Operating Systems

P1

100531510, Bettinger, John
100531523, Emrich, Kurt
100495775, Pascau Sáez, Alberto

Description of the Code: **2**
mywc.c 2
Errors: 2
myls.c 3
Errors: 3
myishere.c 3
Error: 4
Test Cases: **5**
The main idea behind the testers: 5
test_mywc.sh 5
Test cases: 6
Output: 6
test_myls.sh 6
Test cases: 6
Output: 7
myishere.c 7
Test cases: 7
Output: 7
Conclusions: **8**

Description of the Code:

mywc.c

The first program intends to open a file and count its lines, words, and bytes. In our implementation, we intend to take a text file as input, open it without error, and read it byte by byte to count the number of words and lines it contains. It mimics the behavior of the wc command.

Our program first takes input for the name of a file and tries to open it in read-only mode. If a file is not provided, or the file provided does not exist, it will return an error message. Next, it initializes variables to hold the number of bytes, words, and lines and begins a while loop to read through the file. It loads BUF_SIZE bytes of the file to be read through. It updates the number of bytes by how many characters are loaded into the buffer, and then reads character by character to count the number of words and lines. For every newline character, \n, we update the number of lines. Counting words was trickier. We create a variable to keep track of whether or not we are in the middle of a word. This variable is 0 when we pass a tab or whitespace, and then changes to one once we find a non-whitespace character. When it changes from 0 to 1, indicating that we are inside of a word, we increment the number of words. After the loop, the number of bytes, lines, and words are printed. In the console, the number of lines, words, and bytes is followed by the name of the corresponding file.

Errors:

To check for errors, our program returns -1 if no file was specified or if the file could not be opened. ___ (add another error check that we do)

There are three possible errors that can occur with this program. It could be that an insufficient number of arguments are provided (< 2), or the parameter provided in argv[1] is an invalid file name and does not open correctly, or that an error occurs while reading the file due to an issue inside the text file.. If there is only 1 parameter provided to the program, a -1 error code is returned. If more than 1 parameter is provided, the second parameter located at argv[1] is taken as the provided filename and an attempt will be made to open the file using the open() command. If the open command returns a -1, it means that there was an error opening the file. It may not exist or there may be permissions issues. In this case, the program will also return a -1 and error message of “unable to open file”.

myls.c

The next program accepts a specified directory and prints its entire contents to the console. Each entry is printed on a new line. It emulates the function of the command (ls -f -1). It takes an argument for a path to a directory to be printed, or uses the current working directory if not provided with a directory path. It attempts to open a directory with opendir() and returns an error message if it was unsuccessful. To represent the entries of a directory it uses the “struct dirent” type and initializes a pointer called “entry” of that type to hold the entries as it iterates through the directory. Using the “readdir” function it iterates through the entries in the directory and prints them to the console. If there is an issue while it iterates through the directory, an error message is returned. If there is no error, closedir() is called to close the directory and the program returns a 0. Both the current and parent directories are shown in our display.

Errors:

To indicate errors, the program will return -1 if the directory that was passed as a parameter was unable to be opened. _____(another error check that we implement)

The two errors possible in this program are related to directory issues. If there is more than one argument provided, argv[1] is taken as the directory name and an attempt is made to open a path to that directory. If the opendir() command returns NULL, the directory provided is invalid, and the program returns -1 and an error message appears. The other error occurs if only one argument is provided and the getcwd() command fails. If one parameter is provided, the program attempts to take the current working directory as the directory path. If getcwd() returns NULL, then there was an error with that call, and a -1 error code is returned with an error message.

For ls -f -1, if you input a file, that file is just output. For our version of myls, if you input a file, you get a “not a directory” error.

myishere.c

The myishere.c program does not have an exact equivalent terminal command. Its goal is to check for the existence of a file in a directory. The program receives two arguments. The first argument is the name of a directory and the second argument is the name of a file that is supposed to be in that directory. It attempts to open the directory with opendir() and like with myls, it creates a pointer of type “struct dirent” to represent the entries of the directory and iterate through it. Then, it uses a while loop to iterate through the directory. With every entry that it finds, it compares its name to the provided file name using strcmp() to evaluate if the entry is the file sought after. If an error occurs during the while loop, the directory is closed and a -1 is returned. If the file is found, the while loop exits early, the directory is closed with closedir(), and the program returns a 0. If the loop completes successfully without finding the file, the directory is closed and the program returns a 0.

Error:

For error checking, our program returns -1 if there are too few arguments or if the specified directory cannot be opened. Additionally, … (another error check we create ourselves)

Similar to myls and mywc, errors occur in myishere relating to the number of arguments provided and their validity. If less than two arguments are provided, a -1 error and message is returned. If more than two are provided, the program continues. The parameter argv[1] will be taken as the directory name. If opendir() returns a NULL, the directory name provided was invalid in some way and the program will return a -1 and an “opendir error” message.

Our program prints to the console whether or not the specified file was found in the specified directory.

Test Cases:

The three c programs created for this lab have been tested using bash commands such as ls, echo or wc and the diff command to compare the output of the original commands versus ours. The complete code developed for these testers can be found in the following github repository:

OS_Lab1_Repo.

It is important to note that all the members of the team are mac users. Because of this we also implemented the testers into the .zsh shell. The code difference between both testers is nonexistent besides the type descriptor at the end (.zsh vs .sh) as both shells have identical syntax for basic commands.
The screenshots shown in the following parts are those of the .zsh files output. The .sh files create the exact same output.

The main idea behind the testers:

As mentioned in the report, there are ways of obtaining the expected output for each of the .c programs we developed. Once we obtain this information it can be compared using the diff shell command.
This command takes two strings (the output of the expected and the actual output) and compares them to see if they are equal.
We used this concept to create some tests. Once the test was performed, the expected and actual output were saved into .txt files and compared with the diff command.

At first we decided to perform a test directly from the terminal, this however was not efficient when going through different test cases and repeating the same test over time. To solve this problem the .zsh/.sh testers were created.
These files help us by executing all the tests at once with one joined output.

Finally, we would like to explain the use of if statements inside the testers. These statements are used to compare whether or not the diff output was empty. Given this information the program adds a line with “Test passed :)” or “Test failed :(“ . Different uses of the if statement will be better explained for each of the testers.

All of the testers use some directories and files created during the run phase of the files and then deleted at the end. This implementation is done in this way so that if in the future we wanted to give some input directly to the tester we would have a variable holding that information throughout the whole program.

test_mywc.sh

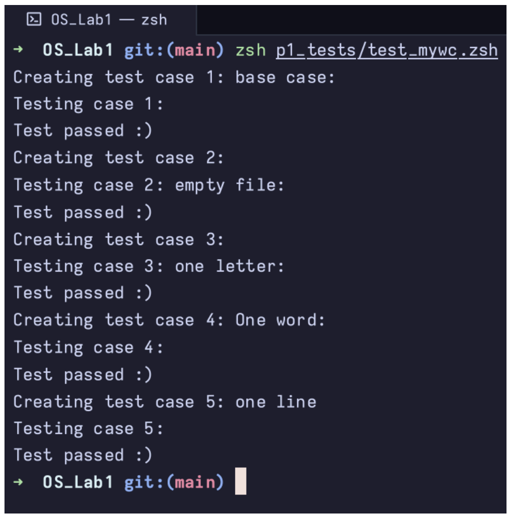
The tester created for mywc.c is test_mywc.zsh. This tester compares the output of the shell command wc with the output of mywc.c over the same input.
The main problem we encountered when creating this tester was the difference in spaces between the words, bytes and lines. Some shells output tabs instead of spaces.
To solve this the “ tr -d "[:blank:]" “ command was used. This command removes all blank spaces from the output of wc or mywc.c leaving only a concatenated string easier to compare.

Test cases:

5 test cases were created:

1. Base case: Input a file containing several lines of text
2. Empty file: Input an blank file
3. One letter: Input a file with only one letter
4. One word: Input a file with only one word
5. One line: Input a file with only one line of text

Output:



```
→  OS_Lab1 git:(main) zsh p1_tests/test_mywc.zsh
Creating test case 1: base case:
Testing case 1:
Test passed :)
Creating test case 2:
Testing case 2: empty file:
Test passed :)
Creating test case 3:
Testing case 3: one letter:
Test passed :)
Creating test case 4: One word:
Testing case 4:
Test passed :)
Creating test case 5: one line
Testing case 5:
Test passed :)
→  OS_Lab1 git:(main)
```

test_myls.sh

Using the ls shell command to compare with our own ls. No further comments or specifications about the development as it is a direct implementation of the idea stated in the introduction of the test.

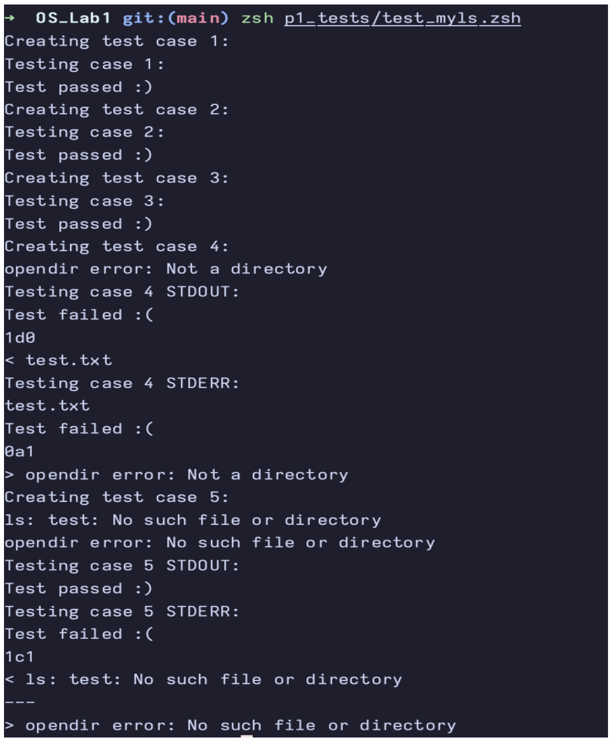
The main note of this test is focused on the two fails that seem to appear for the last two tests. As seen in the diff output (above the fail) this fails are not due to a problem in the .c program but a difference between the original ls and ours when outputting errors.

Test cases:

The following test cases were created:

1. Base case: Input a directory with .txt files and subdirectories
2. Only one file: Input a directory with no subdirectories and only one file
3. Empty directory: Input an empty directory
4. Wrong input: Input a file as an argument instead of a directory
5. Non-existent input: Input a non existing directory

Output:



```
→  OS_Lab1 git:(main) zsh p1_tests/test_myls.zsh
Creating test case 1:
Testing case 1:
Test passed :)
Creating test case 2:
Testing case 2:
Test passed :)
Creating test case 3:
Testing case 3:
Test passed :)
Creating test case 4:
opendir error: Not a directory
Testing case 4 STDOUT:
Test failed :(
1d0
< test.txt
Testing case 4 STDERR:
test.txt
Test failed :(
0a1
> opendir error: Not a directory
Creating test case 5:
ls: test: No such file or directory
opendir error: No such file or directory
Testing case 5 STDOUT:
Test passed :)
Testing case 5 STDERR:
Test failed :(
1c1
< ls: test: No such file or directory
---
> opendir error: No such file or directory
```

myishere.c

For this test we had to implement a different approach than in the previous two. There is no command that does what the myishere.c is meant to do. Therefore we had to come up with some other way of testing the program.

The solution was to use if statements.
When using some file and directory as an input, we use the following if statement to check that the file exist in that directory:

```
 # Compare to output a valid test:
   if [ -s diff.txt ]; then
         echo "Test failed :("
         else
            echo "Test passed :)"
            fi
```

For the cases where a non existent directory is passed, two nested ifs are created in the following way:

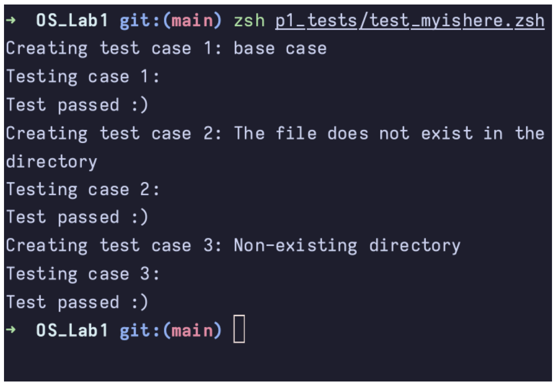
```
if [ -d "testdir" ]; then
      if [ -e "testdir/testfile.txt" ]; then
         echo "File testfile.txt is in directory $directory." > ishere.txt
         else
            echo "File testfile.txt is not in directory $directory." > ishere.txt
            fi
      else
         echo "opendir error: No such file or directory" > ishere.txt
         fi
```

Test cases:

The following test cases are created:

1. Base case: Inputted file and directory exist
2. Base case 2: Non-existent file: The inputted directory exist but no file with the name specified in it
3. Non-existent directory: The directory specified as input does not exist.

Output:



```
→  OS_Lab1 git:(main) zsh p1_tests/test_myishere.zsh
Creating test case 1: base case
Testing case 1:
Test passed :)
Creating test case 2: The file does not exist in the
directory
Testing case 2:
Test passed :)
Creating test case 3: Non-existing directory
Testing case 3:
Test passed :)
→  OS_Lab1 git:(main)
```

Conclusions:

Problems and how they were solved:
Personally, I do not have much experience with UNIX console and terminal commands. Figuring out how to compile the files, compare outputs, write commands to output files, and maneuver through the console in a Debian virtual machine was challenging. I had to spend some time reading back through the class presentation slides as well as looking up resources to better understand certain commands and the different ways in which they can be used.

Personal opinions:

We really enjoyed this lab as it allowed us to apply the topics learned in lecture to a real application. Additionally, it was rewarding to write our own code to build the terminal commands that we have been using for years such as the ls command. This lab was insightful because we were able to learn how these commands actually interact with the operating system and file manager. Most of us had found the operating system abstraction so significant that we had no clue how they were implemented. Now, we can see how C code is able to control the actual hardware of a computer.