Assignment # 3
Concurrent Data Structures
[100 points]

CSE 511 Fall 2022

Assigned: Thursday, November 17, 11:59 PM
Due: Thursday, December 08, 11:59 PM

1. Policy

You will work as individual to solve the problems for this assignment. Academic integrity is crucial. Please read the syllabus regarding the academic integrity policy, which contains a list of things that are not permitted! Note that academic integrity issues may arise when helping other students or looking for an answer online. You are not allowed to use any online sources other than official documentation that describes pthreads, C11 atomics, and other official manuals. For example, looking for an implementation of a lock-free data structure is not allowed, but reading manuals about compare-and-swap (compare-and-exchange), pthreads mutexes (which provide generic information that is not directly linked to a specific data structure implementation) should be fine. **When in doubt, please ask the teaching staff first!**

2. Provided Code and Required Changes

You are given a benchmark to which you need to integrate your data structure implementations.

Please get the code and create your own repository similar to the previous assignments by following this link: https://classroom.github.com/a/uBT6S-5T

main.c is the main file which instantiates different data structures. By default, you only have a "dummy" queue as an example. Please follow that example for your implementation. You will have to add other queues/stacks and register them with the benchmark as necessary.

dummy.h a "dummy" queue example. You will have to create similar **header** files for your queue/stack implementations and add those header files to main.c.

bench.c is the actual benchmark. Do not modify it! We will use the original version that we provide when grading and discard any modifications that you make. This file is compiled separately through Makefile, do **not** include this or any other .c files to your code by using #include!

smr.c is an epoch-based memory reclamation implementation. Do not modify it! We will use the original version that we provide when grading and discard any modifications that you make. This file is compiled separately through Makefile, do **not** include this or any other .c files to your code by using #include!

smr.h is the header for epoch-based memory reclamation that you have to use/include **only for the questions that need memory reclamation.** You can only use smr_start_op(), smr_end_op(), and

smr_retire(). No other functions can be called from your code! Do not modify this file! We will use the original version that we provide when grading and discard any modifications that you make.

3. Compilation and Execution

You have to use a normal Linux distribution. Note that we do not support any other installation (e.g., MacOS, Windows WSL, etc), and cannot advise on whether anything other than Linux can be used. The Linux instance can run in VirtualBox, VMware, etc, but please make sure to allot sufficient memory to the Linux instance. Please make sure that your Linux installation has clang 7.0 or higher installed. In Ubuntu, you can type:
sudo apt-get install clang

[Note that using clang is crucial since gcc lacks proper support for 16-byte C11 atomics (i.e., cmpxchg16b).]

To compile the code, please run
make

To run the benchmark, please run
make test

Whenever you modify any header file (e.g., dummy.h), do not forget to clean up before running **make**:
make clean

You can install and use **gdb** if you would like to use the debugger. We refer you to the gdb cheatsheet [1]. Generally speaking, you will likely need only a limited subset of commands: ‘run’, ‘print’, ‘bt’ (backtrace), ‘frame’ (to select a specific function from the backtrace), ‘info registers’. (To switch between different threads, please use ‘info threads’ and ‘thread <thread_num>’.) Using gdb is completely optional, you can complete the assignment without it, but it can be helpful when you want to avoid using manual debugging through printf. It even allows to iterate through the (node) list in the queue/stack at runtime and print pointers/values in each node.

Generally speaking, you need to enable debugging symbols for gdb. **To compile the code with debugging symbols, please modify Makefile and add the ‘-g’ parameter to clang,** which enables debugging symbols. Sometimes you may also want to change optimization options (while debugging), but please keep in mind that changing the optimization level can change the executable code entirely, and certain bugs will “disappear” (e.g., will not manifest anymore) when you run your code through gdb.

Please do not attempt to hide any bugs by changing optimization options, introducing any delays, or using any other tricks. We will review your code carefully and will use the default version of Makefile to test your code.

We will also use the default parameters in main.c (i.e., 16 threads and 100000000 operations) to test your code. Please note that we are still allowed to change the default parameters when grading if we have any doubts about your code. Thus, we recommend to both reduce and increase those parameters in your main.c to make sure that you test your code exhaustively.

4. Submission

Please read carefully the above statements!

You will receive ***zero*** points if:
- you break any of the assignment rules specified in the document
- your code does not compile/build
- your code is buggy and/or crashes
- the GIT history (in your github repo) is not present or simply shows one or multiple large commits; it must contain the entire history which shows the actual incremental work.

Please commit your changes relatively frequently so that we can see your work clearly. Do not forget to push changes to the remote repository in github! You can get as low as ***zero*** points if we do not see any confirmation of your work. Do not attempt to falsify time stamps, this may be viewed as an academic integrity violation!

Your submission can use the late policy, and we will calculate and use the better of the normal score and the late score with the late penalty.

Finally, include "report.pdf" in your github repository with a short description about what was implemented. Please be specific when describing what was (and was not) implemented but do not be too verbose. Please also provide **one screenshot** of a benchmark run which shows all results (i.e., throughput results for all questions that you have implemented).

Please be careful with memory management and avoid leaking memory. Memory must always be properly freed (lock-based approaches), retired (lock-free approaches), or recycled (lock-free approaches with ABA tags). **Please monitor the memory usage when running the benchmark (across all tests)** by running the **top** utility and observing how 'Free' changes in the 'MiB Mem' row while you are running the benchmark. Generally speaking, the amount of memory used should be small for lock-based implementations and when doing the ABA-safe recycling. With memory reclamation (smr.h), the amount of memory used will likely be within several hundreds of megabytes.

In your Canvas submission, please indicate the GIT commit number corresponding to the final submission. You should also specify your github username. The submission format in plain text <your_github_username>:GIT commit

For example, runikola:936c332e7eb7feb5cc751d5966a1f67e8089d331

5. Questions

5.1 [15 points] Implement a **lock-based** LIFO stack.

Please use only pthread_mutex for synchronization. Do not use CAS, TAS, or any other atomic operations. Use only one mutex per a stack instance. Please make sure to initialize the mutex correctly. The stack has to be implemented as a list of nodes. Use the 'top' pointer to point to the topmost node of the stack. The stack can be empty but it is not bounded (i.e., can never become "full" unless you run out of memory).

5.2 [15 points] Implement a **lock-based** FIFO queue

Please use only pthread_mutex for synchronization. Do not use CAS, TAS, or any other atomic operations. Use only one mutex per a queue instance. Please make sure to initialize the mutex correctly. The queue has to be implemented as a list of nodes. Use the 'head' and 'tail' pointers to point to the head and tail nodes of the queue. The queue can be empty but it is not bounded (i.e., can never become "full" unless you run out of memory).

5.3 [25 points] Implement a **lock-free** LIFO stack with memory reclamation

Please implement an unbounded lock-free stack using Treiber's approach that we discussed during our lectures. You are not allowed to use any locks (e.g., pthread_mutex). You are only allowed to use an ordinary 8-byte CAS. You should also use memory reclamation and return memory back to the OS through the epoch-based reclamation in smr.h.

5.4 [25 points] Implement a **lock-free** FIFO queue with memory reclamation

Please implement an unbounded lock-free queue using Michael and Scott's approach (with a special sentinel node) that we discussed during our lectures. You are not allowed to use any locks (e.g., pthread_mutex). You are only allowed to use an ordinary 8-byte CAS. You should also use memory reclamation and return memory back to the OS through the epoch-based reclamation in smr.h.

5.5 [10 points] Implement a **lock-free** LIFO stack with ABA tags and recycling

Please implement an unbounded lock-free stack using Treiber's approach that we discussed during our lectures. You are not allowed to use any locks (e.g., pthread_mutex). You are allowed to use both 8-byte CAS and 16-byte CAS. However, you are not allowed to use memory reclamation or return memory back to the OS. Instead, you should, effectively, create two stacks: alloc_stack and free_stack. alloc_stack will contain the actual stack elements, whereas free_stack will contain recycled stack elements (i.e., the elements that were previously deleted from alloc_stack and can be recycled for the future operations). Remember that you need to add the ABA tag for the **top** pointer but **not** for the next node pointers.

5.6 [10 points] Implement a **lock-free** FIFO queue with ABA tags and recycling

Please implement an unbounded lock-free queue using Michael and Scott's approach that we discussed during our lectures. You are not allowed to use any locks (e.g., pthread_mutex). You are allowed to use both 8-byte CAS and 16-byte CAS. However, you are not allowed to use memory reclamation or return memory back to the OS. Instead, you should, effectively, create two queues: alloc_queue and free_queue. alloc_queue will contain the actual queue elements, whereas free_queue will contain recycled queue elements (i.e., the elements that were previously deleted from alloc_queue and can be recycled for the future operations). Remember that you need to add the ABA tags everywhere: for **head**, **tail,** and every **next** node pointer.

References and Notes

[1]. GDB cheatsheet: https://cs.brown.edu/courses/cs033/docs/guides/gdb.pdf

[2]. POSIX mutex:
https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_mutex_lock.html
https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_mutex_init.html
https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html
https://man7.org/linux/man-pages/man3/pthread_mutex_destroy.3p.html

[3]. POSIX threads (just in case):
https://man7.org/linux/man-pages/man7/pthreads.7.html

[4]. C11 atomics:

Note that **volatile A * obj** (the first argument) in the documentation represents a pointer to the atomic variable and is actually **_Atomic(A) * obj** in C11. All other arguments typically represent ordinary types (i.e., without _Atomic). Note that compare_exchange (CAS) is somewhat enhanced: it also returns the previous value when CAS fails. Your implementations may or may not take advantage of that but please always remember that the corresponding second parameter changes when CAS fails. Also, use only the '**strong'** version of CAS and **avoid 'explicit'** versions of operations. Use the default versions which already enable sequential consistency.

https://en.cppreference.com/w/c/atomic
https://en.cppreference.com/w/c/atomic/atomic_compare_exchange
https://en.cppreference.com/w/c/atomic/atomic_load
https://en.cppreference.com/w/c/atomic/atomic_store