Daryl Stronge
Github: DSNoNintendo

ID: 917844673
CSC415.01 Operating Systems

Assignment 3 – Simple Shell

Description:
This assignment is to write a Simple shell to emulate a Linux command line with pipe support.

Approach:
My approach started by learning about *fork()*, *execvp(),* and threads. After this I was able to set up code capable of executing commands, but it only ran one time and didn't accept user input.

After learning about *fork()*, I needed to build out the prompt loop. Through learning more about *fgets()* and strtok(), I was able to request and parse input from the user. *fgets()* was used to query user input and store the specified max number of characters in the command to a buffer. *strtok()* was used to tokenize the user input, word-by-word, in a loop. Each word that wasn't a pipe symbol (|) was added to an array. Once the token loop was complete, the array would be passed to *execvp()*, executing the command. Placing this code inside of an infinite while-loop ensured that the program wouldn't stop executing unless an error occurred or the user typed "exit."

After setting up the Prompt loop for traditional commands, I needed to set up pipe support to allow users to enter multiple commands in a single line. First, I needed to figure out a way to execute multiple arguments. The program was initially set up to handle one argument, stored in an array. In an instance, where a user piped multiple commands, I would need to store a command array for each command to a second array that accepted character pointer arrays, and allocate the memory as needed. Each time a pipe was detected from user input, a new command array would be appended to the second array.

After this, I needed to learn about *pipe()* and file descriptors. I was able to set up a test script that accepted piped commands, but the test script only worked in instances where a single pipe was present in the user input. In order to create a loop that handled multiple pipes, I needed to learn more about the *pipe()* command, and attempt to visualize the problem.

```
Multiple-pipe Example:
cat Makefile | wc | wc -l
             ^    ^
```

In the above example, there are two pipes and 3 individual commands. Each pipe requires two file descriptors; one for reading and one for writing. I created a nested array that stored an array of 2 file descriptors for every pipe in a command. In the context of this example, two arrays were stored in the nested array, each containing two file descriptors.

After this, I wrote a for-loop to iterate for every command present in the command array. My logic was as follows:

At point *i* in the for-loop array, if there is a command at point *i+1*, the output of the current command should be redirected to the pipe at point *i*. In the context of the provided example, on the first iteration (*i*=0) the output of `cat Makefile` would be redirected to the pipe at index 0. On the second iteration (*i*=1), `wc` would execute using the output of the pipe at index 0, which is the pipe between the current command and the last. The output of this command would then be redirected to the pipe at index 1, the pipe between the current command and the next.

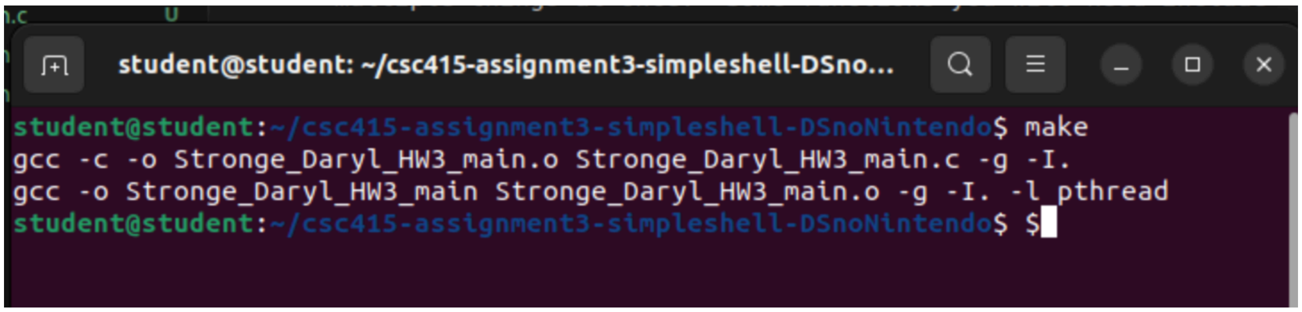
The previously state logic only occurs when there is a command present after the command being executed at the current iteration. If there is none, the command executes normally. In the final iteration (*i*=2), `wc -l` executes using the output from the pipe at index 1.

Issues and Resolutions:

My first issue was figuring out how to properly tokenize strings. When user input was given, I initially didn't know I would have to account for the *n* escape character from users pressing the return key. After accounting for this I was able to check user input as necessary.

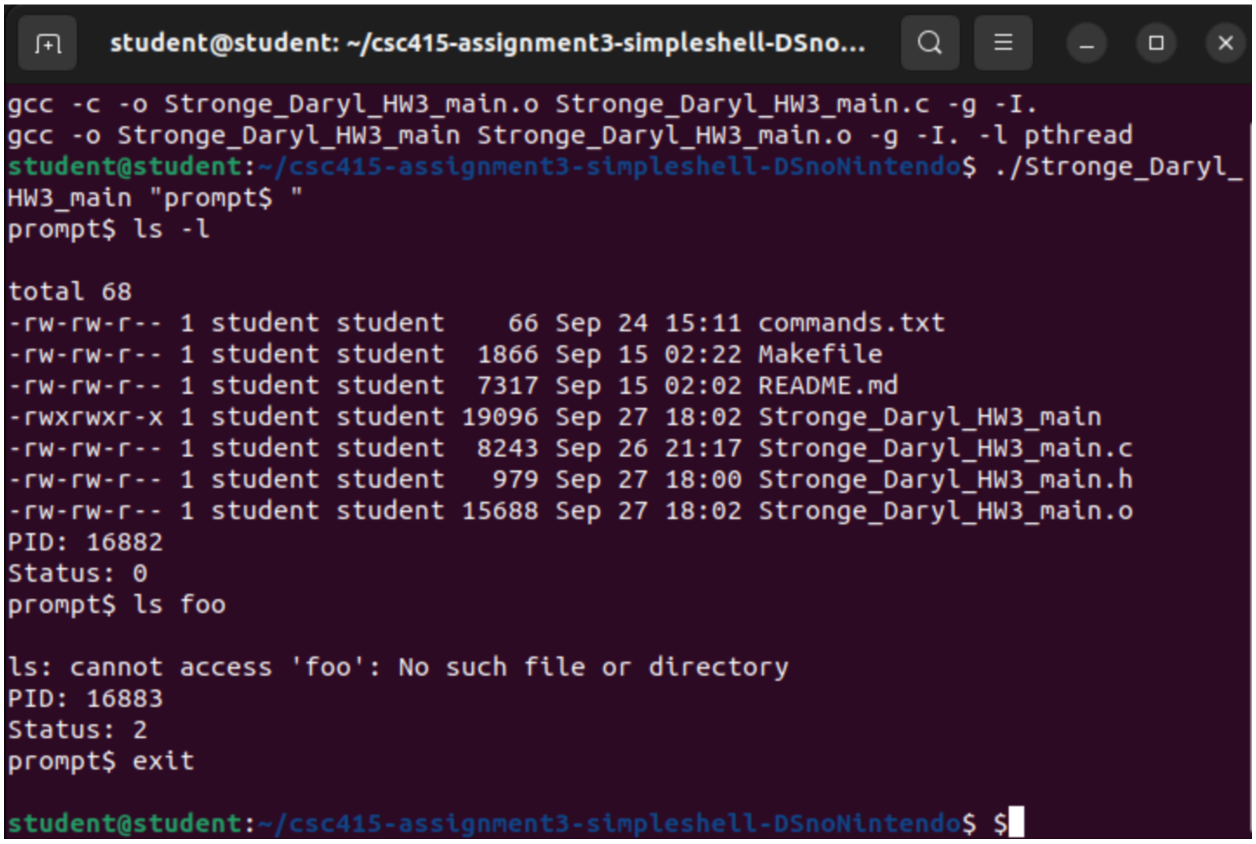
My next issue was figuring out an issue in which after entering a command, the program would skip requesting user input in an infinite loop. I was able to resolve this issue by checking the first string token before running user input through the if-else statement. If the string token was null it would be set to a null terminator "\0" which would be caught in the if-else statement, ending the program gracefully.

Screen shot of compilation:



```
student@student:~/csc415-assignment3-simpleshell-DSnoNintendo$ make
gcc -c -o Stronge_Daryl_HW3_main.o Stronge_Daryl_HW3_main.c -g -I.
gcc -o Stronge_Daryl_HW3_main Stronge_Daryl_HW3_main.o -g -I. -l pthread
student@student:~/csc415-assignment3-simpleshell-DSnoNintendo$ $
```

Screen shot(s) of the execution of the program:



```
gcc -c -o Stronge_Daryl_HW3_main.o Stronge_Daryl_HW3_main.c -g -I.
gcc -o Stronge_Daryl_HW3_main Stronge_Daryl_HW3_main.o -g -I. -l pthread
student@student:~/csc415-assignment3-simpleshell-DSnoNintendo$ ./Stronge_Daryl_
HW3_main "prompt$ "
prompt$ ls -l

total 68
-rw-rw-r-- 1 student student    66 Sep 24 15:11 commands.txt
-rw-rw-r-- 1 student student  1866 Sep 15 02:22 Makefile
-rw-rw-r-- 1 student student  7317 Sep 15 02:02 README.md
-rwxrwxr-x 1 student student 19096 Sep 27 18:02 Stronge_Daryl_HW3_main
-rw-rw-r-- 1 student student  8243 Sep 26 21:17 Stronge_Daryl_HW3_main.c
-rw-rw-r-- 1 student student   979 Sep 27 18:00 Stronge_Daryl_HW3_main.h
-rw-rw-r-- 1 student student 15688 Sep 27 18:02 Stronge_Daryl_HW3_main.o
PID: 16882
Status: 0
prompt$ ls foo

ls: cannot access 'foo': No such file or directory
PID: 16883
Status: 2
prompt$ exit

student@student:~/csc415-assignment3-simpleshell-DSnoNintendo$ $
```