CPSC 3500 Computing Systems
Winter 2018

Programming Assignment 2: Multithreaded Programming
Due: Friday, February 9 at 2:05pm

Part A – Introduction to Parallel Programming (35 points)

Write a multithreaded program (in the file `vowels.cpp`) that counts the number of time each vowel cumulatively appears in 20 different text files.

- The files are named "`file1.txt`", "`file2.txt`", … , and "`file20.txt`". They are available from the directory on cs1: `/home/fac/elarson/cpsc3500/pa2a`

Additional Requirements

- The program should create 20 threads, each thread should process a different input file. The 20 threads must be run concurrently.
- Your program must read the files from the directory `/home/fac/elarson/cpsc3500/pa2a`.
    - Do not assume the input files are in the current directory.
    - The directory and file names must be hard-coded into the program. Do NOT prompt the user for this information.
- Assuming error-free execution, the only output the program should produce is the cumulative frequency for each of the five vowels (a, e, i, o, and u). Do NOT print out the individual counts for each file.
- A vowel should be counted regardless if the vowel is lowercase or uppercase.
- Your program must use appropriate data structures and loops to avoid redundant code. Code that replicates something 20 times (such as 20 `pthread_create` calls) will be penalized.
- Global variables are not permitted! You must properly pass parameters into the thread function and use `pthread_join` to return the tally from each file. Using a global variable for this purpose will result in a minimum 12 point penalty.
- Locks or semaphores are not permitted.
- You must use Pthreads to implement the threading. You are not permitted to use the threading features present in C++ 11.
- No makefile is required for this assignment. The submission program requires the code to be in the file votes.cpp. It will compile your file using this command:

```
g++ vowels.cpp -std=c++11 -lpthread -o vowels
```

Error Checking

- If a pthread function returns an error, abort the program with an error message.

Assumptions

- Each input file will contain at least one line and that one line will contain at least one character. Beyond this, you cannot make any assumptions on how long the input file is.

Part B – Synchronization Programming (65 points)

For a database, there are three types of operations: *search* (read the database), *append* (append a record to the database), and *modify* (alter the database in some way). There are several concurrency rules that must be followed:

- *Search* operations can access the database concurrently with other *search* operations.
- *Search* operations can access the database concurrently with an *append* operation.
- Only one *append* operation can access the database at a time (it is an error if two *append* operations are concurrently accessing the database). As noted in the previous rule, the *append* operation can run concurrently with any number of *search* operations.
- *Modify* operations must have exclusive access to the database (no other *search*, *append*, or *modify* operations can run concurrently).

You will actually be creating two implementations for this problem:

a. Develop a solution using the rules as described.

b. Same as part (a) but with the additional restriction that a maximum of three search operations can access the database concurrently.

Implementation Instructions

To assist with debugging and testing, you will using a tool called MDAT (short for Multithreaded Debugging and Testing) designed for students who are learning to implement synchronization.

To start, download all of the files from `/home/fac/elarson/cpsc3500/pa2b`

The only files you are permitted to modify are:

`sectionsA.c`: Contains the functions that represent the entry, critical, exit, and remainder sections for part (a) of this problem. The critical section is where the hypothetical database is accessed. The program actually does not access any database. It is irrelevant as the goal of the assignment is to provide synchronization for this problem.

`sectionsB.c`: Identical to `sectionsA.c` except that it is used for part (b).

It is helpful to look at `main.cpp` to see how the different threads are set up. Towards the end file, you will notice that each thread carries out this loop:

```
for (int i = 0; i < numRounds; i++) {
  sectionEntrySection(accessType);        // Entry section
  sectionCriticalSection(accessType);     // Critical section
  sectionExitSection(accessType);         // Exit section
  sectionRemainderSection(accessType);    // Remainder section
}
```

The parameter `accessType` indicates the type of access. It is an `enum` type with constants SEARCH, APPEND, and MODIFY.

Your task in parts (a) and (b) is to implement the two functions responsible for implementing mutual exclusion: `sectionEntrySection` and `sectionExitSection`. Currently those functions are blank except for a single call to `mdat_enter_section`. You must leave this statement at the *beginning* of those functions. It will be necessary to add shared global variables – please declare them in the appropriate space and initialize them in the function `sectionInitGlobals`. The function `sectionInitGlobals` is called only once (before any threads are created) at the beginning of the program. The other two functions `sectionCriticalSection` and `sectionRemainderSection` are not to be modified.

Further use of MDAT is described in the MDAT Guide.

Additional Requirements

- The program must not implement a more restricted synchronization scheme that captures the rules provided here. For example, restricting the database to a single thread, easily implemented using a single lock, is too restrictive. Trivial solutions, such as using a single lock, will receive very low scores.
    - MDAT will warn you if the solution has the potential to be overly restrictive. Run your program using `run-mdat.py` with at least six threads. Since the solution does not have to be fair, it is possible you will get the warning message on many runs (especially with small thread counts). It is only a problem if you get the warning for EVERY run.
- Your implementation may only use the MDAT synchronization functions. MDAT only supports locks and semaphores and their basic operations.
- You are not allowed to modify code outside `sectionsA.c` and `sectionsB.c`. The submission script will compile your code use the original Makefile and original copies of the provided code, ignoring modifications made outside these files.

Error Checking

No error checking is needed for this part of the assignment.

Implementation Hint

It is recommended to implement part (a) first. Once you are happy with part (a), then start working on part (b). You may want to start with a copy of part (a) instead of starting from scratch:

```
cp sectionsA.cpp sectionsB.cpp
```

Grading

Grading is based on the following rubric and is primarily based on functionality. Failing to meet additional restrictions can incur further penalties in addition to any deductions due to functionality (your total deduction cannot exceed the number of points for that part).

Part A (35 points)

vowels	*35 points*
• Correct behavior	(no penalty)
• Incorrect output	-5 (or more)
• Program crashes or ends before displaying counts	-25 (or more)

Additional restrictions:

• Threads do not run concurrently	-4
• Used a lock or semaphore	-4
• Did not abort program if `pthread_create` has error	-4
• Did not read files from specified directory	-5
• Did not use a loop to create or join threads	-6
• Used a global variable	-12
• Did not create threads (single-threaded version)	-25

Part B (65 points)

synchronization	*65 points*
• Correct behavior	(no penalty)
• Mutual exclusion violation detected from MDAT	-15 (or more)
• Mutual exclusion violation not detected from MDAT	-3 (or more)
• Allow more than three concurrent accesses (part b only)	-10
• Deadlock	-12 (or more)

Additional restrictions:

• Overly restrictive solution that only allows one access at a time	-25 (up to -55 if blatant)

Submitting your Program

For part A, your program must reside in a single file named `vowels.cpp`. For part B, the only files to be submitted are `sectionsA.c` and `sectionsB.c`. All three files must be in the same directory before submitting (none of the other files are necessary).

On `cs1`, run the following script in the directory with the three files listed above:

```
/home/fac/elarson/submit/cpsc3500/pa2_submit
```

This will copy the files associated with this assignment to a directory that can be accessed by the instructor. Please be sure to name the file correctly or the submission program will not work.

The submission program will attempt to compile your program. For Part B, the script will use the original Makefile and the original copies of the provided code. Modifying those files could cause your program to not compile or be functionally different. Your program must properly compile or the submission program will reject your program. Programs that fail to compile will not be graded. Only the last assignment submitted before the due date and time will be graded. *Late submissions are not accepted and result in a zero.*