Brett Crawford

October 8, 2014

Project 2: Developing a Linux Shell

Test Procedures and Results

In this document I will discuss the procedures used for testing the command line interpreter that I have created and results of the tests that were performed. You will find the documentation relating to the functionality of the command line interpreter and its functions contained within the source code of the project.

The goal of this project is to create a simple command line interpreter (shell) which is capable of parsing a user input into commands and arguments and launching those commands using the fork(), exec(), and wait() system calls. The shell will also provide functionality for the following operators: ‘>’ (output redirection), ‘>>’ (appended output redirection), ‘<’ (input redirection), ‘|’ (pipe), and ‘&’ (concurrent execution). I have combined problems A, B, and C (described in the project specification PDF) into a single, working program.

In this program, I have made use of a library of string functions (bc\_strlib) that I had written as part of a previous course’s assignment (CIS 2107). These functions were tested using a test driver program, which used hard coded strings to test and verify the correctness of their functionality. The output results were compared with the results that I calculated by hand and edge cases, such as NULL input, were tested as well.

For the portion of this shell that reads and parses the user’s input, I used print statements in various locations in my code to ensure the input was being parsed and stored properly. When user provides input to the command prompt, the input is first parsed into commands (separated by ‘|’) and the parsed commands are parsed once again into arguments (separated by ‘ ‘). When the parsing was finished, each command was displayed, argument-by-argument, to the screen so I could verify that the input was being properly handled.

Once the input is parsed into commands and arguments, each command is checked for operators (excluding ‘|’ which was already accounted for). Each command is associated with flags to indicate if the command used redirection or concurrent execution. Once again, I used print statements to verify that correct flags were being set when a command made use of the operators. It is also worth noting that when pipes are used to connect multiple commands, that only the first command can use input redirection and only the last commands can use output redirection.

Once the commands, arguments, and operators are accounted for, the program forks a child process (child handler) to handle the creation of each subsequent child associated with a command. When the child handler is finished creating all of the necessary children, the child handler exits. At this point, all of the commands have been handled and the shell resumes and returns to the prompt. To verify the children handler creating and handling children properly, I used print statements using process IDs to identify when each child was created and what command is was executing.

The following commands were used to test the correctness of the shell. Each command was first tested with the bash shell in Ubuntu to provide a baseline for comparison. The test commands and results are as follows:

bc\_shell-> echo hello

hello

bc\_shell-> ls

bc\_libstr bc\_shell.o makefile out.txt sources.mk

bc\_shell.d bc\_strlib objects.mk Project2-bc\_shell subdir.mk

bc\_shell-> boguscmd

-bc\_shell: boguscmd: command not found

bc\_shell-> echo hello there > out.txt

bc\_shell-> cat out.txt

hello there

bc\_shell-> echo i'm on line two >> out.txt

bc\_shell-> cat out.txt

hello there

i'm on line two

bc\_shell-> wc -m < out.txt

28

bc\_shell-> wc -m < out

bc\_shell.c: main(): error opening out: No such file or directory

bc\_shell-> echo hello there | wc -m

12

bc\_shell-> wc -m < out.txt | wc -l > out2.txt

bc\_shell-> cat out2.txt

1

bc\_shell-> sleep 2

bc\_shell-> sleep 2 &

All of the commands entered into the shell responded with the expected output that was confirmed by previous testing using the bash shell. In the case of an unknown command, an error is returned to the user. In the case of an input redirection using an unknown file, an error is returned to the user. In both cases, the shell resumes functioning after alerting the user of the error.

I did come across one particular sequence of inputs that cause the program to crash:

bc\_shell-> echo hello there > out.txt

bc\_shell-> echo line two >> out.txt

\*\*\* Error in `./bc\_shell': free(): invalid next size (fast): 0x00000000011cd2f0 \*\*\*

Aborted (core dumped)

For reasons unknown to me(at this point) the command that appends the particular text “line two” to an existing file causes an error with freeing a dynamically allocated string used within the parse\_input() procedure which in turn causes the program to crash. The append operator did not cause any issues with many other inputs that I tested. For example:

bc\_shell-> echo hello there > out.txt

bc\_shell-> echo this seems to work >> out.txt

bc\_shell-> cat out.txt

hello there

this seems to work

Unfortunately, I did not have the time needed to fully understand this problem and fix it.

The shell that I have created should handle all the requirements of this assignment, however it is still lacking many of the features of a fully functional shell. In this implementation, the shell does not contain the following features:

* change directory command
* sufficient error handling for all possible inputs
* machine name or current working directory within prompt
* many other common shell features

This project required many hours to complete and as much as I would have liked to include some of these features, it was impractical to spend additional time on features that were not a direct requirement for the assignment. Please refer to the source code for any additional information concerning the command line interpreter.