Name: Robert Irwin

|  |  |  |
| --- | --- | --- |
| Problem | Points | Score |
| 1 | 35 |  |
| 2 | 35 |  |
| 3 | 30 |  |
| Total | 100 |  |

Notes:

1. For this exam you are allowed to open a terminal window on your computer, you are allowed to web surf with Google, but you cannot use online chat or other interactive services.
2. Create your solutions in an MS Word document and email it to the instructor at the end of the exam. Use “ECE 3822” in the subject line, and name your attachment using our usual convention of “lastname\_firstname\_ex01.docx.” Points will be deducted if you get the file name wrong.
3. In addition to providing your code, explain your solution to each problem.

You must show your code for each of these examples and briefly explain the steps you followed to reach your solutions. Your explanations don’t need to be long but must cover all the key points that resulted in your answers.

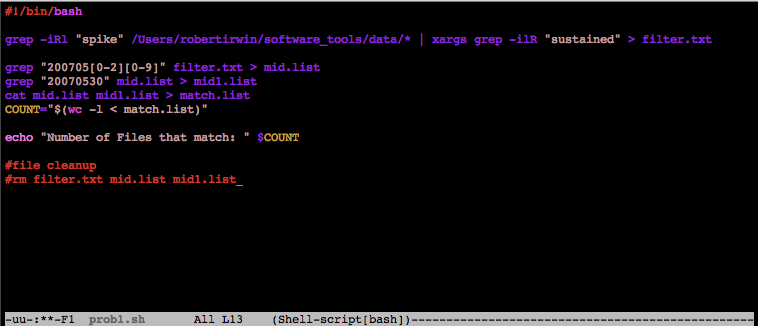
**Problem No. 1**: The text database we provided in class has a directory structure of the form:

data/book\_07/00009869\_20040409

where “00009869” represents the subject ID, and “20040409” represents the date. For all the sessions that occurred in the year 2007 between May 1 and May 30, and count the number of text files for which the word “spike” and the word “sustained” occur at least once. Note that your solution must be case insensitive.

**Solution:**

The code used to solve the problem is shown below.



*Figure 1: Code for Problem 1 <prob1.sh>*

In the code shown in Figure 1, we see that the first thing we want to do is search every file in the specified path recursively for “spike”. The recursive search is accomplished with the –R option. The other flags associated with grep are the –i, which makes the search case-insensitive, and the –l (“ell”) flag, which stops searching the file after a single match is found. This helps the program run a little bit faster. As we know, grep returns the path of each file containing a match. Therefore, we pipe the output to another grep command that uses the same flags and searches for “sustained”. The output of this is written to filter.txt.

There are two important points to make here. The first is the use of xargs in the pipeline. Xargs is a command that builds and executes command lines from standard input. Without xargs, we are trying to recursively search the standard input, which gives the following warning:

grep: warning: recursive search of stdin

Xargs handles the standard input, and removes the warning.

The second point that is important to make is the order in which we search the files. Why do we search for the words first, and then the dates? If we did it the other way around, our program would run faster because we wouldn’t have to go into every single file in the directory. We could simply stop the search at the names of the files. The order was chosen mainly for debugging purposes. When processing large amounts of files, it is important to provide “checkpoints” in your code so you can analyze what exactly your code is doing. This is why we generate so many files in the code in Figure 1. Because we are writing the intermediate grep outputs to text files, we can no longer look into the files specified by the path without passing each line of the text file to grep individually using a for loop. While this is a valid solution, there was a time crunch, and this solution did not come to mind right away.

Once we have all the files that contain both “spike” and “sustained” we look for the files that were created during the time May 1-30 2007. This requires two steps because there are 31 days in May. If we wanted any date in May we could simply grep the filtered list of files created by our first grep command for “200705” because we would not care about what day in May the file was created. Because we do not want May 31, we have to first grep for “200705[0-2][0-9]” which corresponds to any file created during May 01-29. Then we grep the same filtered list of files for “20070530” which corresponds to May 30, 2007. Notice in figure 1 that these outputs are in two different files. We can concatenate the files using the “cat” command, which is seen in Figure. Then we do a “wc –l” on the output of the concatenation to get the number of files that match our search criteria.

The output of the code above is shown below in Figure 2.



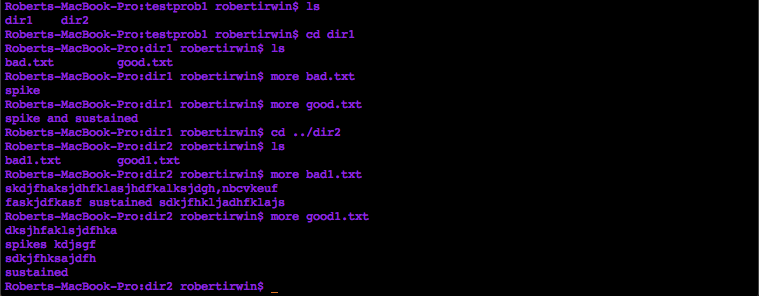
*Figure 2: Output of prob1.sh*

To ensure our code produces the correct output, we will first show that out of all the files that were found, we grabbed only the files from May 1-30 2007. This is shown in Figure 3.



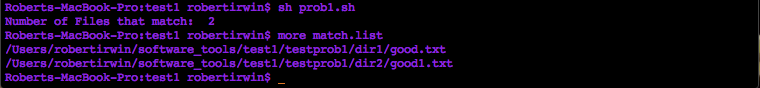
*Figure 3: Files that met Search Criteria*

From Figure 3, it is evident that the grep command we used only found files that met the date criteria. Now we will test our code on a small number of files and directories. Figure 4 shows the contents of the test directory, subdirectories, and files in those directories.



*Figure 4: Contents of Test Directory*

If our code works properly, we expect two matches. The grep-ing for the dates will be omitted from the code that is run because we have already demonstrated that the grep-ing for dates works as expected. This is to ensure us that our program only finds files that contain grep and sustained. The output of running our script on the test directory is shown below in Figure 5. It is important to note that we changed the path of the first grep command shown in Figure 1 appropriately.



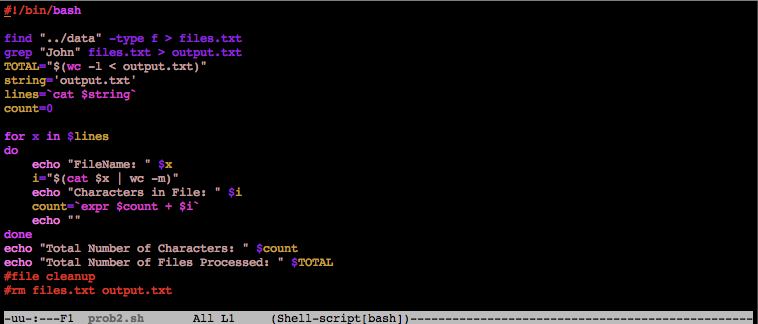
*Figure 5: Output of Test*

In Figure 5, we see that our code produces the expected output, and our code finds the expected files. Therefore, we can conclude that our code functions properly.

**Problem 2:** For the text database, generate a list of filenames whose full pathname contains the name “John”. Write a shellscript that loops over this list and counts the number of characters in the file. Your shellscript should output each filename as it is processed, the number of characters in the file, and a summary that shows the total number of files processed and the total number of characters.

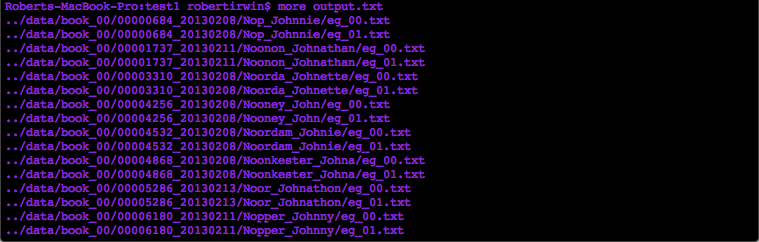
**Solution:**

The code used to solve the problem is shown below in Figure 6.



*Figure 6: Code for Problem 2 <prob2.sh>*

In the code above, the first thing we do is use to find to create a list of every file in the data directory. Then we grep that text document for the string “John”. This outputs any filename containing the string “John”. These files are stored in output.txt. A portion of output.txt is shown below in Figure 7 for reference.

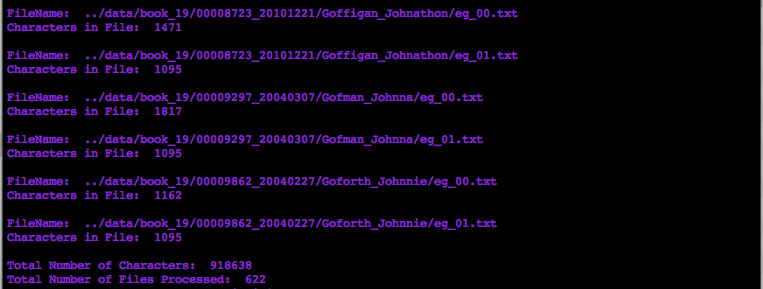


*Figure 7: Portion of output.txt*

To get the total number of files processed, we do a “wc –l” on output.txt. This counts the number of lines in the file which corresponds to the total number of files. This number is stored in TOTAL.

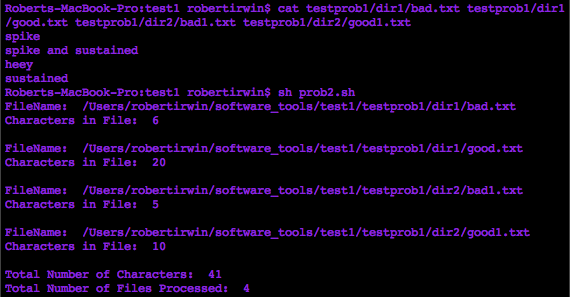
Then we create a for loop where “x” is each line in output.txt. To print the file name as it is being processed, we do an echo $x inside the for loop. This accomplishes the task of printing the file name, as well as ensures us that we our code is functioning properly.

To count the number of charcaters in the file, we pass the filename to wc and throwthe –m flag, which counts the number of characters in a file. This is stored in a variable “i”. We then use “expr” to add “i” to the current value of count, which we initially set to 0. The variable “i” is printed to show the number of characters in each file. After every file has been processed, the final count is printed, as well as the total number of files processed. A sample of the output is shown below in Figure 8, as there are around 600 files being processed.



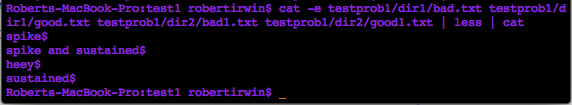
*Figure 8: Sample of Output of <prob2.sh>*

Now we will use the same test directory as problem 1 to debug this script. The contents of the files have been modified to make it easier to count, and we no longer search for files containing “John”. We are simply checking to ensure our program counts the number of characters correctly. The output is shown below in Figure 9.



*Figure 9: Output from Test Directory*

We know that we are only working with 4 files, so the “Total Number of Files Processed” is as expected. We also notice that each of our counts seem to be one more than what we expect. For example, it says there are 6 characters in “spike”. However, if we show whitespace characters, represented as dollar signs, we can see that there is actually a new line character at the end of each line! This accounts for the extra character count. The visible newline characters are shown below in Figure 10.



*Figure 10: Contents of Files with New Line Characters Shown*

We have just verified our script using a small, test directory, and are therefore confident in the functionality of the script.

**Problem 3:** We have discussed the relationship of the .bashrc file to your overall environment. Write a script that sets an environment variable called “MY\_OS” to the specific version of the operating system loaded in your machine. You cannot hardcode the operating system version. You must get this from the system so that your script can run on any Linux machine. You also need to set MY\_PROC to the model name of the processor that your system is using (e.g., Intel Xeon). Again, this must be done in a machine-independent manner and work on any Linux system.

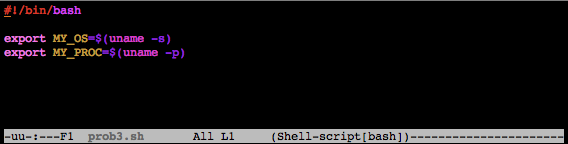
This script must also export this variable back to your root shell. Specifically, I should be able to do the following:

1. login
2. run your script (e.g., sh my\_script.sh)
3. echo $MY\_OS
4. echo $MY\_PROC

and see the information. Alternately, I could embed this script in your .bashrc file.

**Solution:**

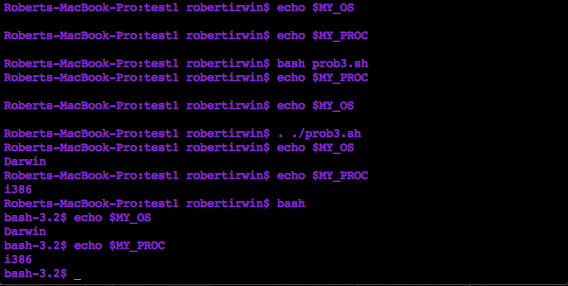
The code used to complete the problem is shown below in Figure 11.

****

*Figure 11: Code for Problem 3 <prob3.sh>*

The code is fairly simple. We simply set the variable names equal to the output of uname, which displays information about your system. Then we use export to have access to the variables outside of this function.

It is important to note that the way this script is run is very important. For example, if you run the code with “sh script.sh” or “bash script.sh” the environment variables will only be available in subshells of the shell the code was run in. If, however, we run the code with “. ./script.sh” or equivalently, “source ./script.sh”, the variables are exported to the current shell, and all subshells. Proof of this is shown below in Figure 12.



*Figure 12: Results of Problem 3*

As seen above in Figure 12, before the script is run, the variables MY\_OS and MY\_PROC do not exist. Then we run the code with “bash prob3.sh”and the variables are still nonexistent in the current shell. When we run “. ./prob3.sh” however, the variables are accessible in the parent shell, as well as subshells of that shell.

It is important to note that the syntax in a Mac terminal is slightly different command. Normally, to print the operating system name, one would use “uname –o”. In a Mac terminal however, the syntax is “uname –s”. A –s flag in Linux will print the kernel name.