //Boolean to check if it has a pipe
    struct outputPipeCommandInfo* pipeOutput;       //Pointer in the event we have a
                                                        output Pipe
};

struct outputPipeCommandInfo{
    char* path;                 //The path to the file
    char* command;              //The first token will always be the command to run
    char** arguments;           //List of all arguments stored in each element of a char array,
                                    the last value of the array will be set to NULL
    int numOfArguments;         //Number of arguments stored inside the array
    char* inputRedirection;     //File that will be used for input redirection. There can't be
```

```
                                        multiple IO redirections        NO INPUT
                                    REDIRECTION IF THERES PIPE
    char* outputRedirection;    //File that will be used for output redirection. There can't be
                                        multiple IO redirections.
    struct commandInfo* input;      //Pointer in the event we have a output Pipe
};
```

Another major advantage of having a struct is allowing us to ensure there are no “undefined behaviors”. One of the major decisions our team had made was to prevent any overlapping IO redirection. Whether that be through pipes or redirections. If a user accidentally inputs a overlapping IO, they will br prompted with several messages to change the error.

Our team had decided to focus on two primary extensions: Home Directory and Directory Wildcards.

CD+Home Directory

- Once given a directory that a user wants to open, the command CD will find it and change the current working directory to the desired directory.
    - If it does not exist, CD will print an error message
- If the user inputs “~”, the directory will return back to the home directory.
- If the user inputs “~/PATH”, then the directory will return back to home and continues to switch in the PATH directory

One important Note about CD is that to prevent undefined behavior, CD will not have any IO redirections nor will it be used in Pipes. This is to prevent any unintentional behavior and because CD does not input nor output anything. It would make sense to place a command in that position when there are other commands that are more important.

Directory Wildcards:

The implementation for directory wild cards allows us to find a specific directory regardless of how many characters are missing. This extension aids users in “autofilling” long directory names. One very important note about Directory Wildcards and Wildcards in general is it’s preference to choose the top most directory. In the event there are multiple matches, the wildcards will choose the top most directory. The user must make sure to be specific if there are multiple directories that might match the prompt. On the other hand, if there are no matches, the wildcard will be returned back to the user. This could lead to some unintended behaviors if the user doesn’t desire to create files with wildcard asterisks.

PWD

- Gets the current directory of the user and prints it out

General WildCard

- First, there is a check for a “*” in the input, if there is then it calls the myWildCard Function
- Often times, wildcards are used in conjunction with a path, allowing us to parse the path provided previously to search for matching files. If there is no path provided, it will search the current working directory for any such matches.
    - There is a loop to determine where is the * and creates 2 arrays based on it called first(size is determined by a counter of where the * is) and last(size is determined by doing string length-counter-1=lcounter). It also assignments a number (called position) to each which are
        - 3 is in the first position
        - 2 is in the last position
        - 6 is the middle position
    - Then creates an array called match with the size set to current size (which is 1 at the start)
    - Then the directory of the user is open and there is a loop until there are no more files
    - If
        - position=3
            - We check the first array with the dirent-d_name (file names) with the first counter amount
            - If it is a match: If we put that file name in the array match
            - Then we realloc the match array to have a size of currentsize
            - Then currentsize is increased by 1
        - Position =2
            - We check the first array with the dirent-d_name (file names) with the last lcounter amount
            - If it is a match: If we put that file name in the array match
            - Then we realloc the match array to have a size of currentsize
            - Then currentsize is increased by 1
        - Position = 6
            - We check the first array and last with the dirent-d_name (file names) with the first counter amount and the last lcounter amount
            - If it is a match: If we put that file name in the array match

            - Then we realloc the match array to have a size of currentsize
            - Then currentsize is increased by 1
    - First and last array are both freed and also the directory is closed
    - Return Match array
    - Edges Cases
        - Hidden Files
            - We check in position=2 if there is a file that has a . in the first spot
        - Invalid Path
            - Return the match array with the input as the only entry
        - File not existing
            - Return the match array with the input as the only entry

The analysis of tokens is done in this specific order:

- Search for wildcards in the command token and ensure to expand it if possible.
- Initialize and store the command token which will tell the program which command the user desires to run.
- Store the path of desired tokens such as path commands and specifically Bare Names. Having the path stored saves us hassle of opening the path in the future during exec.
- Store all the arguments, along with wildcard expansion for all of them. In addition, look for IO redirects. Store them immediately.
- If you find an addition IO redirect despite storing one, report the error. In addition, look for any Pipes.
- When we find pipes, we ensure it doesn't overlap with IO and we create the struct for it immediately.
- Once in the pipe struct, we repeat all the steps above making sure there are no overlaps at all.

With the struct now populated, we can now begin to execute commands. We will initially execute all commands that do not have pipes. This is to ensure proper output for pipes when we need to execute them. The first command we will check for is exit. This is the most important command to execute. The next two commands we check are the built in commands CD and PWD. Once that is done, we will execute the last bunch of commands which are the ones that have a predefined path. These are the slash commands which are the path themselves and the barenames which we already found the path for. We can simply execute them since we already have the path and the arguments. We can also add IO redirections as necessary as they are simple dupe commands with child processes. We will always be looking out for any erors with

forking, duping, or exec. Any command that has not been recognized yet will guaranteed fail because it does not match any of the types of inputs we look for.

With majority of the non-piping done, the last thing we handle is the piping. In this, the first thing we keep a look for is an exit command once again. This is extremely important this time as it is necessary that if a exit is there in a pipe, we must execute the other command first before we exit the program. Once this has been done, we will only have 3 possible commands to execute: pwd, path commands, and barenames. Since they are all in a easily accessible format inside the struct, it is a very simple fork to ensure that the standard output of the left side will be sent to the standard input of the right side of the fork. This can also have effects with pwd since it will output a path. This can be used in conjunction with the Output side of the fork to pipe functions like revline (commands such as revline have been rested and will work with pwd along with any output redirect). Since pwd doesn't take in any information from the standard input, there will be a check to ensure that the pwd isn't the input command.

Batch mode will interact in the same manner, reading from a file and executing the same line of commands as above. For our Batch mode, we only allow one input file at a time.

To test the code, there were varios input and output tests done to ensure that dup2() functions as expected. A custom function was made to test piping. The file helloworld.c represents the executable file, a.out, which will take in a text from the standard input and output the first word in that sentence. This can also be used in conjunction with pipes and output redirections. Batch mode is helpful to test a large *Batch* of commands, allowing us to have one parent file, listoftests.txt to perform multiple commands. Since Batch mode shares the same code as interactive mode, this ensures that both modes will perform the same without any errors. Some of the commands will work only on my device due to the setup of the directories and how some files are named a certain way. When I cd into specific directories, it will fail on the grader's end. This is to be expected. Other than that, most commands will work as long as they run the command "./mysh listoftests.txt".