CSI 402 – Systems Programming

Programming Assignment 3

Date given: Oct. 31, 2017 **Due date:** Nov. 09, 2017
Total grade for this assignment: 100 points **Weightage:** 3%
Note: Programs that produce compiler/linker errors will receive a grade of zero.

A. Purpose. Search about "indexing" in Google and you will get results related to FamilySearch services or even investment articles. This assignment has nothing to do with either of these. Instead, our focus here will be on indexing files. A google search on "indexing files" return results related to improving file search performance in Windows among others. Indexing is an essential preprocessing step for searching. During indexing data is collected, parsed, and stored to facilitate **fast** and **accurate** information retrieval. Index design incorporates interdisciplinary concepts from linguistics, cognitive psychology, mathematics, informatics, and computer science. Indexing has become increasingly important for web searches due to the large number of documents that are available on the Web.

The purpose of an index is to optimize speed and performance in finding relevant data for a search query. Without an index, a search engine would have to scan every document, which would require considerable time and computing power. As an example, while an index of $10,000$ documents can be queried within milliseconds, a sequential scan of every word in $10,000$ documents, each consisting of $1,000$ words on average could take hours. The additional storage required to store the index, as well as the considerable increase in the time required for an update to take place, are **traded off** for the time saved during information retrieval.

Your task in this assignment is to develop an indexing program, called `indexer`. Given a set of files, your program should parse the files and create an inverted index, which will map each term found in the files to the subset of files that contain that term. In your indexer, you will also maintain the frequency with which each term appears in each file. See the examples below for details. The inverted index should be maintained in **sorted** order by word. See the example below for details.

For simplicity, we will define `terms` as any sequence of consecutive alphanumeric characters (a-z, A-Z, 0-9). All other characters are to be treated as separators. Examples of terms according to the previous definition include `‘‘a’’, ‘‘aba’’, ‘‘c123’’, ‘‘1’’, ‘‘454’’`. If a file contains "This is an//$example12", your program should tokenize this text as follows: `‘‘this’’, ‘‘is’’, ‘‘an’’, ‘‘example12’’`. See the example below for details.

The purpose of this assignment is to familiarize yourself with (i) C programming, (ii) splitting a program into multiple files (including header files), (iii) using make and makefile, (iv) file operations, (v) directory operations, (vi) defining and using data structures, and (vi) observing the performance of these structures for search. Notice that objectives (i)-(v) are the same as in Programming Assignments 1 and 2. The necessary information about splitting a program into multiple files (including header files), using make and makefile, file and directory operations, links, and necessary

data structures for this assignment have already been presented in class.

B. Description. The executable version of your program must be named `indexer`. Your `makefile` must ensure this. The `indexer` program must support the following usage:

```
indexer [inverted-index file name] [directory or file name]
```

The first argument, `inverted-index file name`, is the name of a file that your program should create to hold the inverted index. Your indexer should save the index in the following format:

```
<list> term
name1 count1 name2 count2 name3 count3 name4 count4 name5 count5
</list>
```

See the example below for details.

The second argument, `directory or file name`, is the name of the directory or file that your indexer should index. If a file, your program needs to index that single file. If a directory is provided instead, your program will need to **recursively** index all files in the directory and its sub-directories.

If no argument is given, your `indexer` program should operate on the current working directory and generate file "`invind.txt`". If only a directory is provided as argument (i.e., the first argument is a directory), your `indexer` program should operate on the specified directory and generate the default index file "`invind.txt`".

When indexing files in a directory, you can assume that files cannot have the same name. However, files with the same name may be located in different sub-directories. Thus, you need to devise a mechanism to differentiate between files in the inverted index.

C. Error Handling. Your program must detect the following fatal errors. In each case, your program should produce a suitable error message to `stderr` and stop.

- The number of command line arguments is more than two.
- A `directory or file name` directory doesn't exist.
- Directory `directory` cannot be accessed.
- The contents of `directory` or `file name` cannot be accessed.

D. Example.

Let the following be the content of `/home/user/files`, where the first column indicates the filename, followed by the contents of the file:

```
Filename – File Contents
f1.txt    A dog name name Boo
f2.c      A cat name Baa
```

When invoked without arguments in this directory, your `indexer` should read through the files in the directory and produce the following inverted index, in **sorted** order by word:

```
"a" → ("f1.txt", 1), ("f2.c", 1)
"baa" → ("f2.c", 1)
"boo" → ("f1.txt", 1)
"cat" → ("f2.c", 1)
"dog" → ("f1.txt", 1)
"name" → ("f1.txt", 2), ("f2.c", 1)
```

The above depiction just gives a **logical view** of the inverted index. In your program, you have to define appropriate **data structures** to hold the mappings (i.e., term → list), the list of records, and the records (file name, count). After constructing the entire inverted index in memory, your indexer should save it to a file named "invind.txt" (since no argument was passed in this example). The inverted index from the example above would look like

```
<list> a
f1.txt 1 f2.c 1
</list>
<list> baa
f2.c 1
</list>
<list> boo
f1.txt 1
</list>
<list> cat
f2.c 1
</list>
<list> dog
f1.txt 1
</list>
<list> name
f1.txt 2 f2.c 1
</list>
```

D. Structural Requirements. Your submission must have *at least three* C source files, *one* or more header files, and a `makefile`. Additional requirements on the C source files are as follows.

- One source file must contain just the `main` function.

- A second source file must contain only the function that creates the index.
- A third source file must contain only the function that browses through the files.
- Your makefile should include a clean target.

Requirements specific to the index file are as follows:

- The mapping is to be maintained in *ascending* sorted order based on the ASCII coding of characters (i.e., "a" before "b" and "aa" before "ab").
- Records in each list are to be maintained in *descending* sorted order based on frequency counts of the terms in the files.
- You need to normalize all upper case letters to lower case letters in the terms (the other way around is OK too).

E. Submission Instructions. Make sure you are logged in to ITSUnix, and your present working directory contains only the files you wish to submit. Use the `turnin-csi402` command to submit all of the .c and .h files, and any file named 'makefile'. Make sure there are no extra .c or .h files in the current directory. If your makefile is called 'Makefile' (with a capital M), then make sure you use a capital M in the turnin command. Instructions for using the `turnin-csi402` command have been provided in Programming Assignment 0, which you should have already submitted.