Building a Go Shell

By Sydney Berry and Alice Lee

1. Project Goal

Our goal for this project was to write a program that would build a shell in Golang. Our shell includes our own implementation of cd, pwd, mkdir, mv, rename, rm, getpid, setenv, getenv, unset, echo, ls, cat, exit, and kill. Our shell can run other popular commands that we did not implement by calling exec.Command. In addition, our implementation recognizes and handles the linux symbols for piping (|), input redirect (<), output redirect (>) with overwrite capabilities, running a second command if the first command finishes with no errors (&&), and running processes in the background (&). By implementing a basic shell in Golang, we gained a better understanding of linux commands and practiced designing a simple system.

2. Design & Implementation

2.1 Function Tree

The following function tree shows the order in which functions we wrote in our shell.go program call each other. For example, main calls PrintError, which calls execInput. Please note that all external functions from imported packages that are called in the program were omitted from the function tree diagram for clarity.

├── `main()`: Our main function uses a bufio reader object and an infinite for loop to constantly listen for commands the user is typing into the shell. When main receives input, it passes it to PrintError(input). If the input contains "&" at the end of the command, our program executes "go PrintError(input)" to run PrintError(input) in the background and continue executing our main loop. If the input does not end in "&", our program executes "PrintError(input)". Thus, PrintError runs as part of the original process and not in the background.

│ ├── `go PrintError(input)` **OR** `PrintError(input)`: PrintError passes the input to execInput and prints an error message to the terminal if execInput returns in error. We added this function to better implement backgrounding by not having to return the execInput error to a variable in main.

│ │ ├── `execInput(input)`: Our execInput function uses string.Split() and separateSpecialSigns(args) to create an args array that splits the user's input based on the occurrence of spaces and special characters (<,>, |). Our program utilizes a switch statement which compares the first word of the user's input to determine which command to run. For each case, we check to make sure the user provided the correct number of arguments. If they did not, we return an ErrNoPath error to PrintError. Please see section *2.2 Implementing Basic Linux Commands* to see how we implemented each command. If a command has output, execInput inserts each line of output into an array. The output array is then passed to handleOutput. Any errors that occur while executing a command are returned to PrintError. Otherwise, execInput calls checkAnd before returning to PrintError to handle running any additional commands the user included in their input by using "&&".

│ │ │ ├── `separateSpecialSigns(args)`: Our function separateSpecialSigns() takes in args, which is of type []string, representing the original arguments passed in by the user.

Because this original args array passed on as input to separateRedirectSigns() was obtained from separating the command line input by whitespace character only, it may contain strings and special characters all lumped into one entry. To guarantee easier processing of commands in other functions such as checkRedirection() and handleOutput(), this function separateSpecialSigns() is necessary since it refines the original args array. separateSpecialSigns() creates and returns newArgs, which is of type []string, containing arguments separated by opening and closing double quotation marks and special symbols, <, >, |, and =. Some illustrative examples are provided below:

If the user typed `cat file1.txt>file2.txt` as a command line input, the original args is an array of type[] string containing [cat, file1.txt>file2.txt] since the command line input was separated by whitespace characters only. separateSpecialSigns() takes this original args array as input and returns a new args array of type[] string containing [cat, file1.txt, >, file2.txt]. Note how the second entry from the original args array has been separated into three different entries separated by a special character > in the new args array.

If the user typed `setenv hello=“world”` as a command line input, the original args is an array of type[] string containing [setenv, “hello”=“world”] since the command line input was separated by whitespace characters only. separateSpecialSigns() takes this original args array as input and returns a new args array of type[] string containing [setenv, “hello”, =, “world”]. Note how the second entry from the original args array has been separated into three different entries separated by double quotes “” and a special character = in the new args array.

If the user typed `cat file1.txt|echo` as a command line input, the original args is an array of type[] string containing [cat, file1.txt|echo] since the command line input was separated by whitespace characters only. separateSpecialSigns() takes this original args array as input and returns a new args array of type[] string containing [cat, file1.txt, |, echo]. Note how the second entry from the original args array has been separated into three different entries separated by a special character | in the new args array.

│ │ │ ├── `writeToDirectory(args[1], args[2])`: Our writeToDirectory function is only called when executing a mv command. Please see *2.2.4 mv* for more information about writeToDirectory.

│ │ │ ├── `getKeyValue(args)`: Our getKeyValue function is only called when executing a setenv command. Please see *2.2.8 setenv* for more information about getKeyValue.

│ │ │ ├── `checkInputRedirection(args)`: Our checkInputRedirection is only called when executing a cat command. It functions exactly like checkRedirection(args) (which is discussed later in the file tree) except that it checks for input redirection instead of output redirection.

│ │ │ ├── `handleOutput(output, 0` **`OR`** `1, args)`: Our handleOutput function takes 3 arguments: an array of output with each new line as a separate entry, the last args entry used in the previous command, and an args array of arguments from the users input.

If the next argument in the args array after the last one used is “|”, our function pipes the output of the previous command. The output is piped by constructing a new input using for loops and passing the new input to execInput. The new input has the first argument after the pipe symbol in the args array, then all the output from the output array and then all remaining arguments in the args array. If the command's next argument after the pipe is echo, quotation marks are added around the output in the new input.

If the output is not piped, handle output calls checkRedirection, which will return the path to the file to which the output should be redirected or an empty string if the file should not be redirected. If the file to which the output is being redirected already exists, it is removed using os.Remove(redirectOutput). os.Create(redirectOutput) is used to create the file to which the output will be redirected. handleOutput uses a for loop to loop through the output and print its contents to the new file using fmt.Fprintln(newFile, output[i]).

If the output is not printed or redirected, it is printed using a for loop that reads through each entry in the output array and prints it with a new line character.

│ │ │ │ ├── `checkRedirection(args)`: Our function checkRedirection() takes in args, which is of type []string, representing the arguments passed on by the user separated by special characters, double quotation marks, and whitespace characters. checkRedirection() returns a string and an error or nil. checkRedirection() loops through the args array that has been passed as input and makes sure there is only one output redirect sign, >. If there is more than one output redirect sign, it returns an empty string and an error indicating too many redirections. If there is one output redirect sign but no specified path for the output, it returns an empty string and an error indicating an invalid command. If there is one output redirect sign and a specified path for the output, the function returns a string indicating the path of the output file and a nil indicating no errors. If there is no output redirection sign, then checkRedirection an empty string indicating no specified output path and a nil indicating no errors.

│ │ │ ├── `checkAnd(err, 0` **`OR`** `1, args)`: Our checkAnd function is given three arguments: any errors that have been encountered but have not yet been returned to PrintError, the last args entry used in the previous command, and an args array of arguments from the users input. If a non nil error is passed to checkAnd, checkAnd immediately returns that error to be later passed to PrintError. Otherwise, checkAnd checks to see if the next argument in the args array after the last one used is "&&". If not, checkAnd returns nil to execInput which is then returned to PrintError. If so, it constructs a new input with all the arguments currently in args after "&&" with each argument being separated by a space. checkAnd then returns execInput(input2). execInput(input2) will execute the new input and continue to check for any "&&"s until it does not find one and ultimately returns to PrintError.

2.2 Implementing Basic Linux Commands

2.2.1 cd: If the user did not indicate a path, cd was implemented by running os.Chdir("/") to return a user to their home directory. Otherwise, os.Chdir(args[1]) was used to move to the desired directory the user specified.
2.2.2 pwd: pwd was implemented by adding the path returned by os.Getwd() to the output array.
2.2.3 mkdir: mkdir was implemented by calling the function os.Mkdir(args[1], os.ModePerm). args[1] is storing the pathway provided by the user. os.ModePerm gives users rights to list, modify and search files in a directory.
2.2.4 mv: mv was implemented by calling os.Rename(args[1], args[2]) for renaming a file, where args[1] and args[2] are file names provided by the user, and by calling writeToDirectory(args[1], args[2]) for moving a file to a directory, where args[1] is a file name and args[2] is a directory

specified by the user. writeToDirectory(args[1], args[2]) calls os.ReadFile(args[1]), os.Create(args[2]), and fmt.Fprintf() in order to transfer the contents of the original file to the new file at its new location. Our mv switch case then runs os.Remove(args[1]) to remove the original file.
2.2.5 rename: rename was implemented by calling os.Rename(args[1], args[2]) where args[1] and args[2] are file names provided by the user.
2.2.6 rm: rm was implemented by calling os.Remove(args[i]) in a for loop, where args[i] is a file name specified by the user.
2.2.7 getpid: getpid was implemented by adding the integer returned by os.Getpid() to the output array.
2.2.8 setenv: setenv was implemented by calling os.Setenv(pair[0], pair[1]), where pair[0] is a key and pair[1] is a value for the key specified by the user. setenv obtains the pair array by calling getKeyValue. getKeyValue takes the args array and searches each entry in args for an equal sign. When it finds an equal sign, it inserts the characters of the args entry in front of the equal sign into the first entry of pair[]. It then inserts the characters of the args entry after the equal sign into the second entry of pair[].
2.2.9 getenv: getenv was implemented by adding the output returned by os.Getenv(args[1]) to the output array. args[1] is a key provided by the user.
2.2.10 unset: unset was implemented by calling os.Unsetenv(args[1]), where args[1] is a key provided by the user.
2.2.11 echo: echo was implemented by calling strings.SplitN(input, "\"", 3) and then adding the second argument returned to the output array. When calling handleOutput or checkAnd, echo sends strings.Split(split[2], " ") as the args input and 0 as the lastArgs input.
2.2.12 ls: ls was implemented by calling os.ReadDir("./") to get all the files in the current directory of the user, then calling e.Name() to get the names of these files, and then adding the file names to the output array.
2.2.13 cat: cat first checks for input redirection. If there is input redirection, cat uses a bufio object and a for loop to add all the contents of the file provided to the output array. If there is no input redirection, cat loops through all the args provided and adds the contents of each file provided by the user to the output array. cat has some break statements, which are used to handle special characters.
2.2.14 kill: kill was implemented by calling os.FindProcess(os.Getpid()) to get the current process, and then calling process.Kill() to kill the process.
2.2.15 exit: exit was implemented by calling os.Exit(0).
2.2.16 Extending Our Shell Beyond Commands We Implemented: If a command is not found in our switch statements, the user's input is passed to exec.Command(). exec.Command() will execute the command or return an error message stating that it could not recognize the command. We decided to include exec.Command() in our program, so that our shell could still handle common popular commands that we did not implement ourselves.

3. Performance Evaluation

Overall, none of the basic commands we implemented or short tests we ran took a noticeable substantial amount of time to execute. Our commands will take up to linear time based on inputs. All commands will take a linear amount of time to execute based on the number of characters in the original input because seperateSpecialCharacters loops through the entire input. If the command includes “&&”, this process will need to occur for each command executed. mv is dependent on the amount of input in the file that it is moving. echo is dependent on the amount of input it is echoing. ls is dependent on how many files are currently in the active directory. cat is dependent on the amount of input in the files it is reading and outputting.

Overall, our output mirrors what could be expected by running other common shells. These outputs are demonstrated in our demo video. The following are a few differences that the user should be aware of between our shell than some other common shells:

1) None of our commands have flags implemented with them. Thus, flags should not be used when running our shell.
2) If you include quotation marks around your value or key in setenv, the quotation marks will be considered a part of your value.
3) When cat is printed to the terminal, it will add a new line if the original file cat read from did not end with a new line character.
4) Piping moves left to right. Thus if your command is `command 1 | command 2 | command 3`, the output for command 1 will be the input for command 2. The output of command 2 will be the input for command 3. The output for command 3 will be printed to the terminal. (This is the order of operations for piping used in a Linux system.)
5) Output redirection will ALWAYS overwrite a file if it previously existed.
6) Multiple Function Limitations:
    a) You can not use more than one input redirect or more than one output redirect in the same command. However, you can have one of both.
    b) You should not use && **after** piping or redirecting an output.

4. Instructions for Running Our System

1) Enter the following command into your terminal:
   `git clone https://github.com/jiminl-princeton/finalproject`

2) Once you are in the project directory, enter the following command:
   `go run shell.go`

3) You will see that our shell has been opened. It is waiting for your Linux commands. Enter a Linux command and press the enter key for the shell to process the command. You may enter as many commands as you wish until you decide to exit the shell.

4) Exit our shell by running the command `exit` or `kill`.