Program1

Write a sloppy counter simulator. After calling **make**, which should create a **sloppySim** executable file, your code should take the following command line arguments in the following order:

sloppySim <N_Threads> <Sloppiness> <work_time> <work iterations> <CPU_BOUND> <Do_Logging>

The arguments are

- Expected to be in this order (20 points – violating argument order will make grading yours much more time consuming)
- Allow skipping. If number of arguments is 3 then all arguments after the third will be set to default.
- If there is an error parsing the input, print out a message giving the order.
- Note the defaults

<N_Threads>: The number of threads to use. You must create actual threads and have them do the simulation; if you don't create and use the threads you get 0 points for this assignment. You may have *one* more thread for the main thread, which may also handle setup and logging. Use Pthread/ POSIX Threads. Defaults to 2 threads.

<Sloppiness>: How many events to do before updating a global counter. Defaults to 10.

<work_time>: The *average* work time in milliseconds. The time spent "working" should be the in the range 0.5*work_time <= time <= 1.5*work_time, *uniformly at random*. See CPU_BOUND. You should test at some points with 0 and a sloppiness of 1 so that you get a lot of contention on the global counter to ensure the locks/semaphore are setup properly. Read in as an integer without the unit; defaults to 10ms.

<work_iterations>: The number of iterations per thread. When all threads are done, the global count should be N_Threads*work_iterations if work_iterations is a multiple of Sloppiness. Your code should be able to know when stop the simulation. Defaults to 100.

<CPU_BOUND>: "**true**" or "**false**".

- If true, you need to make each thread "CPU bound", meaning it needs to work for approximately work_time on a CPU (even if it is just a busy wait to see how long the thread has been actually working). See https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9 . In the linked article this is called "CPU time", where time spent *not* running on a CPU does not count.
- If false, then each thread can use "Wall time" and should probably just wait. This will simulate I/O-bound work.
- What do you expect to happen if the number of threads exceeds the number of cores on the machine for each scenario? You don't need to answer this explicitly but your testing.txt file should reflect the answer. Defaults to "false".

<Do_Logging>: "**true**" or "**false**". Defaults to "false". Print out the current settings for N_Threads, Sloppiness, etc. Then, write out to console some logging information, perhaps 9 or 10 times and the final global counter variable in work_time*work_iterations / 10 millisecond intervals. I suggest using the main thread to read all the buckets and the global counter. Be careful not to create contention with a new lock or semaphore – a minor race condition reading the local counts will be simpler and okay. Testing code may redirect with '>'. Defaults to "false".

Deliverables. Place all files in a zip folder (make sure what you upload unzips in Ubuntu):
(70) C/C++ source code and a Makefile.

- You need to ensure it compiles on the department lab machines (Ubuntu Linux).
- -15%-100% of total score if we have to do more than call **make**.
- Your name in a comment at the top of the file with the main function.

(10) A Readme.txt or README.md file of some sort.
(20) A testing.txt file: Use your logging function and perhaps some other Linux functionality like the **time** command to test your implementation and demonstrate that it works and give a light explanation. At a minimum you must test for a small number of threads, a large number (~30, probably at least two to four times as many cores as will be on whatever machine we use), and both versus CPU and I/O bound. It should demonstrate scaling, appropriate for CPU and I/O bound. We will run our own scripts to validate as well. (You might also write a little script to do this in Python or bash).