CSE113: Parallel Programming

Homework 2: Mutexes
Assigned: January 21, 2022
Due: February 4, 2022

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-wi2022/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. If you need help, please visit office or mentoring hours. You can also ask questions on Piazza. You are allowed to ask your classmates questions about frameworks and languages (e.g. Docker and C++). You are not allowed to discuss technical homework solution details with them.
6. Your submission will consist of two parts: a zipped directory of code and a pdf report. Upload both files to Canvas when you submit. The contents of the report for each part are detailed in each section. The structure of the zipped directory is described at the end of the document.
7. The code contains empty `checker.h` files. These are stubs that we will fill in when grading. Do not remove them.

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 of the book.

You will measure the throughput and fairness of each mutex using the skeleton C++ code provided in the packet. The main function repeatedly locks and unlocks a mutex on each thread for 1 second. It reports the throughput of the mutex overall (how many total mutex lock/unlocks occurred) 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 mutexes.

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 execute the program. This is done for you if you run `make test`. 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. There are two results you will be gathering:

- **the throughput**: this is recorded as the `total number` value that is printed out. It records how many locks/unlocks occurred across all threads.
- **the fairness**: we need some way how to measure how fair the implementation is. For this, we will use a statistical metric called *coefficient of variation*[1]. While the math can be a bit tricky, the high-level idea can be expressed as:

  It shows the extent of variability in relation to the mean of the population.

[1] see the wikipedia page here: https://en.wikipedia.org/wiki/Coefficient_of_variation

There is an online calculator you can use to compute this statistic here: https://www.socscistatistics.com/descriptive/coefficientvariation/default.aspx

This metric gives a number you can use to compare fairness across your different implementations, even if they have different throughputs. The lower this number is, the more fair the mutex is.

You should run and report on the following experiments:

- Run with as many threads as you have cores. Record the throughput of each mutex and record the fairness using the coefficient of variance (for each mutex).
- Run with 16 times as many cores as your machine. Record the throughput of each mutex and record the fairness using the coefficient of variance (for each mutex).

These results should be gathered without thread sanitizer enabled.

1.2 Adding Yield

For the next part of the assignment, add a yield to the spin loop of both of your mutex implementations (the filter and bakery mutexes). This instruction be discussed in lecture during the week of Jan. 24.

Repeat the experiments with your new mutex implementations using yield.

1.3 What to Submit

The code component of your submission is the completed header files for the filter lock and bakery lock (`filter.h` and `bakery.h`). Please submit the versions with the yield included.

The report component consists of your mutex results, both the throughput and coefficient of variance. Please list your results in a table and provide two graphs: one graph for the throughput and one for the coefficient of variance. The X axis will be 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 as specified in this document) and did you implement the algorithms faithfully? Please comment your code.
- Performance: do your throughput/fairness results match roughly what they should.
- Explanation: do you explain your results accurately based on our lectures.

2 A Fair Reader-Writer Lock

We will discuss Reader-Writer mutex implementations in class lectures during the week of Jan. 24. The shortcoming of basic Reader-Writer mutex implementations is that they can potentially starve writers if enough readers continually access 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.

As two additional constraints: your solution cannot reduce the throughput of readers when there are no writers. Your solution cannot use conditional variables.

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

The code component is the completed `fair_mutex.h`. It will be zipped up with the rest of your code as described at the end of this document.

The report component consists of your results in a table and a graph; one paragraph describing your solution; and another paragraph analyzing your results. Consider using the coefficient of variance, similar to part 1.

Your grade will be based on 4 criteria:

- Correctness: Does your mutex provide mutual exclusion? That is, there should be no writer-writer overlaps in critical sections, nor writer-reader conflicts. Your mutex *should* allow multiple readers in the critical section.
- 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.

3 A Concurrent Linked List

In the homework packet, I have provided a sequential stack implementation using a linked list: 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 single 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.

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).

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 spread of values you see during runs.

4 What to Submit

The code component of your submission is the completed header files. They will be submitted as part of the zip file for the whole assignment (described at the end of the document).

In the report component please give the results of your experiments using a table and a graph. Write 1 paragraph about your solution to SwapTop, and how you used the RW lock. Write 1 paragraph analyzing your results.

Your grade will be based on 4 criteria:

- Correctness: Is your list free from data conflicts and does it maintain the correct behavior of a stack?

- 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?

Zipped file directory structure

Please submit your code zipped up in the following structure:

- Submit your code constrained to exactly the files discussed in the section. Do not rename the files. Only modify the files you are asked to. You can add functions, but do not change the signature of the functions that are provided.
- from the directory where you originally unzipped the packet, run:

  ```
  zip -r homework2.zip part1 part2 part3
  ```

- This will create `homework2.zip`. Please submit this file.
- you can double check your file fits the format we are expecting by copying `homework2.zip` and `check.sh` to a new folder. You should be able to run:

  ```
      unzip homework2.zip;
      bash check.sh;
  ```

  and not see any errors.

In addition to `homework2.zip`, please also submit a single pdf file including the report components of this assignment. Please submit this pdf as a separate file in canvas, and not as part of the zip.