Linux Shell Documentation

Objective:

A C program that will act as a shell command line interpreter for a Linux/Unix kernel. The user is able to type a command followed by parameters. The program then searches the directory system and executes the commands with the optional parameters passed.

Functions:

The program has the following functions:

- main()
- parse (char *userInput, int forking)
- **complie_argv (char *userInput)
- execute (char *userInput)
- execute_gt (char *userInputer)
- execute_lt (char *userInputer)
- execute_pipe (char *userInputer)
- **str_split (char *a_str, const char a_delim)
- *trimwhitespace (char *str)

main():

The mock shell starts with the main method. The main method runs as an infinite loop, and can only break when the user types 'exit' or ctrl + d.

One the user enters his command, it is passed to the parsed method for processing.

This method also has a local variable 'forking.' This variable is set if the command entered has a '&' at the end of it. If a '&' is detected, this method will strip it off and replace it with the null terminating '\0\ symbol. It will also set forking to 1 before it is passed to parse.

parse(char *userInputer, int forking):

The main purpose for the parse method is to determine which execute method to call.

First, the method checks that the input does end with the '\0' symbol instead of the '\n' symbol.

After that check is complete, it then parses the input from the user for a '<', '>', or '|'. There are corresponding execute methods for each.

If the main method detected a '&' in the command, it will have set forking to 1 when passed. If this is the case, the method will call the 'execvp' method directly, without the 'wait' method behind it.

complie_argv (char *userInput):

This method simple builds the argv array that is needed to pass to execvp. It parses through the user's input command and complies the argv array and allocates the correct amount of memory dynamically.

The argv array is then returned.

execute(char *userInput):

When a basic command without any special characters is entered, execute is the method called. It uses complie_argv, and calls the execvp() method to carry out the command.

execute_lt(char *userInput):

Called when a '<' is in the user's command. It uses the str_split method is called to create an array that is split using '<' as a delimiter. It also trims any leading and trailing whitespace that occurs. Finally, it sets the correct file permissions to newstdin.

After that special case is handled, the complie_argv and execvp methods are called to carry out the commands.

execute_gt(char *userInput):

Called when a '>' is in the user's command. It uses the str_split method is called to create an array that is split using '>' as a delimiter. It also trims any leading and trailing whitespace that occurs. Finally, it sets the correct file permissions to newstdout.

After that special case is handled, the complie_argv and execvp methods are called to carry out the commands.

execute_pipe(char *userInput):

Called when a '|' is in the user's command. It uses the str_split method is called to create an array that is split using '|' as a delimiter. It then calls the complie_argv method twice, one for the right side of the pipe and one for the left.

The method then creates two child process, it executes or execvp's the right side first, then continues on to the right side.

str_split(char *a_str, const char a_delim):

Splits a string based on a delimiter. For example, if the following is passed:

```
months=[JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC]
```

Return is:

```
month=[JAN]
month=[FEB]
month=[MAR]
month=[APR]
month=[MAY]
month=[JUN]
month=[JUL]
month=[AUG]
month=[SEP]
month=[OCT]
month=[NOV]
month=[DEC]
```

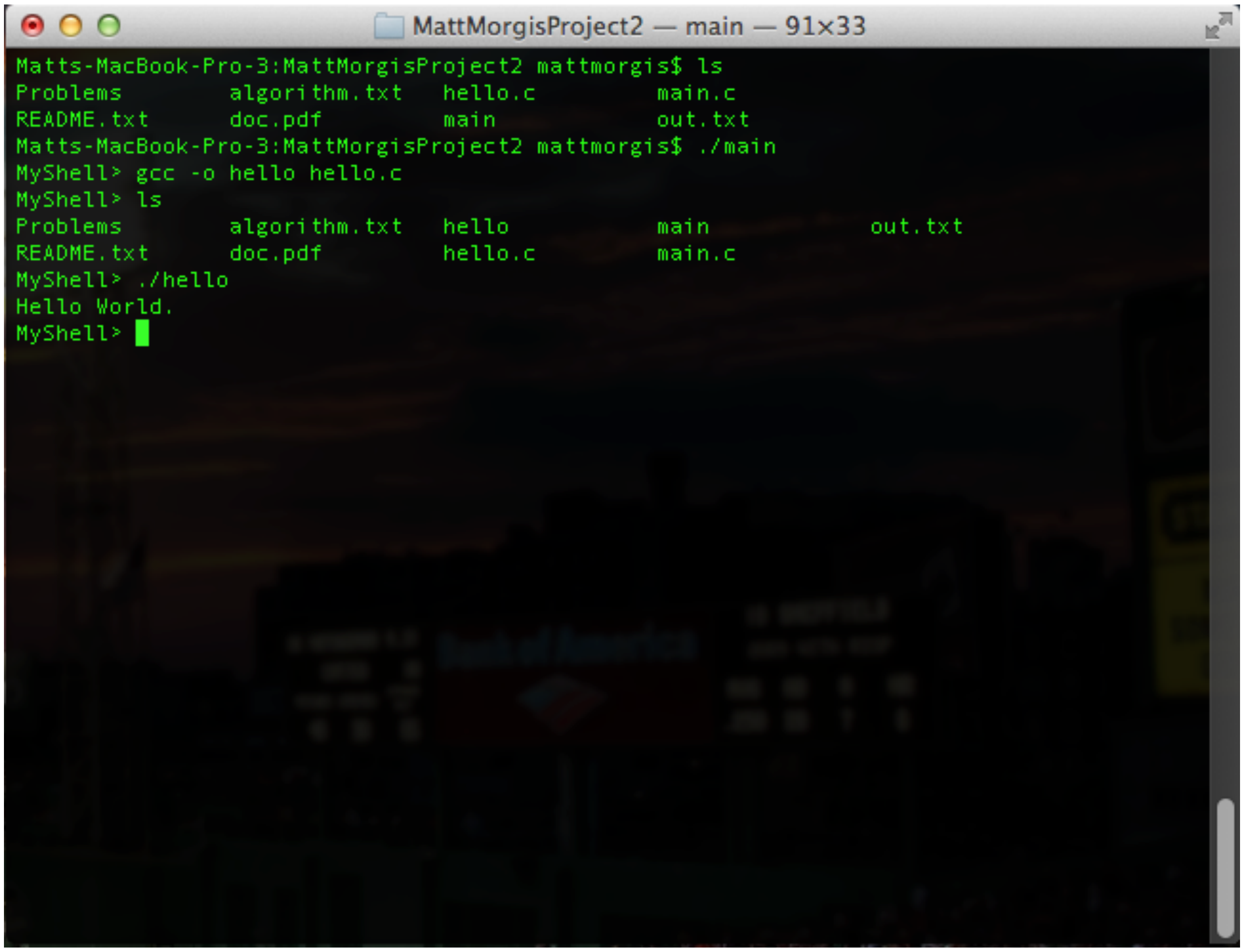
trimwhitespace(char *str):

Basic method to remove leading and trailing white space when a < or > are used in the command.

Testing

Problem A:

The objective of Problem A was to get the shell up and running, and to execute basic commands.



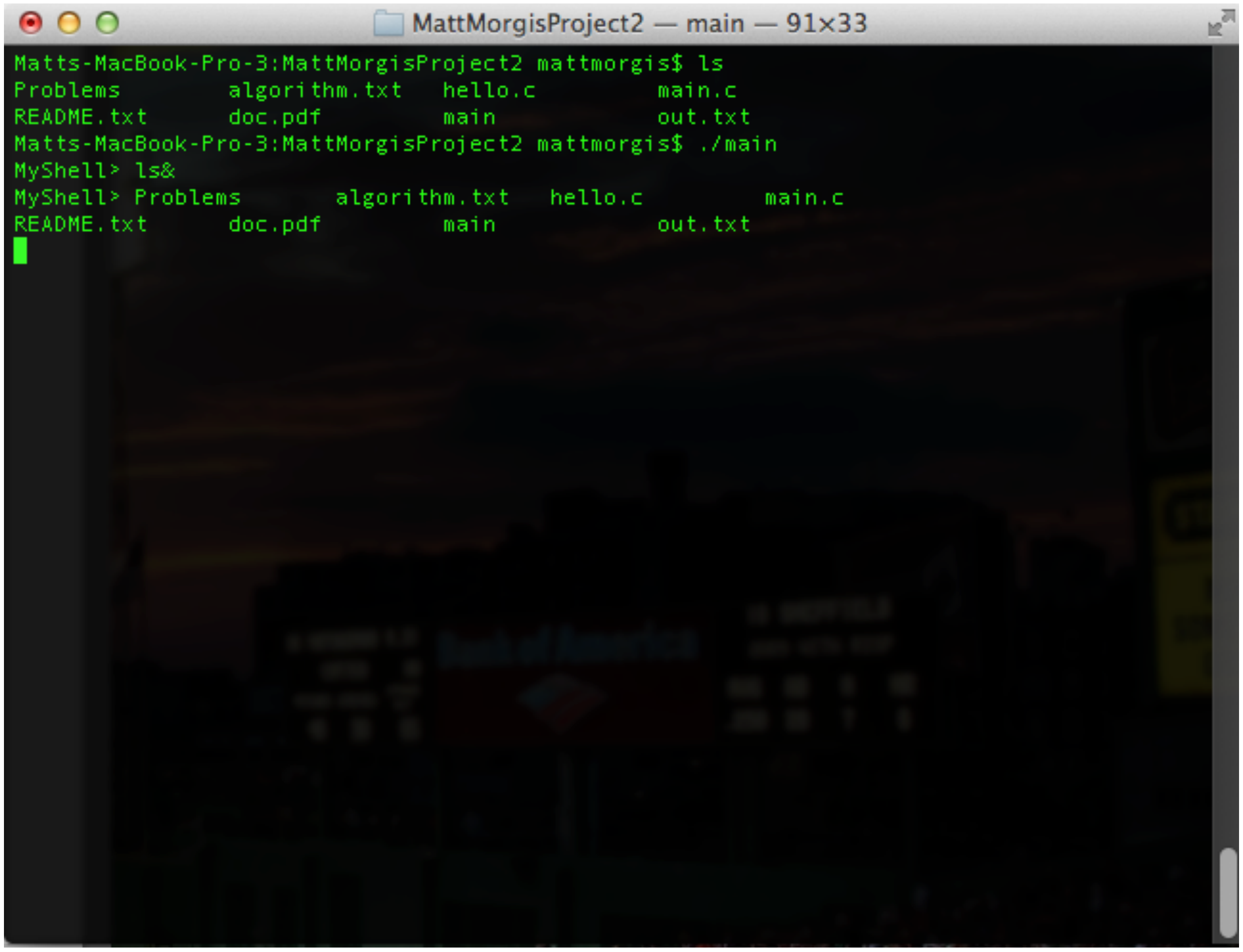
```
Matts-MacBook-Pro-3:MattMorgisProject2 mattmorgis$ ls
Problems        algorithm.txt   hello.c         main.c
README.txt      doc.pdf         main            out.txt
Matts-MacBook-Pro-3:MattMorgisProject2 mattmorgis$ ./main
MyShell> gcc -o hello hello.c
MyShell> ls
Problems        algorithm.txt   hello           main            out.txt
README.txt      doc.pdf         hello.c         main.c
MyShell> ./hello
Hello World.
MyShell> 
```

As seen in the screen shot, I run my version of the shell. I then use gcc to compile a simple Hello World program, and call the executable ‘hello.

I then ls to see if the file is created, and then run. The program runs correctly, and my test is successful.

Problem B:

Problem B had the objective of handling the & sign at the end of a command.



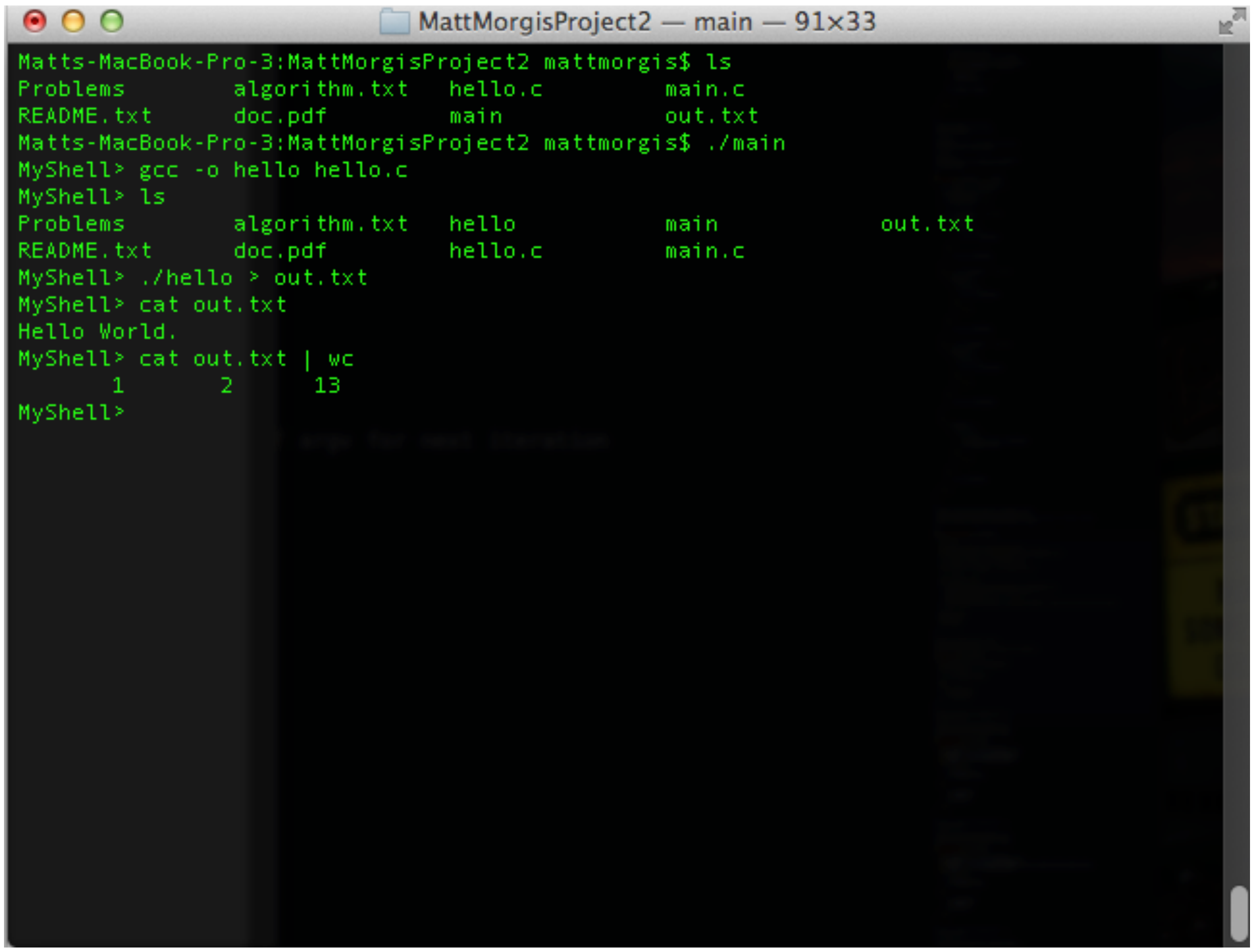
```
Matts-MacBook-Pro-3:MattMorgisProject2 mattmorgis$ ls
Problems        algorithm.txt   hello.c         main.c
README.txt      doc.pdf         main            out.txt
Matts-MacBook-Pro-3:MattMorgisProject2 mattmorgis$ ./main
MyShell> ls&
MyShell> Problems       algorithm.txt   hello.c         main.c
README.txt      doc.pdf         main            out.txt
```

In the above screen shot, I run a simple ls command with a trailing &. The prompt for the shell prints out before the command is executed. It then lists the files in my directory, and waits for the next command.

This happens on line 162 and 163. The execvp command is called without normal wait command following. This test proves that when a & is entered, the command is executed concurrently with the shell rather than waiting for the child to terminate.

Problem C:

Problem C's objective was to handle special characters like '<', '>', and '|'. I have special execute methods for each one.



```
Matts-MacBook-Pro-3:MattMorgisProject2 mattmorgis$ ls
Problems        algorithm.txt   hello.c         main.c
README.txt      doc.pdf         main            out.txt
Matts-MacBook-Pro-3:MattMorgisProject2 mattmorgis$ ./main
MyShell> gcc -o hello hello.c
MyShell> ls
Problems        algorithm.txt   hello           main            out.txt
README.txt      doc.pdf         hello.c         main.c
MyShell> ./hello > out.txt
MyShell> cat out.txt
Hello World.
MyShell> cat out.txt | wc
       1       2      13
MyShell>
```

The above commands give a full and complete test to the shell. I first compile a Hello World program, and print the output to an empty text file called out.txt.

I then print the contents of the file, and get a successful "Hello World" printed on the console. The > and < are now working correctly.

I then run 'cat out.txt | wc' to test the pipe command. The lines, world and character could are all printed to the screen before the shell asks for another input. This confirms piping is now working.