PROGRAM 4

Description

The goal of this program is to use POSIX threads to concurrently analyze the words in a set of text files. To do this, you will use a concurrent hash table facility that you constructed in Lab 8. For full credit, you will modify the concurrent hash table facility to support the optional use of thin locks instead of mutexes, as well as provide a way for the symbol table to store more information than just the frequency of occurrence of a symbol.

The goal of the text-file analysis program is not only to find the 20 most frequently occurring words (that you did in Lab 8), but also to find the longest word in in the entire set of ASCII files provided to the program.

We want to clearly understand how to separate out the function of an application (test driver in this case) with the implementation of the data structure.

You will need to write your own test driver in a file called prog4.c (this will be the main.c file that you were given in Lab 5 and modified to test your Lab 8 implementation). **You will need to submit symtab.c and prog4.c.**

Requirements:

- The symbol table module (symtab.c) must not create any threads, it simply supports threads by using appropriate locks. It's the responsibility of the test driver to create threads to execute a relevant worker function.
- The callback function must be present in the test driver (not in the symbol table module).
- Provision must be made in the callback function to also store the length of each symbol in the data area for each symbol in the hash table data structure. As far as the symbol table module is concerned, the data associated with a symbol is still a void *.

In order to support the options for the size of the table, number of locks and the use of thin locks easily during run-time, you will need to do this via environment variables that can be accessed through your main function. The library call getenv() will get the value of an environment variable (from Linux) and return its value as a character string. The environment variable we will use are:

TABLE_SIZE for the size of the table
NUMBER_OF_LOCKS for the number of locks
USE_THIN_LOCKS for using thin locks

Your program will be graded in the following manner:

A. Testing of Lab 8 functionality: 30%
B. Testing multiple files on the command line (each file name should be processed by a thread) with varying number of mutexes: 15%

C. Your test driver (text-file analysis program) has support for finding the longest word in a single or a set of files (test with my driver also): 20%
D. Thin lock support (for all the above cases): 20%
E. Valgrind check: 5%, Helgrind check: 5%
F. Code modularity, organization, documentation: 5%

Keep in mind that your code while adding new features (C, D) should still work correctly for all the previous test cases (A, B).

Also keep in mind that even if your tests cases pass on mimir, if your code doesn't really support the feature we are asking for, you will get a zero for those test cases (for example, your test driver accepts the environment variable for supporting thin locks but if your code doesn't really use thin locks you will get a zero for all thin lock tests).

If the requirements mentioned above or not met, your score will be reduced by 10% per unmet requirement even if your test cases pass.

All debug output must go to stderr, you must remove all unnecessary output before your final submission to avoid losing 5% of your total grade.

Output format:

Follow the same output format given to you from Lab 5 output for the 20 most frequently occurring words (in other words you shouldn't be changing that part of the program at all). Printing the longest word should be done at the very end with a line that says:

Longest=*actual-word*

If there are multiple words of the maximum length, report the one that is lexicographically smallest. If there is no word that is found in any file, leave the word name blank.

Sample output showing both frequency of words and longest word:

```
letter 2928
should 2633
lovelace 2047
myself 1488
before 1330
thought 1285
harlowe 1276
mother 1225
cannot 1200
little 1031
family 1015
indeed 982
clarissa 970
brother 955
gutenberg 954
belford 939
father 923
friend 922
project 907
against 900
Longest=irreconcileableness
```

The program should take a list of files on the command line. (You can assume there will be no more than 25 files). If a bad filename is provided (one that cannot be opened), print an error message to stderr and consider the file to have no words in it.

If the user does not specify at least one file to be processed, then terminate the program with an appropriate message to stderr.

Each file should be processed by a different thread. A single concurrent hash table should be used to collect information about the words in the files. For testing on your own, you can size the table, and specify the number of locks and the type of lock, to be whatever you think is best.

Once you have the multithreaded program working, then return to the concurrent symbol table facility and add support for the use of thin locks. That is, use the *atomic_flag* type, the *atomic_flag_clear* function and the *atomic_flag_test_and_set* function, which are all available in *stdatomic.h*, to build your own lock. Both functions take a pointer to the *atomic_flag* value that is to be manipulated.

- When the third parameter to *symtabCreate* is not 0, allocate space for an array of *atomic_flag* values, rather than an array of mutexes. Initialize the array of flags by calling the *atomic_flag_clear* function on each of them.
- Use the *atomic_flag_test_and_set* function to attempt to "lock" the flag. It will return false if the lock attempt is successful. If it returns true (meaning the thin lock is already locked), then call *pthread_yield* as a means to delay before re-trying to obtain the thin lock. (Loop until the thin lock is obtained.) You will need to add this prototype to the top of your file, because the function is not defined in *pthread.h*: int pthread_yield(void);.
- To unlock a thin lock, call *atomic_flag_clear*.

You should make sure that all malloc-ed memory is free before the program exits. We will use valgrind to make sure you do this.

Much like for Labs 5 & 8, a word in a text file is defined as below:

A word starts with a letter (either uppercase or lowercase) and continues until a non-letter (or EOF) is encountered. Only consider words that are at least six, and no more than 50, letters long. Non-words in the file should simply be ignored. Once a word is identified, convert all uppercase letters to lowercase before you process it.

Therefore, "elephant's" will be two words, "elephant" and "s", and since "s" is less than six letters long, it will be ignored. Likewise, "double-precision" will be two words, "double" and "precision".