CSE113: Parallel Programming

Homework 2: Mutexes
Assigned: April 22, 2021
Due: May 6, 2021

Preliminaries

1. This assignment requires the following programs: `make`, `bash`, and `clang++`. This software should all be available on the provided docker.
2. The background for assignment is given throughout the lectures of module 2 and in the textbook: The Art of Multiprocessor Programming. There is a link on the course website for an online copy hosted by the library. I will provide reference to book sections when needed.
3. Find the assignment packet at https://sorensenucsc.github.io/CSE113-2021/homeworks/homework2_packet.zip. You might collect this from a a bash cli using `wget`. That is, you can run:

   ```
   wget https://sorensenucsc.github.io/CSE113-2021/homeworks/homework2_packet.zip
   ```

   Download the packet and unzip it. But do not change the file structure.
4. This homework contains 3 parts. Each part is worth equal points.
5. For each part, read the *what to submit* section. Add any additional content to the file structure. To submit, you will zip up your modified packet and submit it to canvas. If you have questions about the structure of your zip file, please ask!
6. It is okay to discuss docker questions and file structure questions with your classmates. Please don't share results until the week after the assignment has been posted.

1 Implementing Mutexes

In this part of the assignment you will implement two mutex variants. The first is the filter lock, an N threaded generalization of Peterson's algorithm. You can find the algorithm description in section 2.4 of the book. The second mutex you will implement is Lamport's bakery algorithm. It is given in section 2.6.

You will measure the throughput and fairness of each mutex using the skeleton C++ code provided in the packet. We reviewed the structure of the main function in the April 22 lecture. It repeatedly locks and unlocks a mutex on each thread for 1 second. It reports the throughput of the mutex overall (how many total mutex locks) and a histogram for how many times each thread obtained the mutex.

Your work will largely be constrained to the two mutex header files: `filter.h` and `bakery.h`. You must implement 4 functions:

- the constructor
- `init`: an initialization function that is called before threads are launched. It takes in the number of threads as an argument.
- `lock`: as described in lecture. It takes the thread id as an argument.
- `unlock`: as described in lecture. It also takes a thread id as an argument.

You must also provide the necessary private variables to implement the mutex. You should use atomic data types (only when required), and you must use the `store` and `load` methods to access memory through the atomics. *You are not allowed to use atomic RMWs in any part of your implementations!*

You are allowed to use the book as a reference, as well as the class lectures/slides. Please do not explicitly search online for C++ implementations of these locks.

You can gain confidence in your implementations by running them with the clang thread sanitizer. That is, add the following command line option `-fsanitize=thread`, and then executing the program. Your mutex implementations should execute without errors.

The make file produces three executables, one for each lock:

- `cpp_mutex`: which uses the C++ mutex object.
- `filter_mutex`: which uses the mutex from `filter.h`
- `bakery_mutex`: which uses the mutex from `bakery.h`

1.1 Experiments

Once your locks are implemented, you will run the three executables with various configurations and record the results:

- Run with as many threads as you have cores. Record the throughput of each mutex and record the variance in the number of times each thread obtained the mutex (for each mutex).
- Run with 32 times as many cores as your machine. Record the throughput and the variance like above (for each mutex).

1.2 Adding Yield

For the next part of the assignment, add a yield to the spin loop of both of your mutex implementations. This was discussed in the April 22 lecture.

Repeat the experiment with your new mutex implementations using yield.

1.3 What to Submit

You will submit completed header files for both the filter lock `filter.h` and the bakery lock `bakery.h`. Please submit the versions with the yield included. Like mentioned above, it is suggested that you test your code with clang thread sanitizer.

You will also submit a pdf report detailing your results. Record your throughputs and variances of each mutex in a table. Then report your results in two graphs: one graph for the throughput and one for the variance. The X axis is the different mutex implementations, the Y axis is the values you obtained. Write one or two paragraphs explaining your results, and how they compare to the C++ mutex.

Your grade will be based on 4 criteria:

- Correctness: Do your mutexes provide mutual exclusion.
- Conceptual: do your implementations use atomic operations correctly and implement the algorithms faithfully. Please comment your code.
- Performance: do your performance results match roughly what they should.
- Explanation: do you explain your results accurately based on our lectures.

2 A Fair Reader-Writer Lock

In the April 22 lecture, we discussed a Reader-Writer mutex implementation. The shortcoming is that it could potentially starve writers if enough readers continually accessed the mutex.

As you might have suspected, I have written a benchmark that does exactly that. It is a similar wrapper to Part 1, with the exception that we have 6 readers and 2 writers. The wrapper records the throughput of the readers and of the writers. You will notice that the writers obtains the mutex much less frequently than the readers.

Your job is to develop a scheme which provides more fairness to the writer. Your results should show a significant increase in the number of times that the writer is able to obtain the mutex. You should also notice that the readers will suffer in throughput.

Your implementation is constrained to `fair_mutex.h`. Your solution must not allow data conflicts in the critical section of the RWMutex, i.e. the writers must have exclusive access when they acquire the mutex. You can check this with Clang's thread sanitizer, similar to Part 1.

2.1 Experiments

Record the throughput of the readers and writers as provided by the original code. Then record the throughput of the readers and writers as reported by your modifications.

2.2 What to Submit

Please submit your modified in `fair_mutex.h`. Write up the results of your experiments in a pdf file and submit it along with the mutex file. Write one paragraph describing your solution. Write another paragraph analyzing your results.

Your grade will be based on 4 criteria:

- Correctness: Does your mutex provide mutual exclusion?
- Conceptual: does your implementations use a conceptually sound strategy to increase fairness to writers?
- Performance: does your solution actually provide more fairness for the writers?
- Explanation: do you explain your results accurately based on our lectures.

2.3 Bonus

For those of you looking for an additional challenge: try modifying the reader-writer implementation to obtain even higher throughouts (e.g. using backoffs, or even conditional variables). You can google for ideas, or look at Chapter 8 in the text book.

At this point I am not providing extra credit for bonus questions, they are simply for your enjoyment :)

3 A Concurrent Linked List

In the homework packet, I have provided a sequential stack implementation using a linked list implementation: your homework is to use C++ mutexes to make this structure safe to access concurrently.

You will be modifying this code, however, the structure of the stack must remain the same: that is, each operation starts at the beginning node and traverses the list until the end. As a further constraint, the only values that will be pushed are integers between 0 and 2, inclusive. Peek and pop return -1 when the stack is empty.

There is a wrapper benchmark file that calls the stack methods concurrently and records the throughput. You have three implementations you need to provide. Similar to part 1, the makefile will produce executables for each one.

3.1 Coarse-grained locking

In this implementation (`coarse_lock_stack.h`), you should add a mutex to the class's private variables. You should perform locking and unlocking for each of the three public methods.

3.2 RW locking

In this implementation (`rw_lock_stack.h`), you should use the C++ shared_mutex object. You should identify when you need to use the full lock, and when you can use the reader lock. Recall that the reader lock call is `lock_shared` and `unlock_shared`.

3.3 SwapTop

Building on your RW locking implementation, for the final file (`swaptop_stack.h`), you need to implement a new API function called `swaptop` (swap top). This function takes in an integer and swaps the top element of the stack with the new value. It should do this indivisibly, i.e. the pop and push aspects need to be protected in a single critical section.

This function can (and should) be optimized using a read lock. That is, there are parts of the function that can be efficiently implemented just with the read lock. As a hint, please recall that the stack contains only 3 possible values: 0,1,2.

The rest of this third implementation should be the same as the RW locking.

3.4 experiments

Run each of your 3 executables and record the throughputs of the different operations. Due to timing variations, please run each experiment 10 times and report the average, and comment on the variance you see during runs.

4 What to Submit

Please submit the three completed header files. Please report the results of your experiments in a pdf. Write 1 paragraph about your solution to SwapTop, and how you used the RW lock. Write 1 paragraph about your results.

All of your solutions should be free from data-conflicts. That is, they should pass Clang's thread sanitizer check when executing the wrapper. Your modifications to the stack should not effect its single threaded behavior (i.e. a push should add an element to the stack: a subsequent pop should retrieve the element).

Your grade will be based on 4 criteria:

- Correctness: Is your list conflict free and does it maintain sequential behaviors?
- Conceptual: Are the methods correctly locked with the different mutexes? Does the SwapTop function efficiently provide its functionality?
- Performance: do your performance results match roughly what they should? Do your RW locks provide higher throughput?
- Explanation: do you explain your results accurately based on our lectures?

4.1 Bonus

Develop a fine-grained locking scheme for the list, e.g. where each node is protected by a unique lock. This is more difficult to implement, but should provide even higher throughout.