Assignment 02

Due: Beginning of the class, Mar 1st

You must work on this assignment on you own

Part II Programming – Thread communication (70 points)

Threads share the same process address space and other resources like opened files etc.

This assignment is designed for you to exercise the use of POSIX threads to carry out multiple related tasks in a concurrent fashion. These threads communicate and coordinate with each other via the shared data to accomplish the desired logic.

Do **NOT** implement critical sections in this assignment for threads to access the shared data. Do **NOT** use semaphores to ensure the precedence constraint of executing threads. You will have an opportunity to do those in a later assignment after the concepts of critical sections are covered as part of the inter-process communication topic.

Be sure to read the reference on POSIX threads as well as the related sections in **FAQ** under Canvas module "Course Resources – Programming".

Functionality

In this assignment, you would use separate POSIX threads to accomplish similar flow of operations as what you wrote for the previous assignment, with added functions for using optional command line arguments and displaying task progress.

Your program starts from the **main** thread, which reads and processes command line arguments, creates and initializes a shared data structure to be shared between the main thread and worker threads, then spawns two worker threads to carry out the following sequence of logic in that order:

1. ***populatetree*** thread for reading words from the dictionary source file and populating a dictionary tree, and
2. ***countwords*** thread for reading words from a test file, and for every word read in from the test file:
    a. search and count the number of words in the dictionary tree that **start** with this word,
    b. if the count is equal to or greater than the number specified by an optional command line argument (default is 1 if not specified), **print this word and the count (separated by a space)** to a file named **countwords_output.txt**, **ONE line per word**, in the exact format as:
    hour 10
    'hour 10' here means for the 'hour' read from test file, there are 10 words in the dictionary tree that start with 'hour'.

The main thread **must spawn** the populatetree and countwords threads at the same time, then let the two worker threads coordinate with each other to ensure the order of execution.

While the worker threads carry out their respective task, the main thread monitors and **displays the execution progress** of each worker thread.

Note all communication, coordination / synchronization among the threads is accomplished by using the shared data, again, do NOT use critical sections or any other language construct (such as semaphores, or monitors, etc.) for facilitating thread communication and coordination.

Specifications

- Compilation and execution
- User Interface
- Output file countwords_output.txt
- Shared data structure among threads
- Display of the progress bar
- Code structure

Compilation and execution

- You must create and submit a Makefile for compiling your source code. Refer to the Programming page in Canvas for makefile help.
- The make file must create the executable with a name as **dicttest.**
- Your code **must compile and run correctly on edoras.sdsu.edu**, regardless of where you wrote, compiled, and tested your program.
- **Important**: Note when using C library function getopt() for processing optional command line arguments, if you programs in C, **getopt** does not play well the C99 compilation flag. Changing the makefile to use **"-std=gnu99"** or **"-std=c11"** instead would correct the issue.

User Interface

The executable requires **two mandatory command line arguments** and **accepts** a few optional arguments.

Optional arguments:

-p N	N progress marks (either hyphen or #) for displaying 100% progress of a thread execution, default is 50 if not specified. Minimal number is 10. If a number less than 10 is specified, program should print to the standard output:

Number of progress marks must be a number and at least 10
then exit.

-h N place a hash mark “#” in the progress bar every N characters, default is 10 if not specified. If an out-of-range number (<=0 or > 10) is specified, program should print to the standard output:
Hash mark interval for progress must be a number, greater than 0, and less than or equal to 10
then exit.

-n N print word and its count (refer to 2. b in Functionality) to an output file (countwords_output.txt) only if the number of dictionary words starting with the word is equal to or greater than N, default is 1 if not specified.

Note:
Error checking on optional arguments would **first check -p, then -h**. It would stop and exit at the first error; for example, if the program detects an out-of-range number from -p N, it would stop and quit.

Also, **optional** arguments are processed **before mandatory** arguments, see FAQ for code examples of processing command line arguments.

Mandatory arguments:

- The **first argument** specifies the file path to the source vocabulary text file that provides the words for building the dictionary tree.
    - Use **dictionarysource.txt** file as the source vocabulary text file for testing.
    - Appropriate error handling should be present if the file is not existent or cannot be opened. It must print to stdout the following error messages:
      Unable to open <<dictionarysource.txt>>
- The **second argument** specifies the file path to the text file that provides the words to be searched in the dictionary tree.
    - **testfile1.txt** is given for your testing.
    - Auto-grading on gradescope will use all or **part** of **testfile1.txt**.
    - Appropriate error handling should be present if the file is not existent or cannot be opened. It must print to stdout the following error messages:
      Unable to open <<testfile1.txt>>
- **Your program should have minimal error checking** on whether the mandatory command line arguments are provided from execution, fail gracefully when there is a wrong number of arguments.

Examples:

```
./dicttest dictionarysource.txt testfile1.txt -p 40 -h 5 -n 3
```

- populatetree thread reads words from dictionarysource.txt to populate the dictionary tree
- after the execution of populatetree thread, countwords thread reads words from testfile1.txt, searches and counts words in the dictionary tree, and prints to the **countwords_output.txt** file
  - For each word read from testfile1.txt, only print it and its count if the count of dictionary words that start with the word is equal to or greater than 3 (from -n optional argument)
  - A sample output file **countwords_output.txt** from the above command line execution is given.
- main thread displays a progress bar along the execution of each thread, the full progress would display 40 (-p optional argument) progress marks. Most characters (marks) are hyphens (-), but every 5th progress mark (-h optional argument) should be displayed as a number sign (#).

Execution without specifying optional arguments would use defaults as designated above. For example, the following execution

```
./dicttest dictionarysource.txt testfile1.txt
```

would be equivalent to

```
./dicttest dictionarysource.txt testfile1.txt -p 50 -h 10 -n 1
```

Output file countwords_output.txt

A sample output file from the execution of

```
./dicttest dictionarysource.txt testfile1.txt -p 40 -h 5 -n 3
```

command line execution is given, see above.

Shared data structure

The shared data structure among the three threads is used for:

1) the worker threads to update ongoing execution progress of their logic: use the number of characters read from the file and processed by the worker thread as the progress indicator,
2) the main thread to read and display the progress of worker thread executions,
3) ensuring the precedence constraint of thread logic executions, i.e., populatetree thread logic must happen before the countwords thread logic, and
4) ensuring the proper exit of the threads.

The following **EXEC_STATUS struct** is recommended to be used in your implementation.

Follow the **code comments** as hints for your implementation.

Important: Of the data members in the EXEC_STATUS struct, **you MUST use** the following member **as the progress indicator** for tracking the progress of the thread execution (see comments there)

```
long  *numOfCharsProcessedFromFile[NUMOFFILES];
```

```
/**
 * Shared constants and data structures among threads
 **/

#define NUMOFFILES 2
#define DICTSRCFILEINDEX 0
#define TESTFILEINDEX 1

/* default number of progress marks for representing 100% progress */
#define DEFAULT_NUMOF_MARKS 50

/* minimum number of progress marks for representing 100% progress */
#define MIN_NUMOF_MARKS 10

/* place hash marks in the progress bar every N marks */
#define DEFAULT_HASHMARKINTERVAL  10

/* default minimum number of dictionary words starting from a prefix for printing or
writing to the output */
#define DEFAULT_MINNUM_OFWORDS_WITHAPREFIX  1

/* Common data shared between threads */
typedef struct {

  /**
   * root node of the dictionary tree
   */
  dictentry *dictRootNode;

  /**
   * parameters for printing progress bar
   */
  int numOfProgressMarks;
  int hashmarkInterval;

  /**
   * print a word and its count to the output file only if the
   * number of dictionary words starting from the word is equal to or
   * greater than this number
   */
  int minNumOfWordsWithAPrefixForPrinting;

  /**
   * filePath[0] - file path for the dictionary vocabulary file
   *              providing words to populate the dictionary tree
   * filePath[1] - file path for the test source file
   *              providing words to be used for testing
   */
  const char  *filePath[NUMOFFILES];

  /** store total number of characters in files, total word count in files
   * totalNumOfCharsInFile[DICTSRCFILEINDEX]
   *   - number of total chars in the dictionary vocabulary file.
   *     use stat, lstat, or fstat system call
   * totalNumOfCharsInFile[TESTFILEINDEX]
   *   - number of total chars in the test file
   *
   * Hints: see man stat, lstat, or fstat to get the total number of bytes of the file
   * stat or lstat uses a filename, whereas fstat requires a file descriptor from a
   * low-level I/O call: e.g. open. If you are using high-level I/O, either use stat
   * (or lstat), or open the file first with the low-level I/O, then call fstat,
   * then close it.)
   *
   * Important: assume all text files are in Unix text file format, meaning, the end
   * of each line only has a line feed character. The stat, lstat, fstat would include the
   * count of the line feed character from each line.
   */
  long  totalNumOfCharsInFile[NUMOFFILES];

  /**
   * Use numOfCharsProcessedFromFile to track ongoing progress of
   * number of characters read in from files and the subsequent thread logic.
   * We will refer to the long integer to which this variable dereferences as the
   *  progress indicator.
   *
   * This progress indicator is updated by worker threads, and used by the main
   * thread to display the progress for tracking the execution of the worker threads
   *
   * Important: this number can be incremented by the number of characters in
   *                  the line that is being read and processed, plus one to include the
   *                  line feed character at the end of each line of a Unix text file.
   *                  Do NOT convert the text files to a Windows DOS format.
   *
   * File is read in line by line, for each line read in:
   *    1) tokenize the line by delimiters to a word array, with each separated word:
   *        insert the word to the dictionary tree or search, count, and print the word
   *        in the dictionary tree, and then increment the word count of the file:
   *        wordCountInFile (see below)
   *
   *    2) update the numOfCharsProcessedFromFile by incrementing it by the
   *        number of characters in the line, plus one to include the line feed
   *        character at the end of each line of a Unix text file.
   *
   * numOfCharsProcessedFromFile[DICTSRCFILEINDEX]
   *   - number of chars read in and processed from
   *     the dictionary vocabulary file
   * numOfCharsProcessedFromFile[TESTFILEINDEX]
   *   - number of chars read in and processed from
   *     the test file
   */
  long  *numOfCharsProcessedFromFile[NUMOFFILES];

  /**
   * wordCountInFile[DICTSRCFILEINDEX]
   *   - number of total words in the dictionary vocabulary file.
   * wordCountInFile[TESTFILEINDEX]
   *   - number of total words in the test file
   */
  long  wordCountInFile[NUMOFFILES];

  /**
   * completed flags indicate the completion of the thread logic, one
   * for the populatetree thread, one for countwords thread

   * Important: the completed flag of populatetree thread may be
   *                  used to force the precedence constraint that the populatetree
   *                  thread logic must be executed before the countwords thread
   *
   * taskCompleted[DICTSRCFILEINDEX]
```

```
  *   - boolean flag to indicate whether the tree population
  *     thread has completed the task: read words from the
  *     dictionary source file and populate the tree
  *
  *     important: you may want to set the completed flag for the
  *                     populatetree thread to true only after the main thread fully displays
  *                     the progress bar upon the completion of the thread logic
  *
  * taskCompleted[TESTFILEINDEX]
  *   - boolean flag to indicate whether the counting words
  *     thread has completed the task for reading and processing all words from the
  *     test file.
  */
 bool  taskCompleted[NUMOFFILES];
} EXEC_STATUS;
```

Display of the progress bar

The main thread monitors and displays the progress of each thread execution.

For displaying the progress bar along each thread execution, you may use a loop to compute the percentage of the task that has been completed and add to **a progress bar** representing the amount of progress that has been made. **Most characters are hyphens (-), but every Nth progress mark should be displayed as a number sign (#)**.

You may use this formula to calculate the percentage of the execution progress (refer to the variables in the above shared data structure spec.):

(double) *numOfCharsProcessedFromFile[DICTSRCFILEINDEX**] /**
(double) totalNumOfCharsInFile[DICTSRCFILEINDEX**]**

The progress bar starts from zero marker character. When new marker characters (- or #) need to be added, print them **without** a line feed character, so that the user will see a smooth progression of progress bar on their terminal. Note that putchar()/printf()/cout() typically buffer their output and only make a system call for output once the buffer is full. Request that they be printed immediately by using fflush(stdout) for C or cout.flush() for C++.

The loop for displaying progress bar would end when the percentage of progress reaches 100%. Remember that it is possible that you may need to print more than one marker at a time if more than an additional marker of the task has completed since the last time that the main thread executing the displaying progress logic was scheduled.

Each thread (populatetree and countwords) would also count the number of words during reading and processing their respective file, and update the wordCountInFile[] in the shared data. At the end of the thread execution, the wordCountInFile[] would have the total number of words in the file that was processed. After the full progress bar is printed, a new line feed should be printed, and the word count of the file will be printed with a line feed at the end.

A sample session should appear as follows; you start the execution of your **`dicttest`** program from the console prompt:

```
edoras> ./dicttest dictionarysource.txt testfile1.txt -p 40
-h 5 -n 3
```

The output format of your program to the standard output **should be EXACTLY the same as** below:

```
----#----#----#----#----#----#----#----#
There are 116687 words in dictionarysource.txt.
----#----#----#----#----#----#----#----#
There are 125322 words in testfile1.txt.
```

Code structure

You will be **required** to **separate** the logic of main thread, populatetree thread, and countwords thread to separate code files, although you are certainly encouraged to use more if you see your program becoming unwieldy.

Turning In to Gradescope

Make sure that all files mentioned below (Source code files and Makefile) contain your name and Red ID! **Do NOT compress / zip** files into a ZIP file and submit, submit all files separately.

- Source code files (.h, .hpp, .cpp, .c, .C or .cc files, etc.)
- Makefile
- Single Programmer Affidavit (must be a text or pdf file) with your name and RED ID as signature
- Do NOT submit any .o files or test files

Programming references

Please refer to Canvas Module "Course Resources – Programming" (particularly the FAQ) for coding help related to:

- Process command line arguments

- Create and use POSIX pthreads, and how to link the pthread library for compilation:
    a. You must include the header file pthread.h.
    b. You must link the library (-lpthread) at the end of the compilation line. If you do not do this, your program will not work although it will compile without any warnings.
- Code structure – what goes into a C/C++ header file
- Many other c/c++ related coding practices

Grading

Passing 100% auto-grading may NOT give you a perfect score on the assignment. The structure, coding style, and commenting will also be part of the rubrics for the final grade (**see Syllabus Course Design - assignments**). Your code shall follow industry best practices:

- Be sure to comment your code appropriately. Code with no or minimal comments are automatically lowered one grade category.
- Design and implement clean interfaces between modules.
- NO global variables.
- NO hard code – Magic numbers, etc.
- Have proper code structure between .h and .c / .cpp files, do not #include .cpp files.

All test data are given to you, **any hardcoding** to generate the correct output without implementing the required specifications will automatically **result in a zero grade** of the assignment.

Academic honesty

Posting this assignment to any online learning platform and asking for help is considered academic dishonesty and will be reported.

An automated program structure comparison algorithm will be used to detect code plagiarism.

- The plagiarism detection generates similarity reports of your code with your peers as well as from online sources. We will also include solutions from the popular learning platforms (such as Chegg, etc.) as part of the online sources used for the plagiarism similarity detection. Note not only the plagiarism detection checks for matching content, but also it checks the structure of your code.
- Refer to Syllabus for penalties on plagiarisms.
- Note the provided source code snippets for illustrating shared data structure would be excluded in the plagiarism check.