Project #1 - Concurrent SkipList with PThreads

SWE3021 Multicore Computing - Fall 2019

Due date: October 9 (Fri) 23:59

1 Goal

The primary purpose of this assignment is to gain hands-on experience with multicore parallelism using the standard POSIX Pthreads library, including exposure to multithreading synchronizations such as mutex and conditional variables. The secondary purpose of this assignment is to gain experience on implementing a master/worker task queue model of parallel computation.

2 Description

You will develop a parallel version of the serial program you can download from
http://dicl.skku.edu/class/multicore/skiplist.h
http://dicl.skku.edu/class/multicore/skiplist.cpp

The program reads a list of "qeuries" from a file. Each task consists of a character code indicating an action and a number. The character code can be either a "i" (for "insert"), "q" (for "qeury"), "w" (for "wait"), or "p" (for "print"). The input file simulates workloads entering a multi-threaded key-value system. Your main function must read this file and assign each action to background worker threads.

- If "i" is the action, the number n should be inserted into your SkipList by one of the background threads.
- If "q" is the action, one of your background threads should search for the number in your SkipList.
- If "w" is the action, one of your background threads should sleep for the specified time.
- If "p" is the action, your main thread should stop and print out the inserted numbers in an increasing order. Only after the main thread prints out the numbers, it can continue to the next line of the input file.

For example, the following input file simulates two initial jobs entering the system, followed by a 2000 -micro second delay (just one of the background threads goes sleep by calling usleep()). After the delay, we insert one more data, and print out all the numbers we inserted up to that point. Please note that some background threads may not have completed insertions. Also, some background threads might be sleeping. With 'p' command, the main thread prints out the numbers only after all the background threads finish their current tasks or wake up, i.e., you should consider using a barrier. Then, we search 30 and insert 30. Please note that, we search 30 before we insert 30 in this given input file. With concurrent execution, the search thread may or may not find 20. It depends on which thread runs first.

```
i 20
i 15
w 2000
i 10
p
q 30
i 30
```

```
$ ./a.out input.txt 4
Elapsed time: 0.00233788 sec
```

In this project you will extend this program to take advantage of multi-core CPUs using a task queue model. In such a model, the main program spawns a set number of worker threads. You should read the number of worker threads from the command line as a second parameter. (The first parameter is the input text file name.)

To achieve load balancing between multiple threads, you should probably consider making your main thread communicate with worker threads using a task queue (producer/consumer model) to keep track of tasks that still need to be processed.

The following is an example of possible execution scenarios, assuming each insertion and search takes 1 millisecond.

```
t=0              t=1                 t=2               t=3                   t=4
--------------------------------------------------------------------------
| enq,enq,enq,enq, p(barrier)                            enq,enq            |  master
--------------------------------------------------------------------------
| deq, i 20      |  deq, w 2000      |                 |(barrier) deq, q 30 |  worker 1
--------------------------------------------------------------------------
| deq, i 15      |  deq, i 10        |(barrer)         | deq, i 30          |  worker 2
--------------------------------------------------------------------------
```

This can be achieved by splitting the actual processing work into worker threads that can work parallel to the original master thread. This allows the master thread to focus on receiving jobs while the worker thread focuses on doing the actual work. Note that the the purpose of "w" is to emulate an unexpected delay in worker threads.

Your program should work as follows. At the beginning of execution, the master thread spawns a set number of worker threads (given by a command line parameter). The worker threads are idle at first. Once the workers have been fully initialized, the master then begins to handle tasks from the input file by adding them to a task queue, waking up an idle worker thread (if there are any) for each task. When a thread is awakened, they begin to pull tasks from the queue and process them. If the queue ever runs out of tasks, the worker should block again until awakened by the master. If the worker encounters a "w" (wait) command, it waits the given number of seconds before continuing in the input file. After all tasks have been added to the queue, the master waits for the queue to be exhausted, waking idle threads as necessary to help. When the queue is empty, the master waits for non-idle workers to finish, then sets a global flag to indicate that the entire program is done, re-awakening all worker threads so that they can terminate. The master then cleans everything up and exits.

To implement the above system, you would need Pthread threads, mutexes, r/w locks, or conditional variables we covered in class. Your program should take the number of worker threads as the second command line parameter; the performance on parallelize-able workloads should scale linearly with the number of threads.

Note that skiplist.h is not a fully optimized implementation. If you want, you can improve it. Probably, you should improve it to achieve high performance.

3 Grading:

1. If you do not submit, you will get 0 points.
2. No late submissions will be accepted.
3. If you cheat, you will get -10 points.
4. If your code does not compile or your output is incorrect, you will get 2 points.
5. If your submitted code shows you didn't try hard to improve the performance, you will get 0∼2 points.
6. The remaining 8 points will be the performance score, which is prorated by the following equation.

$$\frac{(the\ slowest\ code's\ execution\ time) - (your\ code's\ execution\ time)}{(the\ slowest\ code's\ execution\ time) - (the\ fastest\ code's\ execution\ time)}$$

4 Class Accounts in In-Ui-Ye-Ji Cluster

In this class, you will have to work on projects on "In-Ui-Ye-Ji" cluster machines. Your ID is your student ID. If you have problems with ID and password, please send an e-mail to TA: ahnshzzang@gmail.com.

SSH command is as follows:

```
$ ssh swin.skku.edu -p 1398 -l your_student_id
```

Note that you have to use ssh port 1398, which is the year when SKKU was first established in Joseon dynasty. The cluster has eight nodes, swin, swui, swye, swji, titan1, titan2, titan3, and titan4. You should be able to login to any node of this cluster using the same id and password.

5 How to Submit

To submit your project, you must run `multicore_submit` command in your project directory in swin.skku.edu server as follows.

```
$ cd your_working_directory
$ multicore_submit project1 ./
```

This command will submit all your files in your current directory to the instructor's account.

For any questions, please post them in Piazza so that we can share your questions and answers with other students. Please feel free to raise any issues and post any questions. Also, if you can answer other students' questions, please don't hesitate to post your answer.