Assignment 2

Deadline: Due at 11:59PM on 30 March 2024

In this assignment, you will work in a group of maximum 3. The goal is to explore parallel programming with threads and locks using a hash table. We will be working with the provided ph.c file, which involves putting and getting keys from a hash table. It is recommended to perform this assignment on a computer with a multicore processor for optimal results.

```
./a2.out 1
0: put time = 16.541346
0: lookup time = 35.906829
0: 0 keys missing
```

Instructions

- The skeleton ph.c file is provided with the assignment instructions and you are required to download and compile the code.
- After running the program with 2 threads, observe the output which includes the time taken for put and lookup operations for each thread, as well as the completion time.

```
./a2.out 2
0: put time = 6.265808
1: put time = 6.672662
1: lookup time = 41.644365
1: 3 keys missing
0: lookup time = 41.707698
0: 3 keys missing
completion time = 48.380855
```

Understanding the Skeleton

- Note that each thread runs in two phases: putting NKEYS/nthread keys into the hash table and getting NKEYS from the hash table.
- The print statements tell you how long each phase took for each thread. The completion time at the end tells you the total runtime for the application.
- Achieving perfect parallelism means that the threads don't interfere with each other, resulting in a completion time that matches the total runtime for each thread.
- You see that that completion time is about 48 seconds. Each thread computed for about 48 seconds (~6 for put + ~42 for get). This indicates that we achieved perfect parallelism; the threads didn't interfere with each other.

- When you run this application, you may see no parallelism if you are running on a machine with 1 core or if the machine is loaded with other applications.
- Even if perfect parallelism is achieved, you may observe that the code is incorrect, as some keys inserted in phase 1 are not found in phase 2.
- Run the application with different thread counts (e.g., 4 threads).

```
./a2.out 4
1: put time = 3.250894
0: put time = 3.322325
3: put time = 3.463618
2: put time = 4.038054
3: lookup time = 38.953077
3: 6 keys missing
0: lookup time = 38.966961
0: 6 keys missing
2: lookup time = 39.570139
2: 6 keys missing
1: lookup time = 39.830595
1: 6 keys missing
completion time = 43.873740
```

- Observe the completion time and the number of missing keys. Note that the completion time may be similar, but more keys are missing, indicating good parallelism but with correctness issues.
- Independent of whether you see speedup, you will likely observe that the code is incorrect. The application inserted 1 key in phase 1 that phase 2 couldn't find.
- The completion time is about the same as for 2 threads, but this run did twice as much work as with 2 threads; we are achieving good parallelism.
- More keys are missing. In your runs, there may be more or fewer keys missing. There may be even 0 keys missing in some runs. If you run with 1 thread, there will never be any keys missing.

Identifying the problem

- Why are there missing keys with 2 or more threads, but not with 1 thread?
- Investigate and identify the sequence of events that lead to missing keys for 2 and 4 threads but not for 1 thread.

Implementing Locks

- Insert lock and unlock statements in put and get to ensure that the number of keys missing is always 0.
- Use pthread calls for locking and unlocking:

```
pthread_mutex_t lock; // declare a lock
pthread_mutex_init(&lock, NULL); // initialize the lock
pthread_mutex_lock(&lock); // acquire lock
// Critical section
pthread_mutex_unlock(&lock); // release lock
```

Testing and Performance Analysis

- Test the modified code with 1 thread then with 2 and 4 threads.
- Check if the modification has eliminated missing keys and whether the multi-threaded version is faster than the single-threaded version.

Optimizing for Parallelism

- Modify the code to ensure get operations run in parallel while maintaining correctness.
- Consider whether locks in get are necessary for correctness in this application.
- Modify the code to enable some put operations to run in parallel while maintaining correctness.
- Consider the possibility of using a lock per bucket for this purpose.

Compile and Execute

- The Makefile for assignment2 is provided.
- The Makefile is supposed to work with ph.c, file so, make sure to modify the skeleton file only.
- Run the following command in vs code Terminal to compile the code.
  `make`
- Run the following commands to run the code for 1,2 and 4 threads.
  `make run1`
  `make run2`
  `make run4`
- Run the following command to clean the out file.
  `make clean`

Submission Guidelines

- Submit the modified ph.c code with locks implementation.
- Submit the README file as in assignment 1.

- Include your student numbers at the top of your code as comments.
- Zip all files in a folder and submit it to the webcampus.

Grading

- Implementation of locks for correctness to eliminate missing keys. **(5 Pts)**
- Optimization for parallelism **(5 Pts)**