Warm-Up Homework One
Streaming Text Processing, Tokenization, Counting Words Occurrence, Sorting, pointers manipulation, file I/O and Timing utility, Threading(optional)

Problem Description

You are required to count word occurrence in a large text file, in which we have all types of punctuations. Problem (1), in a **sequential** program, you have to extract English words in the provided text file and count occurrence for each word in the text file. In addition, you have to sort the list or the array of distinct words that you obtained. a) Sort the collection according to dictionary order of the word literal, and b) Sort the collection according to the word occurrence count. After sorting, you have to write the sorted list of words and their occurrence into two different output files, each for separate sorting rules. **Optional** Problem (2), in a **parallel** program using 4 threads, you implement the same functionality using Pthreads library and compare your sequential code with your parallel code, then output speedups and efficiency information as shown in class demo. (You earn extra credit 40 points if you do problem 1 and 2.)

What is Provided?

Two files are provided for processing, testfile1 and testfile2. You can use testfile1 to debug your code because it is smaller. You have to run your program on testfile2 to see if it works. The testfile2 is a big text file, which is a subset of Wikipedia and contains around half of million English words.

Basic Idea to Extract Word

We assume all words in the provided text are correct English words. Suppose we have a line of text, with a line feed at the end,

"You are a student. Who's your advisor? i.e. teacher. I'm your friends."

We can see that English words are delimited by white space or punctuations. Basically, starting from the first letter of a word, we remember the start position of a word. As we scan down the next few characters, if we encounter a space or a non-alphabetic character, we know that from the start position we remembered to the character that precedes this non-alphabetic character, defines an English word.

After we find a word in that line of text, we increment a counter for that word if this word has already been in the list/array, otherwise we copy it into an object and insert the object into the list/array. NOTE that, when check whether a word has already existed in the list/array, you are required to use CASE-INSENSITIVE comparison.

After we extracted a word, if its length is one (contains only one letter), we discard it except for 'a' and 'i'. That is, we will not insert the one-letter word into the linked list/array EXCEPT THE WORD "I" or "A" (or "i" or "a").

For example, in the bold sentence above, after "who" we get an apostrophe ', that means we get an English word "who". Then, after we throw away the apostrophe following word 'who', we get a new word start position, that pointing to letter 's'. Following 's' we got a white space. According to our algorithm, the single letter 's' is a word. But here we discard this single-letter word that the algorithm finds. The same is true for letter 'e' in "i.e. " and letter 'm' in "I'm".

In order to extract all English words in a piece of text, we have to specifically look at each character in the text from the first line to the last line. It is like processing a stream of ASCII characters.

Requirements

1) You write two solutions to a same problem by using both sequential programming and parallel programming. **Note that the parallel solution is optional, which is considered as extra credits. Please clearly output a message on the standard output if you did the bonus part.**
2) When implementing a parallel solution to this problem, in which each thread processes a different set of lines of texts, using data parallel approach. Each thread is responsible to tokenize many lines of text, generating a local list of words. Then the thread sorts the **local** list of words according to two rules. Note: **the local sorted list of words has to be merged into a global sorted list at last. You have to write this merge function. If you can parallelize this merge operation using multiple threads, you get a bonus point of another 30 points.**
3) You have to measure the execution time-cost for your sequential solution and your parallel solution if applicable. Then you calculate the speedups and efficiency information as shown in class demo, and display this information on the standard output.
4) When you are measuring your program execution time-cost, please exclude the file I/O operations, instead only include the word tokenization and sorting operations. That is, in your program you have to first read all lines of the text into a data structure, then you process them.
5) If applicable, in your parallel solution, you parallelize the word tokenization and the sorting operations. Please **do not** parallelize the file read and write operations.
6) You are free to choose your own data structures (arrays or linked list or doubly linked list) and to define your own structure types in C. You are free to use the build-in **qsort** function or write your own merge sort. But You cannot use the build-in strtok() or strtok_r() function to extract words.
7) But you are required to use either merge sort or quick sort, because they are quick.
8) Run your solutions on the testfile2 file, the big one. You have to generate two text files with required format. The first output file should be word occurrence for each word in the provided text file, records are sorted according to word literal in an alphabetical order. The second output file is also word occurrence for each word in the file, but records are sorted by the word occurrence in descending order. That is, the most frequently used words are listed first in the file. File format are provided in the test case section.
9) Carefully design your functions by decomposing the whole problem domain, try your best to use small functions. So that, you can **reuse most your functions** in the Parallel Solution if applicable.
10) Please organize your code using .h file and .c files, with all function prototypes placed in .h files.
11) Add a make file to compile your code into a target **hw1**. Also add a rule in the make file to run your executable, that is, after the grader types in **make,** we should be able to see the two output files.
12) Use free() function to free all dynamically allocated memories after they are no longer useful. Please document your code properly where they are hard to understand.

Submission:

Wrap up your source files, input files and your output text files into a single zip file. Name your zip file as *LastNameFirstInitial*hw1.zip. For example, if your legal name is Will Smith, you should name your zip file as smithwhw1.zip. Please do **NOT** modify the provided text file. So that we make sure

that every student use the same data file. Please submit your single zip file on EWU Canvas by following CSCD439 Course →Assignments→hw1→ Submit Assignment to upload your single zip file.

Test Cases

For the input file **testfile1,**

We should get two output files. The first sortedWord.txt is as follows.

English Word	Count
a	2
Basically	1
calling	1
each	1
file	3
for	1
from	2
fscanf	1
function	1
I	1
in	1
my	1
scanning	1
string	1
strings	1
the	1
then	1
using	1

Your output file is required to have the same format in a tabular fashion. This is a part of grading rubrics.

The second output file is sortedOccur.txt, looks like

English Word	Count
file	3
a	2
from	2
then	1
strings	1
string	1
scanning	1
my	1
in	1
I	1
function	1
fscanf	1
the	1
for	1
using	1
each	1
calling	1
Basically	1

Your output file is required to have the same format in a tabular fashion. This is a part of grading rubrics.