CPS1012 Operating Systems and Systems Programming 1 Assignment: eggshell

Aidan Cauchi

23/05/2018

Contents

Chapter 1

Introduction

This report describes how the *eggshell* command-line interpreter for Linux was implemented with respect to the specifications provided. The purpose of this report is to prepare the reader with the mindset required to easily understand how the respective features work internally.

This report is split into the following sections:

- Chapter 2 provides a high level overview of how the source code is split among the different C files available and what the purpose of each file is.
- Chapter 3 explains the different features the program has as well as the logic behind the implementation of each feature.
- Chapter 4 lists a number of known issues and limitations of the program.

For implementation and lower level details in the C language, one should refer to the source code supplied with this report.

Chapter 2

Program Structure

The source code is split into the following C files:

main.c:

- main(): Contains the code required to start the shell as well as the linenoise loop.
- commandParser(): Responsible for parsing user input to call the correct methods to execute the command.

internalCmds.c:

- getPath(): Returns the PATH environment variable.
- changeEnvCwd(): Responsible for changing the CWD environment variable with the passed one.
- getPrompt(): Returns the PROMPT evironment variable.
- showAllEnvVars(): Outputs all the environment variables as well as user declared ones.
- loadEnvVars(): Responsible for loading the environment variables at the start of the shell.
- showEnvVar(): Output the value of the specified environment variable on screen(if it exists).
- variableHandler(): Responsible for deciding whether to show the value of an existing user defined variable, to edit an already exisiting one or to create a new one.
- changeEnvVars(): Changes the required environment variable value to the one specified.
- changeExitCode(): Changes the EXITCODE environment variable with the one passed to it.

internalCmds.c:

- sourceCmd(): Contains the code responsible for handling the 'source' command.
- changeDir(): Contains the code responsible for handling the 'chdir' command.
- printCommand(): Contains the code responsible for handling the 'print' command.

externalCmds.c:

- initialiseProcessStacks(): Initialises the runningProcesses and suspendedProcesses stacks.
- killAllChildren(): Responsible for terminating all child processes.
- sigHandler(): Called when a signal is raised. Calls the method required to handle the type of signal raised.
- handle_Child(): Signal handler for when SIGCHLD is raised. Used for removing zombie processes.
- handle_Stop(): Signal handler for when SIGTSTP is raised. Responsible for suspending the process.
- handle_Interrupt(): Signal handler for when SIGINT is raised. Used to terminate the process.
- bgHandler(): Responsible for handling the "bg" command.
- fgHandler(): Responsible for waiting for either a newly executed process to finish or a process that is in the background(bringing it to the foreground).
- checkIfBgProcess(): Checks whether a command has an "&" at the end to decide whether to run the process in the background or not.
- externCommand(): Responsible for forking a child and replacing it with the specified process.

redirectionHandler.c:

- tokenCount(): Returns the number of times a certain character appears in the passed string.
- redirHandler(): Responsible for checking whether the command has any input/output redirection and taking the necessary steps to perform redirection.
- hereStringHandler(): Responsible for performing the necessary steps to handle here strings.

- redirStream(): Replaces either the stdin or stdout file stream with the filename supplied.
- extractFileName(): Returns the filename specified after a redirection operator.
- getLineWithoutFilename(): Returns the line without the redirection operator and filename. Used to get the command needed for execution.
- closeRedir(): Replaces the modified file streams to their default streams(stdin, stdout).
- catenateFileInput(): Used to add the input of a file to the command, when the input redirection operator is used.

piping.c:

- pipeInterface(): Responsible for extracting all the commands that require piping and counting the number of commands.
- pipeCommandHandler(): Responsible for managing the usage of the pipes and passing the commands to the command parser.

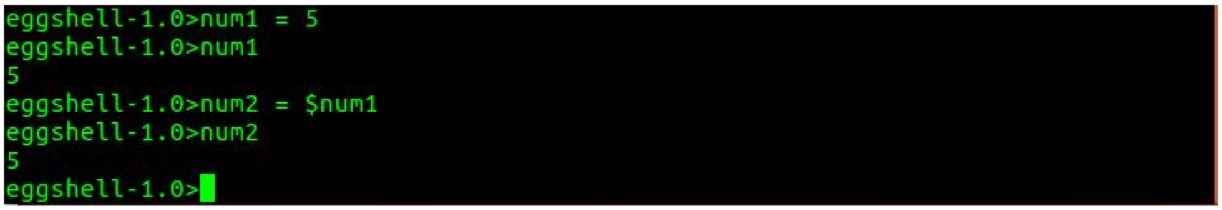
Chapter 3

Program Features

3.1 Environment Variables

The environment variables are loaded before linenoise is called via the loadEnvVars() function. This function takes a pointer to the environment variables list passed when main is called. Upon being called, the function iterates over the list of passed variables until it finds the variables matching the PATH, USER, HOME, PWD(current working directory) and stores them in global variables. Furthermore, the TERMINAL variable is initialised by calling the ttyname(STDIN_FILENO) command. In order to store the SHELL variable, the program just gets the current directory when the shell is started. Finally, the EXITCODE is initialised to -1.

Apart from this, the shell has the ability to store user declared variables. This can be done with the following format: **VARNAME = VALUE**. This same format can be used to edit already existing variables. Moreover one can declare a variable with the value of another variable by using this format: **VARNAME = $EXISTINGVARNAME**.



```
eggshell-1.0>num1 = 5
eggshell-1.0>num1
5
eggshell-1.0>num2 = $num1
eggshell-1.0>num2
5
eggshell-1.0>
```

Figure 3.1: Variable assignment in action

3.2 Command Parser

The commandParser() method takes two parameters: the command text and a pointer to a dynamic array where the user declared variables are stored. After being called, the method proceeds to make a copy of the command since strtok() modifies the original string. Thus, to work around this, the copy of the command is fed to strtok() and tokenised using a white space. The token should now be the type of command the user wants to execute. The command parser then goes through a chain comparing the token to each type of internal command offered by the shell. Should it find a match, the method to handle that type of command is called. However, if no internal commands are found, the external command handler method is called to check whether the command is a type of external command. If this fails, then that means that the command is either an assignment of a variable or a call to display the value of a variable (the shell is made in such a way that if only a variable name is entered, its value is printed out assuming it exists). As a result, the original command is copied again (since the initial copy was modified by strtok) and tokenised with '=' as a delimiter. If the token and the command are not the same after tokenising, that means that an '=' was found, leading to the assumption that the command is an assignment. The variable name and the value are then extracted and sent to the variableHandler() method, which takes care to add the new variable to the dynamic array or to update an existing variable. However, if token and the command are the same after tokenising then no '=' was found, which either means that the user wants to display the value of a variable or it is an invalid command. Finally, the variable's value is shown otherwise an error is displayed to the user.

```c
if (strncmp("print", token, stringSize) == 0)
    printCommand(line, vars);

else if (strncmp("chdir", token, stringSize) == 0)
    changeDir(token);

    ...
```

Listing 3.1: Snippet of how the parser recognizes the commands

NOTE: It is incredibly important to ensure that there is a white space between every argument and/or operator as the command parser is built around tokenising white spaces.

3.3 Internal Commands

3.3.1 exit

All the exit command does is it first kills all the processes spawned by the shell (if there are any) and then proceeds to close the shell by calling exit();

3.3.2 print

The print command tokenises the argument using whitespace as a delimiter. After this, it checks the first character. If the first character is a '$' that means that the letters after the '$' are referring to a variable name. As a result, the function extracts the name and shows the value of the variable in the output instead of its name (assuming it exists). On the other hand, if the first character is a ' " ' that means that the word is the start of a string literal. Should this be the case, the function then tokenises the string using ' " ' as a delimiter to get the end of the string literal. It then outputs the string literal exactly as it is (ignoring any calls to variables inside the literal). Finally, if the first character is neither of these cases, then just output the word as it is. This process is repeated until the end of the string is reached.

3.3.3 chdir

The chdir command starts by checking whether the argument is equal to ".." which means that the user wants to go one directory level up. If it is, the function tokenises the CWD environment variable using "/" to get the directory levels. Each level is stored inside an array of strings until the end is reached. The new directory is then formed by catenating the elements of the array with the exception of the last one (since the user wants to go one level up) and stored in a temporary string. However, if the argument is not "..", it is either a completely new directory or a folder inside the current directory. If the argument starts with a "/" that means that the user is specifying a different directory (thus store the entire argument as the new directory). If none of these cases are met, then the argument is catenated with the CWD environment variable to get the next level. Once the new directory is determined, this is sent as an argument to chdir() (from *unistd.h*) to determine whether it is a valid directory. If it is, then the current working directory is updated and the environment variable CWD is updated to reflect this change. Otherwise an error is printed out.

3.3.4 all

The all command simply shows all the previously mentioned environment variables, including user specified ones, on screen.

3.3.5 source

The source command takes the file name and type (ex. test.txt) as an argument. All it does is read the file line by line until the end of file is reached. Each line read is sent to the command parser to execute the command.

3.4 External Commands

The externCommand() starts by parsing the line to get the command without any arguments, catenates it to a "/" and stores it as the first argument (since execv uses a path as input). After this, the function stores all arguments supplied to the command in an array of strings. The function then proceeds to fork a child. Inside the child, the function starts parsing the PATH environment variable, delimited by ":". Following this, the first argument is catenated with the current tokenised part of the PATH and is then sent to the execv function along with the arguments. If execv fails, the function tokenises the PATH environment variable again and repeats the process until either the child is replaced with the new process or there are no more paths left to tokenise.

```c
while(token != NULL)
{
  strcpy(tokenCopy, token);
  strcat(tokenCopy, firstArgWithSlash);
  //no need to check if execv is < 0 since it has to iterate multiple time
  execv(tokenCopy, argList);
  token = strtok(NULL, ":");
}
```

Listing 3.2: Snippet of how the PATH is parsed to check for external commands

The **token** variable holds the the first path delimited by a ":". The command is then catenated with the tokenised path and passed to execv along with the arguments. If execv fails, the path is tokenised again until the end is reached.

3.5 Input and Output Redirection

Before the shell executes the command, it calls the redirHandler() function that parses the command to check whether the output/input requires any redirection.

3.5.1 Output Redirection

To check whether the command has any output redirection, the function tokenises the command using ">". If a ">" is found, the function then checks how many times this token appears in the command text. If it appears only once, the function opens/creates the file specified after the token in "write" mode thus replacing any text that might have been there before. If the token appears twice, then the file is opened in "append" mode meaning that the output will be added to the contents already in the file (unless it's blank, then it acts exactly like the "write" mode). After opening the file, the function proceeds to replace the stdout file descriptor with the newly opened file so that all output automatically goes to the file(using dup2). Finally, it executes the command and resets the stdout file descriptor before accepting the next command.

3.5.2 Input Redirection

To check whether the command takes input from another file, the function tokenises the command using "<". If a "<" is found, then the number of times it occurs in the command is noted once again. If only one instance is found, the function replaces the stdin file descriptor with the newly opened file thus reading from the file. However if three instances are found, then the process is a bit different. The here string is stored in a temporary file on disk and the stdin of the program is replaced with this file. Once this is read, the newly opened file descriptor is replaced again with stdin and the temporary file is removed.

3.6 Piping

When a command is input, the program checks whether any piping is necessary before actually executing the command. The command gets sent to the pipeInterface() method where it is tokenised using " | ". If such a character is found, the string is tokenised further and each command is stored inside an array(between each pipe there should be a command). This is done to calculate the number of pipes that are needed. After this is done, the pipeCommandHandler() method is called. Firstly, the number of pipes to be used are opened. This is calculated depending on the number of commands. If there are n commands then there must be n-1 pipes. Once the pipes are opened, the program goes into a loop until all commands in the array have been executed. As the commands are executed, the loop makes sure that each command is reading/writing from/to the correct pipe. It uses two simple formulae to calculate which pipe is going to be used next:

$$pipeIndexToReadFrom = (iterationNumber - 1) * 2$$

$$pipeIndexToWriteTo = (iterationNumber * 2) + 1$$

All this is a fancy way of saying that the commands will read from the even indices while they will write to the odd indices. However, it is important to note that the command reads from the pipe preceding the one being used (hence $iterationNumber - 1$). When the pipes are determined, the necessary file desciptors are replaced with the pipes at the indices indicated by the formulae and the command is passed on to the command parser for execution. It is important to note that there are two special cases when this does not apply: when it is the first iteration and when it is the last one. On the first iteration, the loop reads from **stdin** not from a pipe and on the last iteration it writes to **stdout** not to a pipe.

3.7 Process Management

The program makes use of 2 stacks during its execution: the runningProcesses stack and the suspendedProcesses stack. Whenever an external command is executed, its PID is pushed on the runningProcesses stack until it finishes. The suspendedProcesses stack is used to hold the PIDs of (you guessed it) suspended processes. These stacks are beneficial as they allow the shell to forward signals towards its child processes.

3.7.1 Interrupting and Suspending Processes

In order to be able to terminate or suspend processes, the program assigns two signal handlers at the beginning that are dedicated to catching the **SIGINT** and **SIGTSTP** signals which are generated by pressing **CTRL + C** and **CTRL + Z** respectively. When a **SIGINT** signal is received, the program pops from the runningProcesses stack and sends a **SIGTERM** to terminate the child process. Should termination fail, a **SIGKILL** signal is sent to forcefully kill the process. On the other hand, when a **SIGTSTP** signal is received, the PID is retrieved from the runningProcesses stack and a **SIGSTOP** is sent to the retrieved PID. The PID is then pushed on the suspendedProcesses stack. It is important to know that the signals are handled in the shell process not in the child. The signals in the child are ignored so as to keep the parent responsible for managing the signals.

3.7.2 Resuming Processes

If the user types "fg", the program will first pop from the runningProcesses stack to get the first PID. If no runningProcesses are found, it will pop from the suspendedProcesses stack. Since the process is to be run in the foreground, the shell simply waits for the process to finish. If the process initially was in the suspendedProcesses stack then a **SIGCONT** is sent to it to resume the suspended process before waiting for it. On the other hand, if the user types "bg", the same procedure is repeated with the only difference being that the shell does not wait for the process to terminate, thus allowing the shell to be used simultaneously with the executing processes.

3.7.3 Running Processes In The Background

Before a child is executed, the command is checked to see whether there is an "&" at the end. Should this be the case, this means that the process is to be run initially in the background. To achieve this, the child simply executes the process and the parent pushes the child's PID on the runningProcesses stack, without waiting for the child to finish(to allow use of the shell while the child is executing).

Chapter 4

Issues and Limitations

4.1 Piping and Redirection

The program does not allow piping and redirection to be combined in one single command. Since the program checks for pipes before checking for output/input redirection, this will cause problems if it finds both cases inside a single command.

Apart from this, input redirection is quite limited in terms of the size of files it can read, as it ends up reading the file and then crashes.

4.2 source Command

The 'source' command does not allow redirection or the piping of commands. Instead the operators are treated as normal arguments, which might cause some problems.

4.3 Process Management

It seems that the program does not always respond to commands like "fg" and "bg". When "fg" is received, the program waits for the first program at the top of the runningProcesses stack for the majority of cases but sometimes the program keeps executing without waiting for the process, which in turn causes problems when it comes to keeping track of processes. However all this seems quite random as it sometimes works as expected and sometimes it doesn't.

4.4 Here Strings

It is known that here strings are not fully functional as they were not understood correctly. Here string commands might work with certain commands but not with all commands and they also don't have the functionality to handle variable names.

Chapter 5

Conclusion

In conclusion, the majority of the features work as specified in the assignment. While the assignment was quite a challenge, it was a great primer to programming in Linux and manipulating strings.