Released: Nov 3rd, 2023
Due: Nov 19th, 2023, 11:59 PM EST

CS-GY 6233 - Fall 2023
Homework #4
Concurrency

Intro	[00 pts]
Mutex	[30 pts]
Spinlock	[30 pts]
Mutex, Retrieve Parallelization	[20 pts]
Mutex, Insert Parallelization	[20 pts]

General Notes

Read the assignment requirements carefully, especially what to include in your final submission for each part.
Refer to the submission instructions on the bottom of this document.

Background

This assignment will use the pthreads (Posix Threads) specification and APIs
This assignment will NOT use xv6. You will need to use a machine (or VM) that has multiple cores available, as well as the GCC compiler.

Part 1 - Intro (00 Points)

In this assignment, you will take a non-thread-safe version of a hash table and modify it so that it correctly supports insertions and retrievals from multiple threads. Start by downloading the file parallel_hashtable.c from Brightspace and compile it with the following command:

```
gcc -pthread parallel_hashtable.c -o parallel_hashtable
```

Run this program with 1 thread:

```
./parallel_hashtable 1
```

And you should see an output similar to the following:

```
[main] Inserted 100000 keys in 0.006545 seconds
[thread 0] 0 keys lost!
[main] Retrieved 100000/100000 keys in 4.028568 seconds
```

With one thread the program is correct, but try it with more than one thread

```
./parallel_hashtable 8
```

```
[main] Inserted 100000 keys in 0.002476 seconds
[thread 7] 4304 keys lost!
[thread 6] 4464 keys lost!
[thread 2] 4273 keys lost!
[thread 1] 3864 keys lost!
[thread 4] 4085 keys lost!
[thread 5] 4391 keys lost!
[thread 3] 4554 keys lost!
[thread 0] 4431 keys lost!
[main] Retrieved 65634/100000 keys in 0.792488 seconds
```

In general, the program gets "faster" in (terms of total execution time) up to a certain number of threads. However, items that are inserted into the hashtable are getting "lost."

Part 2 - Mutex (30 Points)

What circumstances cause an entry to get lost? Analyze the initial code and write a short answer to describe what it means for an entry to be "lost," and which parts of the program are causing this unintended behavior when run with multiple threads.

Next, copy the initial code to a new file `parallel_mutex.c`; we're going to start first with using **mutex(es)** to correct the problems you've identified above.

1. Modify the `insert` and `retrieve` functions so that you do not lose items when running the program with multiple threads
    a. You will likely need to use parts of the `pthread` library; things like `pthread_mutex_t`, `pthread_mutex_init`, `pthread_mutex_lock`, and/or `pthread_mutex_unlock`
2. Once your `insert` and `retrieve` functions are correct, compare the time taken to run without the mutex (original) vs. with the mutex (your changes) for various numbers of threads.
    a. In your writeup, show a line plot with the x-axis being the number of threads, the y-axis being the time taken to complete, and two lines as the series (one for the original code and one for the mutex-based code)
    b. In your writeup, estimate the time overhead that your new implementation uses (i.e., how much slow down is required to guarantee correctness using mutexes); explain your estimate

Include: working code for parallel_mutex.c; In PDF: short answer for analysis of prior code's unintended behaviors, a graph of prior running time vs. updated running time, an estimate of the timing overhead, and an explanation of how you came up with that estimate

Part 3 - Spinlock (30 Points)

If you were to replace all mutexes with spinlocks, what do you think will happen to the running time? Write a short answer describing what you expect to happen, and why the differences in mutex vs. spinlock implementations lead you to that conclusion (it's okay if your intuition turns out to be wrong, but start with this answer first).

Now, copy your `parallel_mutex.c` code to `parallel_spin.c`; we're going to test your hypothesis by replacing all mutexes with spinlocks (things like `pthread_mutex_t` become `pthread_spinlock_t`, and `pthread_mutex_lock` becomes `pthread_spin_lock`).

Once these modifications have been made, show another line plot with the x-axis being the number of threads, the y-axis being the time taken to complete, and three lines as the series (one for original/unsafe, one for mutex-based, and one for spinlock-based)

Lastly, estimate the time overhead that your spinlock implementation uses and explain your estimate

Include: working code for parallel_spin.c; In PDF: short answer for what you expect to happen when replacing mutex(es) with spinlock(s), explanation for why you have that hypothesis (based on the differences in how a mutex operates vs. a spinlock), a graph of prior running time vs. mutex running time vs. spinlock running time, an estimate of the timing overhead of spinlocks, and an explanation of how you came up with that estimate

Part 4 - Mutex, Retrieve Parallelization (20 Points)

Let's revisit your mutex-based code. When we retrieve an item from the hash table, do we need a lock? Write a short answer and explain why or why not.

Copy your `parallel_mutex.c` code to `parallel_mutex_opt.c`. In this new program, update your code so that multiple retrieve operations can run in parallel.

Once these modifications have been made, explain what you've changed.

Include: (together with part 5) working code for parallel_mutex_opt.c; In PDF: short answer for why we do or don't need a lock for retrieval, and an explanation of what you've changed to allow retrieval to be parallelizable

Part 5 - Mutex, Insert Parallelization (20 Points)

Last, let's consider insertions. Describe a situation in which multiple insertions could happen safely (hint: what's a bucket?).

Update your parallel_mutex_opt.c to handle this scenario, and explain what you've changed.

Include: (together with part 4) working code for parallel_mutex_opt.c; In PDF: short answer for when insertions could be safely parallelized, and an explanation of what you've changed in order to allow insertion to be parallelizable

Submission

1. Submit parallel_mutex.c
2. Submit parallel_spin.c
3. Submit parallel_mutex_opt.c
4. Submit a PDF with answers to each part
5. Partner.txt

Academic Honesty

Aside from the narrow exception for collaboration on homework, all work submitted in this course must be your own. Cheating and plagiarism will not be tolerated. If you have any questions about a specific case, please ask the Prof/TAs. We will be checking for this!

NYU Tandon's Policy on Academic Misconduct:
http://engineering.nyu.edu/academics/code-of- conduct/academic-misconduct