CS167 Project 1: Shell

Zhang, Shu (szhang)

February 4, 2013

Contents

1 General Overview **2**

2 Code Brief Explanation **2**

3 Command Processing and Parsing **3**

4 Conclusion **3**

1 General Overview

My Shell program can simulate a large part of work that a read shell (like bash) can do, and runs in a stable status. The main features my shell program provides are:

- Supporting for a set of built-in commands.
- Being able to check for any direction and relatex syntax fails.
- Won't crash for Ctrl-C, Ctrl-Z signals, but will be turned down when user typing Ctrl-D without any other chars.
- Zero memory leak.

2 Code Brief Explanation

The code could be analyzed from the main() function. First It will check the args, and print them out to STDOUT, but the args won't be used anymore. Then, there are 4 parts of string buffers which are allocated with MAX_LENGTH space. These string buffers are global so that all functions could visit them easily. MAX_LENGTH is set to 512 bytes.'commandline' is a buffer to store the whole user inputs in a line. 'command' is to store the parsed command part with arguments. 'input' is to store the path of the redirected input file path. For the output path, I used a linked list to store all output files. However, only the last added file will be used for output redirection. The reason for this design is that I observed that even user types in multiple redirected output file path, the bash will not crash, so this observation let me to use the linked list to store these outputs all together without printing an syntax error. *refresh_buffers()* function is used to clean up all buffers by setting zeros to the heap spaces or cleaning all nodes in the linked list. *refresh_buffers()* is called at the beginning of every loop in order to ensure that user inputs won't interfere with other times' user inputs. As always appeared in pair, *free_all_resources()* is used to free all buffers before exiting the program.

The major part of the program is the *while(1)* loop in the main function. The workflow is like this: first it will register signal handlers in case user input CTRL-C/CTRL-Z which might otherwise stop the program. Then, the shell reads user input from standard input, and first check if there is 0 byte read, if yes, which means user only inputs CTRL-D, then the program will exit in response for this operation. Then, it will check whether the last char read is '
n', which is 10 in ASCII code. If no, which means that the user input CTRL-D in the end of the string, the program will ignore the user input this time.

If everything goes well, then the main loop goes to the core function of the program - *process_commandline()*. This function deals with the user input and returns a result code. If the result code is not zero, then the program calls its *error_handler()* to print out error messges.All error codes are recorded in a global string array whose indeice are corresponding to the error codes. At the end of the loop, the buffers are refreshed and starts another loop.

3 Command Processing and Parsing

This section describes the meat inside the *process_commandline()*. It first calls *eliminate_dup_tab_space()* which shrinks all spaces or tabs into one, and eliminate all spaces and tabs in the start and end of the commandline input string. This pre-processing will make string parsing much easier. Then it calls a subfunction called *parse_commandline*. Inside this function, it calls *split_to_parts()* to parse the command line and fill global string buffers (commandline, command, input, output string linked list). So, the core parsing function is actually *split_to_parts()* function. The basic idea of the parsing is to detect special chars like spaces, or redirection symbols. If encountered, it will see whether to put them into corresponding buffers or whether they have been appeared for multiple times. If there are multiple input redirection symbols detectedm, the function will repord error. (Since *strtok()* is forbidden, I think the splitting processing a little bit not elegant, but it works!).

After having splitting all parts, the *process_commandline()* function inspect the command buffer just parsed out. It calls *process_builtin* function to see whether the command includes the builtin command supported by the shell. If the command is built-in, then it returns zero and the *process_builtin* returns. If the function returns NOT_BUILTIN, which means that the command calls a another program, the function enters the block which is used to parse the command buffer and fork a child process to execute. The code block will extract all args in the command. Then *fork()* is called . In the child process block, it will try opening the input and output redirection files by closing STDOUT and STDIN before if they are specified by the user, if everything goes will, the last step is to call the exact child process.

4 Conclusion

So the major parts of the program are explained in this REAME. If you are not clear about details, you can read the comments I wrote in the code, personally I think they can explain 99% of your puzzles.