Lab 3, lock-free concurrent skiplist

Requisites

- Read lazy and lock-free skiplists in HSLS ch 14
- The basic algorithms are given in HSLS, ch 14.4
- Check the textbook companion website here

Tasks

1. Implement a class LockfreeConcurrentSkipListSet following the approach in HSLS chapter 14.4

2. Devise a little program that will populate your skiplist using two populations of 10ˆ7 integers in an interval (value range) of size 10ˆ7 (in the first instance, later we will vary this range). For the first population the integers should be chosen uniformly at random, and the second should use a normal distribution with a suitably chosen mean and variance, to ensure that the mean is distinct but all points have a "reasonable" minimum probability of getting sampled. Implement a little test to verify that the two populations have roughly the expected means and variances. (Of course, since your skiplist will eliminate duplicates, this check will not be exact.) This will require you to implement an additional call that allows you to list the skiplist members in order.

3. Devise a test class that, for a given number of threads up to 48, exposes the populated skiplists to 10ˆ6 operations with an adjustable mix of random add, remove, and contains operations, for instance 10% add, 10% remove, 80% contains. Make sure the test class does not impose any sequentialisation constraints. Plot the average execution times over 10 runs for 2, 12, 30, and 46 threads on both populations. Repeat for the following distributions of operations: a) 50% adds and 50% removes, b) 50% contains, 25% adds, 25% removes, c) 90% contains, 5% adds, 5% removes. Explain your observations.

4. We believe the skiplist implementation is linearisable. We now wish to test this hypothesis. We want to use System.nanoTime() to record the time at which the linearisation point was encountered. To do this, identify the linearisation points and sample the times they are crossed by inserting calls to System.nanoTime() at appropriate places in the code. This needs to be done with care.

5. This will introduce measurement errors, since the traversal of the linearisation point and the sampling of the clock is not an atomic operation. It is possible for a thread to intervene between the point of sampling the time and the time when the linearisation point is traversed. This may cause a trace constructed by ordering the sampled events according to the time samples to violate the sequential specification, for example contains(x)

may return true if if the last prior event on x was to remove x. The purpose of the remaining parts of the lab is to devise ways of instrumenting the skiplist as non-intrusively as possible, in order to validate that our skiplist implementation is indeed linearisable, and to estimate the error resulting from any imprecision in our sampling of the linearisation point times. Below I suggest you to explore three different ways to accomplish such sampling. If you get inspired to devise a method of your own, feel free to substitute one of the methods suggested below with your own approach.

6. First we use a global lock to protect the interval between the linearisation point and the time sampling event (in whatever order you implement this). This will prevent measurement errors since these intervals are now executed atomically, but it is also highly intrusive, due to the use of a global lock, which will introduce sequential execution constraints into the system that are not present in the original system. To get an idea of this, at least in terms of the effect on execution time repeat a few of the measurements in 3. above and compare. Do you observe a noticeable effect as the number of threads increase? If so, how do you explain this?

7. In order to answer if you've identified the correct linearisation points use the critical sections introduced by the global lock to write a global log of the call and return values along with their linearisation point timestamps. Observe that you don't need to do all the logging inside the critical section. You can keep the log in memory and only write to file when you are done. If the linearisation points are correctly identified and the log is computed correctly you should be able to verify that the computed log satisfies its sequential specification, for instance that contains(x) only succeeds when the most recent add/remove of x was an add, etc. Write a small routine implementing the check that the sequential specification is met and verify that the logs obtained from the experiments in 6. indeed does pass your check.

8. For a less intrusive approach we try instead to log the time samples without using a global lock. This will in general produce sampling errors since the interval between crossing the linearisation point and sampling the time is no longer protected. The purpose of the remaining part of the lab is to get a feel for the magnitude of this error.

9. We use two methods to construct such a log. In method 1 the logs are built only locally, i.e. per thread, at runtime, by writing the samples into a suitable structure like a large enough local array or linked list. At the end of execution the local logs can then be merged into a single global log ordered by sampled time instances. Implement this. We expect that System.nanotime() samples the time at a fine enough granularity that duplicate time samples are eliminated.

10. In method 2 a special thread is assigned the task of aggregating the samples at runtime. This will obviously reduce the number of "payload threads"

that perform operations on the actual skiplist by one. This aggregation can be performed by implementing a multiple producer-single consumer queue that the skiplist threads can use to pass samples on to the aggregator thread. Implement both these methods carefully, remembering to introduce as little extra synchronisation as possible. Use your knowledge of the Java Memory Model and the happens-before relation to determine what new execution constraints, if any, are introduced on your skiplist by your two (hopefully) non-blocking logging methods.

11. If both methods are correct and maximally non-intrusive they should produce similar results, so as the final part of the lab we try to check if this is the case. Note that the probability of a race occurring that can result in a log violating the sequential specification must increase as the integer value range decreases. For both logging methods repeat the experiment of 3. 10 times for 50% remove, 50% add and 46 cores first for the 10ˆ7 then 10ˆ6 etc value range until either a violation of the sequential specification is found or the value range cannot be reduced further. What do you find, and do your two non-blocking logging methods produce consistent results, if not, why?