A Student Approach to Implementing a Shell

Benjamin Trans

bvt2nc

4/16/18

The assignment was completed!

**Problem Description**

Implement a UNIX shell from scratch that meets the requirements of the boss. The shell does not need to support job management, environmental variables, or most other standard shell features. The shell must be able to run programs and support file redirection and pipes. The input to the shell is a sequence of lines where each line must have no more than 100 characters. Characters must be of the following set: {A-Z, a-z, 0-9, -, ., /, \_, >, <, |}. Every command except “exit” should be interpreted as a UNIX executable. If “exit” or an end-of-file is encountered, the shell shall terminate. Commands beginning with the character ‘/’ are absolute and should be executed as is. All other commands are relative and should be executed from the current working directory.

**Approach**

Assignment was completed in C. Code to handle shell input was first developed using fgets() to read from stdin indefinitely until the “exit” command or end of file is reached. It was ensured that the number of characters read from each line was limited to 100 characters. If there are more than 100 characters in a line, an error is printed and the shell continues to the next line.

After a line is read in and validated to contain only valid characters as specified by the problem description, the line is parsed for commands and operators. Operators consist of the set {><|}. Commands are strings between operators, other operators, the beginning of the line, and the end of the line. Commands are stored in a data structure with the first command found at the beginning and the last found command found at the end. The same goes for operators.

After all commands and operators and read in, and check is made for base cases. The bases cases include a set of operators of length 0 or if the number of operators is equal to or greater than the number of commands. If there are no operators in the line, then the line consists of a single command and the command is executed. If the number of operators is equal to or greater than the number of commands, an error is printed and the shell reads the next line. This is because a valid executable line must contain more commands than operators as a line cannot begin or end with an operator and a command must follow before and after an operator.

Once the base cases are checked, the operators are iterated through with a pointer placed at the second command. The first command cannot contain a namespace for a file redirection and thus, must always be executed. The command pointer thus points to the next command which must either be executed or contains the namespace for a file redirection. A second pointer called the execute pointer originally points to the first command and is used to execute the command if the correct operator is encountered.

While iterating through the operators, these are the following rules for each type of operator encountered:

1. If the ‘>’ operator is encountered, stdout should be redirected to the namespace pointed to by the command pointer. The file is created if it does not exist, opened, and cleaned. Stdout is redirected to the file and the command pointer is incremented.
2. If the ‘<’ operator is encountered, stdin should be redirected to the namespace pointed to by the command pointer. If the file does not exist, an error message is printed out and the next line is parsed. If the file does exist, the file is opened in read only mode, stdin is redirected to the file and the command pointer is incremented.
3. If the ‘|’ operator is encountered, a pipe is created and a child process is spawned using fork(). Stdout is redirected to the write end of the pipe and the command pointed to by the execute pointer is executed with the exit status communicated to the parent process. The parent process waits for the child process to finish and checks the exit status. If the exit status is nonzero, the child process failed to execute the command and returns. If the exit status is zero, then the command was executed successfully and the command with its exit status is printed to stdout. Stdin is redirected to the read end of the pipe, the execute pointer points to the command pointed to by the command pointer and the command pointer is then incremented.

After all operators are iterated through, if the last command pointed to by execute pointer is in scope, the command is executed.

The following functions were developed for this assignment:

* **getcmd()** – This function reads a line from stdin and copies the line to the buffer. The line is checked to have only 100 characters. If the line contains more than 100 characters, an error is printed and the function signals for the shell to read the next line.
* **checkline()** – This function checks the buffer passed to ensure that only valid characters appear in the buffer. If an invalid character is found, the invalid character is printed and the function signals for the shell to read the next line.
* **parseline() –** This function parses the line for all commands and operators. The base cases are checked and the operators are iterated through following the rules described above. If a command is unable to execute, the shell stops parsing the line and reads the next line.
* **execcmd() –** This function executes the command passed. The arguments are extracted by splitting the white space. If arg[0] is absolute, the command is executed as is. If not, then the following file paths are prepended and execution is attempted: {“”, “/bin/”, “/usr/bin/”}. If a command does not execute, a signal is sent back that the command was unable to execute.
* **trimwhitespace() –** This function removes all leading and trailing whitespace and new line character from the passed character buffer.

**Analysis (Lessons learned)**

**I.**

Execution of a command is always done in a child process to ensure that the main process does not exit. This also allows for the exit code of the executed command to be communicated back to the parent process to ensure proper execution. If a command was successfully executed (signaled by an exit code of zero), the command and its exit code of zero is printed out to stdout. If the command was not successfully executed (signaled by a nonzero exit code), it is printed that the command failed to execute.

**II.**

A new pipe is created when a new operator is iterated regardless of if a pipe is needed or not. This pipe is only used if the operator is the ‘|’ operator. If the ‘|’ operator is encountered, a child process is spawned. The child process redirects stdout to the write end of the pipe so that the output may be saved to be used as stdin for another command. Both ends of the pipe must be closed before execution otherwise execution of the command hangs as it waits for the ends of the pipes to close. The parent process redirects stdin to the read end of the pipe which contains the results of the execution of the command done in the child process and iteration continues.

**III.**

If stdout is redirected, output must be redirected back to stdout if you chose to print to stdout. After file redirection when the operator ‘>’ is found, it was discovered that printing to stdout resulted in the output being redirected to the file descriptor pointed to by the ‘>’ operator. This was resolved by saving stdout before it is redirected and redirecting stdout to this saved copy.

**Results**

The shell is able to function to the specifications of the problem description and assignment handout. A shell prompt “>” is printed out and appropriate execution of legal commands results in the output of the command followed by the exit code of the command. Execution of illegal commands results in a message stating that the command failed to execute and the shell moves on to the next line.

Testing was done manually and through a testing script to ensure that the output of each command matches that of a UNIX shell with the addition of the exit code of each command. Multiple operators were tested in a single line along with invalid commands at the beginning, middle, and end of the line to ensure proper output. Invalid output was tested to ensure that the commands do not execute and parsing of the line stops.

**Conclusion**

The assignment was successfully completed. All communication between processes were accomplished with the use of pipes and execution of all commands resembled that of a UNIX shell. A few design choices to accomplish the problem description are described in the analysis section which consists of the lessons learned throughout the completion of this assignment. Adequate knowledge in the usage of pipes was essential to this shell assignment as described under Analysis Section II. Future work should look to implement additional UNIX shell features and reduce redundancy in the code for the design choice described under Analysis Section I as the code block pertaining to it could be implemented in a function and called each time execution of a command is needed.