CSE113: Parallel Programming

Homework 4: Barriers and Memory Models
Assigned: May 24, 2021
Due: June 7, 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 this homework is in the class material for Module 4, and in the book, chapter 17.
3. Find the assignment packet at https://sorensenucsc.github.io/CSE113-2021/homeworks/homework4_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/homework4_packet.zip
   ```

   Download the packet and unzip it. But do not change the file structure.
4. This homework contains 2 parts. Part 1 is worth 40% and part 2 is worth 60%.
5. For each part, read the *what to submit* section. Add any additional content to the file structure (e.g. the reports). 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. Please do not share code at all.
7. You should feel free to modify any part of the function that you are working on. In some cases I have left some loop structure in the code to guide you, but in some cases you may need to change loop bounds, increments, etc.

1 Barrier Throughput

In this part of the assignment, you will implement three different approaches to barrier synchronization. You will evaluate the throughput of each approach in two different configurations each. This is similar to the Mutex implementations in Homework 2.

The program in question performs a repeated *blur* operation on an array of doubles. A blur operation on an array computes an average for each index, using its immediate neighbors (i.e. the proceeding and preceeding index). In our implementation, we will not blur the boundary indexes,

i.e. the first and last indexes of the array. A blur operation can be completed in parallel by chunking the array indexes across threads.

A repeated blur is simply a series of blurs executed in sequence. Each blur operation relies on its predecessor being fully completed before starting. This can be achieved using barrier synchronization. The input and output array can be swapped at each blur (i.e. the output of a blur is in the input to the next blur). You can assume that there are always an odd number of repeats, so that the results will always be stored in the output. You can also assume a power-of-two size.

The specification of the repeated blur operation is as follows:

```
void repeated_blur(double *input, double *output, int size, int repeats) {
  for (int r = 0; r < repeats; r++) {
    for (int i = 1; i < size - 1; i++) {
      output[i] = (input[i] + input[i+1] + input[i-1])/3
    }
    double *tmp = input;
    input = output;
    output = tmp;
  }
}
```

Your task is to implement the repeated blur using 3 different approaches:

1.1 Spawn and join

For every blur, launch a set of C++ threads, and then join them. That is, you will launch/join threads `repeats` times. Implement this solution in `main1.cpp`. I have written a skeleton for you which takes in the number of threads as a command line argument. You will need to swap the input and output arrays in the main thread and update the arguments you provide to the threads. The Makefile will compile this program into an executable: `sjbarrier`.

1.2 Basic sense reversal barrier

Before starting this approach, please read chapter 17 (up to 17.4). Your task is to implement the sense reversing barrier. Please do not google solutions to this implementation. Please try to read the book and implement your understanding of the algorithm. Implement this barrier in `SRBarrier.h`. Modify `main2.cpp` to implement the repeated blur using this barrier. Your main function should launch threads once, and join threads once. Use a call to `barrier` to synchronize threads between blur operations. The Makefile will compile this program into an executable: `srbarrier`.

1.3 Optimized sense reversal barrier

In this approach, implement an optimized variant of the sense reversing barrier of the previous section by adding relaxed peeking and yielding (similar to the mutexes). Implement this optimization in `SROBarrier.h`. I have added preprocessor directives to compile `main2.cpp` using this barrier, so there is no need to change a main file for this approach. The Makefile will compile this program into an executable: `srobarrier`.

1.4 Experiments

Please time the computation for the repeated blur function in each approach using the `chrono` C++ library, similar to Homework 1.

Each of the executables takes a command line argument to specify the number of threads to run with. For each approach, time with the following number of threads: 1, cores, cores $\times$ 2. That is one thread, as many threads as you have cores, and 16 times as many threads as you have cores. Record your timing experiments.

You are encouraged to run your timing experiments multiple times and take an average. You are encouraged to run your experiments with as few other programs running as possible.

You are encouraged to run your program with the thread sanitizer to check for data-races. You are also encouraged to check your output using a reference computation, although you should ensure not to time results for either of these validation checks.

1.5 To submit

Please submit all existing files in the directory, including a completed `main1.cpp`, and `main2.cpp`, `SRBarrier.h` and `SROBarrier.h`. Please include a report that includes a graph of your results. Please write two paragraphs: one explaining your implementation, and a second one to explain your results. Include your report as a pdf in this directory.

As with the rest of the assignments, your grade will be based on:

- Correctness: Do your approaches produce the right results for the repeated blur?
- Conceptual: do your implementations synchronize threads correctly?
- Performance: do your performance results match roughly what they should.
- Explanation: do you explain your results clearly based on our lectures and the book readings?

2 Store buffering on x86 processors

This part of the homework deals with observing relaxed behaviors on the x86 processor. As discussed in lecture, x86 implements a relatively strong memory model, known as TSO (or total store order). You will first work to observe some TSO behaviors using the store buffering litmus test.

Recall that the store buffering litmus test is as follows:

```
Thread 0:
store(x,1);
%var0 = load(y);

Thread 1:
store(y,1);
%var1 = load(x);

Check: can %var0 == %var1 == 0 at the end of the execution?
```

2.1 Relaxed store buffering

Your first task is to implement a straight forward program that runs the store buffering litmus test and checks the output. You should do this in `main1.cpp`. You should launch two threads that implement the above testing threads, and you can use global variables as needed. Implement loads and stores as relaxed memory order atomics.

Before you begin implementation, you should reason about the different outcomes allowed by the store buffering litmus test, in terms of the final values in the `var` registers (or variables in your case). There are 3 outcomes allowed by sequential consistency, and one extra outcome allowed by the TSO relaxed memory model. Your job is to record a histogram of each outcome; there is skeleton code for you to instantiate the conditionals. You should reserve `output3` to describe the weak behavior.

The provided file has a skeleton that you should complete. It will execute the test (1024 * 256) times; your job is to check the outcomes of each test iteration and report a histogram of the outputs. You should pay special attention to whether or not you can observe the weak behavior.

The Makefile will produce an executable called `relaxed_sb` that executes this code.

2.2 Synchronized store buffering

Your second task is to use your `SRBarrier.h` from part 1.2 of this assignment to help increase the frequency of your weak behavior observations. Using your histogram produced from part 2.1, try to reason about why you might not be seeing weak behaviors frequently and how you might use the a barrier to help fix the issue. Implement this approach in `main2.cpp`. Your code will be largely the same as `main1.cpp`, except with some judicious calls to the barrier function. You should be able to observe many more relaxed behaviors.

The Makefile will produce an executable called `sync_sb` that executes this code.

2.3 Disallowing store buffering

Your third task is to change the memory order of the atomic operations to the default memory ordering (`memory_order_seq_cst`). Check to see if you are able to observe the relaxed behavior still. Implement this in `main3.cpp`. It should be the same as `main2.cpp` with only the memory order changed.

The Makefile will produce an executable called `sc_sb` that executes this code.

2.4 Experiments to run and what to report

Run the executables from each section and record your histograms. In your report, provide a graphical representation of each of your histograms and justify your observations.

Run the executables each 5 times and comment on whether your observations across iterations are similar or not. Comment on the implications this has for robust and reproducible testing for relaxed memory behaviors.

2.5 What to submit

Submit a completed version of all the files in packet, along with your pdf report.

Unlike the previous assignments, as always it will be graded on:

- Correctness: did you implement store buffering correctly? Did you use a barrier to correct the skew in the histogram? Did you correctly strengthen the memory orders?
- Conceptual: Are you able to reason about the differences between 2.1 and 2.2?
- Observations: Did you observe weak behaviors in part 2.2? Did you not observe weak behaviors in part 2.3?
- Explanation: Does your report present your results and reasoning clearly?

3 Relaxed behaviors in a mutex

In this section, you will implement a 2-threaded mutex known as Dekker's algorithm. Please get your implementation from the Overview section in the wikipedia article[1] as your only source; *please do not google for other implementations in languages closer to C++.*

3.1 A correct implementation

Implement Dekker's algorithm in `SCDekkers.h`. For all atomic operations, use the default memory order (sequential consistency). You will need to implement the `lock` and `unlock` function, as well as the constructor.

The file `main.cpp` will test this mutex (along with 2 others you will implement). The Makefile will produce an executable called `scdekker`. You will see a throughput measurement, as well as two counters: one that is incremented non-atomically in the critical section, and one that is incremented atomically outside of the critical section. For a correctly implemented mutex, they should match exactly.

You can run the thread sanitizer on this executable to help check for correctness.

3.2 A relaxed implementation

Start by coping your `SCDekkers.h` into `RDekkers.h`. Now modify `RDekkers.h` so that all atomic accesses are relaxed.

Similar to the first part of this problem, it will be compiled to an executable called `rdekkers`. It is driven by `main.cpp` and will report the same metrics. You should be able to see that mutual exclusion is violated.

Note that this program will contain data conflicts as reported by the thread sanitizer. This is expected; as long as it is exactly the same the as `SCDekkers.h` with the exception of memory orders.

3.3 A TSO-fixed implementation

Start by copying your `RDekkers.h` into `TSODekkers.h`. I have provided a `FENCE` macro for you to use which inlines an x86 `mfence`. Place this fence judiciously inside `TSODekkers.h` to fix the relaxed behaviors causing the mutual exclusion violations.

[1] https://en.wikipedia.org/wiki/Dekker%27s_algorithm#Overview

Similar to the first part of this problem, it will be compiled to an executable called `tsodekkers`. It is driven by `main.cpp` and will report the same metrics. A correct solution should show no mutual exclusion violations.

Note that this program will contain data conclicts according to the thread sanitizer, as the thread sanitizer does include the x86 `mfence` (and it is a very difficult problem to do so).

3.4 Experiments to run

Run the executables from each of the mutex implementations. For each one, record the mutual exclusion violations, and the throughput.

3.5 What to submit

Please submit all files in the directory, with the mutex files completed. Include a brief pdf report: include a graph of the throughput of each of the mutexes. Describe the number of mutual exclusion violations you observed for the relaxed mutex, and describe the fences you inserted to the TSO-fixed implementation.

As with the rest of the assignments, your grade will be based on:

- Correctness: Do your approaches implement Dekker's algorithm faithfully? Are parts 3.1 and 3.3 correct in disallowing problematic relaxed behaviors and provide mutual exclusion?
- Conceptual: Did you implement fences in the right places according to TSO? Did you relaxed the atomic accesses correctly?
- Performance: do your performance results match roughly what they should?
- Explanation: do you explain your observations clearly based on our lectures and the book readings?