CS 214: Systems Programming, Fall 2020
Assignment 2: File Analysis

0. Introduction

In this assignment you will make use of the filesystem API and the POSIX thread library to implement a simple plagiarism checker.

Plagiarism is a self-defeating shortcut that plagues many student bodies. For those unacquainted with it, plagiarism is the act of claiming someone else's work as your own. Paradoxically, many college and university students pay large sums to access experts in their professed fields, but do not make use of them. They instead substitute the work of others, guaranteeing little to no utility for the tuition dollars they spend, making subsequent courses even more difficult and all but assuring they will not have the skills they ought to once employed! It is naturally all but impossible to imagine, but it certainly does happen.

Your task will be to write an extremely simple plagiarism detector using some basic computational linguistics. Given a base directory to work from your detector will scan it for all files and subdirectories. It should start a thread to handle each file or subdirectory it finds. A subdirectory should be handled by doing the same operation, namely scanning it for all files and subdirectories and making a thread to handle each. A file should be handled by reading in its contents and building a discrete distribution of all the words that appear in it. Once done, your detector should compare all distributions and classify 'suspicious' files.

1.a Implementation - Scanning & Tokenizing

Your code should take a single command line argument, a base directory to start from. The directory name may be absolute or relative and may or may not be terminated with a slash. Be sure to check if the argument is indeed a valid directory you can access before attempting to use it. You should write two functions: a file-handling function and a directory-handling function that write results to a shared datastructure.

Directory Handling:

Your directory-handling function should check the directory to see if it is accessible. If it is not, it should output an error and return gracefully. If it is accessible, it should open the directory with opendir() and iterate through its contents with readdir(). Whenver is finds a directory it should create a new pthread running another copy of the directory handler. It should pass as an argument to that thread that holds the name of the directory to be scanned concatenated to the path so far, a pointer to the shared data structure and a pointer your synchronization mechanism to the new thread. Whenver is finds a file it should create a new pthread running another copy of the file hander. It should pass as an argument to that thread that holds the name of the file to be tokenized concatenated to the path so far, a pointer to the shared data structure and a pointer your synchronization mechanism to the new thread. Anything it finds other than a directory or regular file it should ignore and pass over without warning or error. Make sure you join() all your threads.

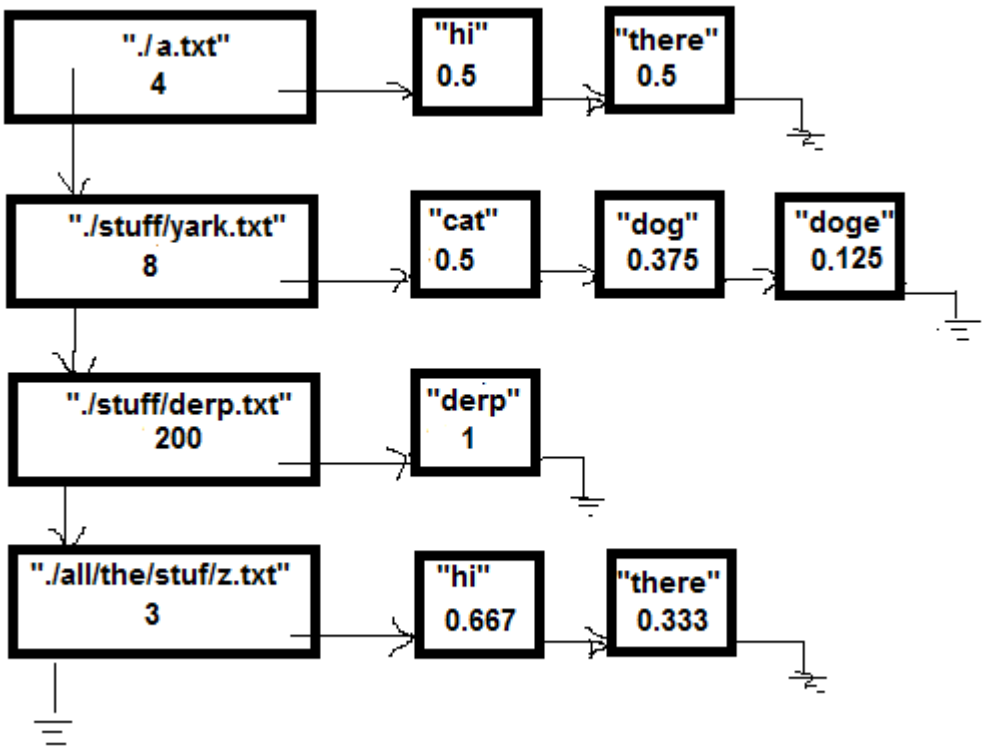
File Handling:

Your file-handling function will update a shared datastructure representing the frequency of all tokens in all files as it scans through the file it has been given. It should first examine the file and make sure it can access the file. If not, it should print an error message and return gracefully. If it can access it, it should lock the synchronization mechanism it was given, add an entry in the shared data structure it

was given holding the file's path/name and total number of tokens scanned, unlock the synchronization mechanism and then start tokenizing the file. Once done tokenizing, your thread should divide each token amount by the total number of tokens found, computing the discrete probability of each. Your thread should also sort its token list alphabetically. This is easiest to do using insertion sort as you add new tokens to the list.

Notes:
All files will consist of ASCII text. All tokens will be space-separated or newline-separated. Tokens will consist of alphabetic characters and hyphens. Ignore all punctuation and convert all tokens to lowercase in order to ease classification.

Your shared data structure can take any shape you like, although nothing more complex than a linked list of linked lists, each representing a file's tokens and the likelihoods for each is necessary.



1.b Implementation - Threading & Synchronizing

Threading:
As noted in 1.a above, each new directory or file found should result in a new pthread being created to handle it. This will allow your code to work on multiple files at once without slowing down due to IO effects. In order to make sure your threads all run to completion, make sure your directory handler function stores the handles for all threads it creates and join()s them all before it quits.

Synchronizing:
Each file-handling thread should access the same data structure, as once you are done reading in file information it will need to be analyzed. Since each thread will be working independently, but working on the same data structure, you should maintain a synchronization mechanism to control modification of the data structure. When a file-handling function starts up, once it has verified its file can be read from, it should: lock the synchronization mechanism, add an entry to the data structure to

represent the file it will be tokenizing, then unlock the synchronization mechanism. It does not need to lock or unlock again. This is to ensure that you do not have two threads attempting to add a new entry to the shared data structure at once. You do not need to lock again in same thread because each thread accesses only its own entry, once created.

Note:
Your file-handling function does not need to create any threads and your directory-handling function does not need to synchronize or access the shared data structure.

1.c Implementation - Preparation & Pre-Analysis

Preparation:
Your main() should examine the intial directory given. If it is accessible and exists, it should intialize the shared data structure and a synchronization mechanism. A mutex is recommended. Since all threads will need to acceess both the data structure and the synchronization mechanism, make sure to allocate space on the heap for both. You'll need to define some type of struct to pass your multiple parameters to your threads. Your main() should then call the directory-handling function on the initial directory it was given. You do not have to make this call a thread since your main() has nothing else to do until all tokens from all files have been read in. If you want to make it a thread, that is fine, but be sure you join() on it.

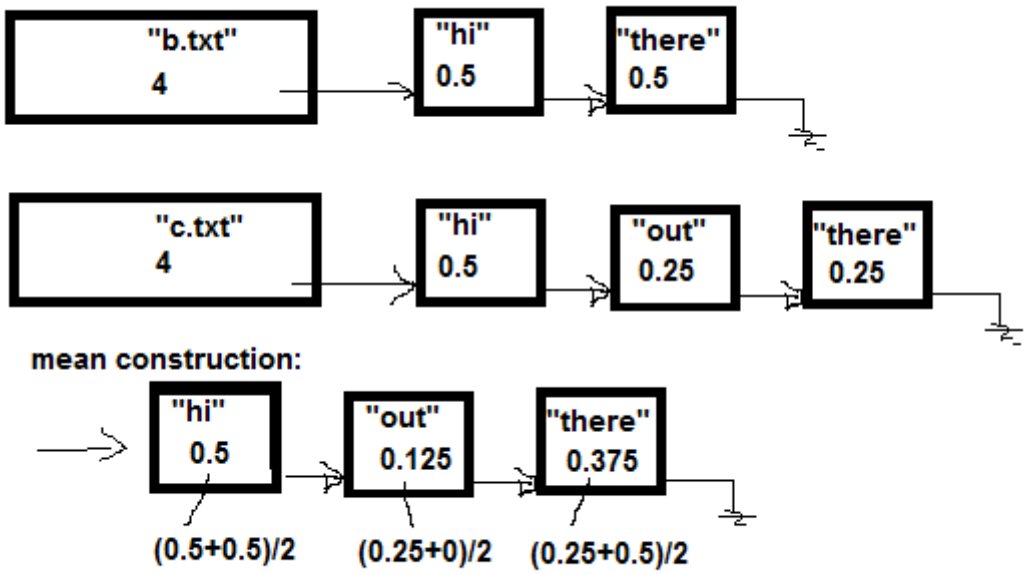
Pre-Analysis:
Once the initial call has returned, main() can begin examining the data. It should first check to make sure the data structure has been written to, if not it can emit an error and stop. If defined, it should next sort all the files in the shared data structure by their total number of tokens. If there is only one entry in the data structure your main() should emit a warning and stop.

1.d Implementation - Analysis

In order to test your discrete distributions for similarity, you'll compute the Jensen-Shannon Distance between each of the distributions you've selected for comparison.

First you need to compute the mean distribution of the two token distributions you are analyzing. Construct a new mean token list by iterating through the two and computing the mean of the probabilities of any tokens that are the same and halving the probabilities of any that are unique to one distribution.

$$MeanProb(tokenX) = (FirstDistributionProb(tokenX) + SecondDistributionProf(tokenX))/2$$



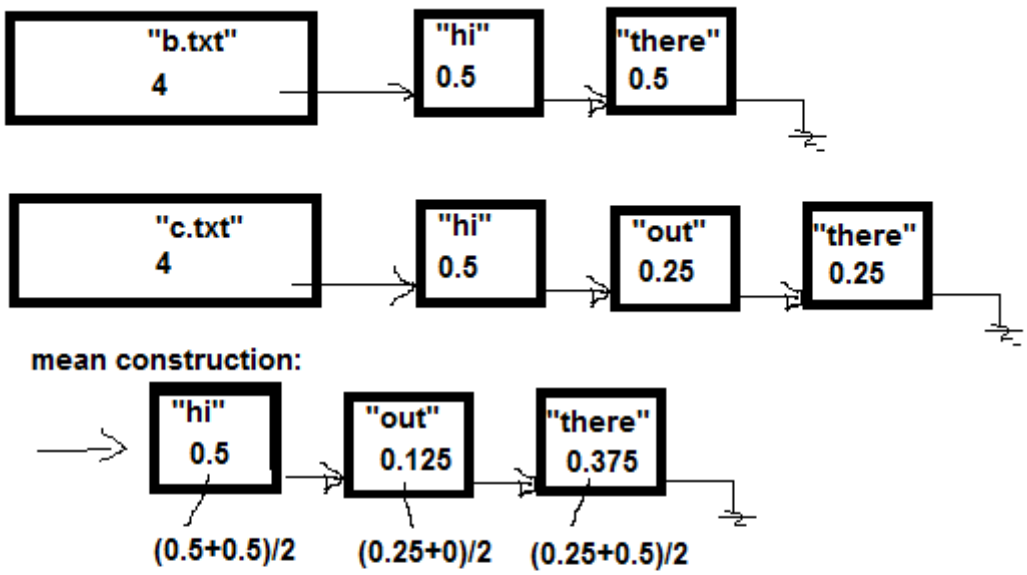
Next, compute the Kullbeck-Leibler Divergence of each distirbution you are analyzing from the mean distribution. You do this by, for each token in a list, if you find that token in the mean, multiply the discrete probability of its occurrence in the list by the logarithm of its discrete probability in the list divided by the probability computed for it in the mean distribution.

$$KLD(First||Mean) = \sum First(x) * \log\left(\frac{First(x)}{Mean(x)}\right)$$

$$KLD(Second||Mean) = \sum Second(x) * \log\left(\frac{Second(x)}{Mean(x)}\right)$$

Note that if a token is in one list but not the other, you will never get a divide by zero issue.

Note that this computation is directional. While calculating the Kullbeck-Leibler Divergence of a list from the mean distribution, you may ignore tokens that are not in a list, but are in the mean distribution. For instance, in the example above, “out” is in the mean distribution, but not in the list of tokens for file “b.txt”, so its probability of occurrence is 0.



"b.txt" KLD:

0.5*log(0.5/0.5) + 0.5*log(0.5/0.375)
0 + 0.6247 = 0.6247

"c.txt" KLD:

0.5*log(0.5/0.5) + 0.25*log(0.25/0.125) + 0.25*log(0.25/0.375)
0 + 0.0075 + -0.0440 = -0.0365

Finally, to compute the Jensen-Shannon Distance, compute the mean of the two Kullbeck-Leibler Divergences. Given the example above: (0.6247-0.0365)/2 = 0.2940.

Theoretically, the Jensen-Shannon Distance can generate a 'score' on the interval [0, 1), in practice, the score ranges from [0, 0.5). The smaller the 'score' is, the closer together the two distributions are. As you might notice, two distributions that are exactly the same will result in a Jensen-Shannon Distance of 0. Look at the value for the token “hi” in the example above.

1.e Implementation - Output & Reporting

Output the Jensen-Shannon Distance for each file pair compared. From smallest to greatest number of tokens. Color-code the Jensen-Shannon Distance as:

red for [0, 0.1]
yellow for (0.1, 0.15]
green for (0.15, 0.2]
cyan for (0.2, 0.25]
blue for (0.25, 0.3]
and white for any greater than 0.3.

Report with the format:

0.01 “hi.txt” and “yurp.txt”
0.162 “hi.txt” and “hello.txt”
0.293 “yurp.txt” and “snarf.txt”

2. Hints

You can develop much of this code in distinct modules. The segments that iterate through directories and files can be written and tested by outputting the names of the files they find, made in to void*/void* functions and then run as threads. The mathematical calculations can be coded and tested on hard-coded test data. Tokening can be developed as a function on its own.

It is both intended and recommended that you make good use of your previous work to complete this assignment. A number of homeworks and assignments span segments of the functionality you need to implement here.

3. What to Turn In

A tarred gzipped file named Asst2.tgz that contains a directory called Asst2 with the following files in it:

Asst2.c:	your main
Makefile:	must have both an “all” and “clean” directive all should build an executable named “detector” and be first clean should remove any files automatically constructed by the Makefile
testcases.txt	a brief discussion of your testcases. Be sure to include at least five.

Your code will be invoked as: ./detector “./somestufftotest”

4. Grading

Correctness - how well your code operates
Testing thoroughness - quality and rationale behind your test cases
Design - how well-written and robust your code is, including modularity and comments