Fall 2019
CS 370 – Operating Systems

Programming Project #2
Assigned on Tuesday, October 1st

DUE: October 22nd (Tuesday)11:59pm

Total 120 points

Note: submit your code/solution 'tarball' on cardiac and leave a WebCampus note by the due date. Late submission must indicate in the note how many allowance days you are going to use.

0. General instructions

As with project 1, your work should compile/execute on cardiac. You are free to work on this project on a different machine, but you have to test your code on cardiac.

cardiac (cardiac.cs.unlv.edu) is the Linux server dedicated to CS 370 projects. You must use cardiac instead of bobby. Your work should compile/execute on cardiac. You should have account on cardiac already. If not, please go to http://tux.cs.unlv.edu/accounts.html to request an account.

You need to turn in your work (i.e., code, binary, makefile, script, text files, etc) in the directory as follows:

```
proj2/
```

You should make a tarball for the entire project and turn in the tarball with a descriptive name. E.g., `$ tar –cJvf cs370_proj2_MyName.tar.xz ./proj2/`

A special 'drop box' directory has been set up on cardiac. You can now just copy your work (i.e., tarball file) into that directory from shell command prompt.

The directory is located at `/home/cs370-1003/` (This directory is write-only – you only have write permission).

After you create your tarball file for your project, you can just copy the tarball file into the directory as shown below:

```
[student@cardiac ~]$ cp cs370_proj2_StudentName.tar.xz  /home/cs370-1003/
[student@cardiac ~]$ _
```

(Make sure you have your full name as part of your tarball file name.)

You then need to log in to WebCampus and submit a note there, which should contain:

- The name of the file you submitted
- The late-days you have used in this project and the remaining late-days
- Special instruction, if any, that I should know to compile/run your program

You may add any additional information to the note regarding your project.

It's important for you to submit the note on WebCampus as this will allow WebCampus to record that you have submitted your project.

In case you need to update your submission, you can just overwrite the old file in the directory. If you don't remember old file name, that's OK, I'll use the latest one that's submitted before the deadline. Just make sure you have your name as part of the tarball file name.

Also note that both tarball on cardiac and the note on WebCampus should be submitted before the deadline. The timestamp of tarball file on cardiac should be before the deadline (or within extended deadline if you're using late days). I'll be a bit reasonable for the note on WebCampus, so I'll tolerate a few minutes late for the note.

1. Write merging for small log entries (120 points)

Purpose: to understand producer-consumer semantic in a multithreaded environment with performance implication.

Description: Your task is to improve a multi-threaded program that frequently writes small entries to a shared file by introducing a layer that merges in memory before writing to file to reduce the number of expensive I/O operations.

Imagine a program which writes to file small log entries very frequently. The program is multi-threaded and each thread writes log entries to a common log file. To simplify implementation, each thread just writes log individually with blocking I/O interface. Even though threads write to the shared file concurrently without any locking, this is correct because the OS guarantees the atomicity of individual `write`() operation.

But this simple (i.e., dumb) implementation has negative performance implication because of the inefficiency of the large number of small I/O operations. Performance would be improved if we gather log entries in a memory buffer first, and then write the gathered entries in a single I/O. Of course, this 'gathered-write' can be implemented by changing application to explicitly maintain memory buffers. However, this requires significant modifications of application logic.

A less intrusive buffering solution is to use the producer-consumer paradigm in multi-threaded program. The idea is to have the log-writer threads to write to a shared memory buffer, and have a log-flusher thread flush the memory buffer to the file. The shared memory buffer acts as the log-gathering space to reduce the number of I/O operations.

This data structure is similar to the BBQ we studied in class, but the differences are that the consumer thread consumes multiple entries at a time, and consumption involves long-latency I/O operation (at least trips to OS kernel).

Details: Two c files are provided as reference. The `logwriter_dumb.c` simulates the situation where 4 threads repeatedly write 32-byte log entries 32K times each to a shared log file named '`logfile.txt`'.

The '`logwriter_gather.c`' partially implements the ring buffer. Your task is to fully implement the ring data structure and the necessary functions. **See the source code and comments in the code for details**.

Upon successful run, the program should drop a 4MB `logfile.txt`. The program should finish less than a few seconds on cardiac – YOU SHOULD BE PREPARED TO KILL YOUR PROGRAM if your program runs longer than a few seconds. MAKE SURE YOU DON'T CREATE A HUGH LOG FILE when you're developing your program. DELETE ANY BIG LOG FILE YOU ACCIDENTALLY CREATED.

What to turn in:

- Your source code logwriter_gather.c with DESCRIPTIVE/HELPFUL COMMENTS
- Your binary file logwriter_gather
- Makefile

What NOT to turn in:

- The 4MB logfile.txt (→ this file will be generated by when I test your code)

Grading:

- 30 points for demonstrating you put enough efforts in coding (ring buffer implementation, using mutex/condition variables, compiles OK, etc.)
- 30 points if program works partially.
- 30 points if program works correctly all the time.
- 10 points for elegance (succinct and crisp) and completeness (error/corner case handling).
- 20 points for comments helpful to understand why your code works.