CSMC 23010-1	Due March 11, 2014

Homework 5

Professor: Hank Hoffmann — *TA: Harper Zhang*

Theory

From the book: 108, 109, 120 (you will need to make some assumptions to answer this, please clearly state your assumptions), 121, 124, 125, 127, 129, 132, 159, 160, 185, 186, 188, 192.

As before, I may cut from the end of these assignments if I don't get to the material in class, so start with stuff we have covered.

Programming

This assignment is all about high-performance concurrent hash tables. You will build a set of different hash tables and extend one of your choosing in a couple non-trivial ways. Each design will support ADD(), REMOVE() and CONTAINS() methods (as described on page 299 in the text) and will be resizable, based on a *fullness* criteria. That is, you will specify some measure of the occupancy of the table that will trigger a RESIZE() operation, which will then double the space and copy the data. There are many such criteria (*eg.* max total number of items in the table, max total items in any given bucket etc.) and for this assignment you will decide which heuristic to use and provide the reasoning and experimental data (of your own design) to support your decision.

The world is awash in variations on hash tables - in this assignment we will focus on two canonical types: **Closed-Address** hash tables have linked lists located at each bucket (or hash index), which support the ADD(), REMOVE() and CONTAINS() methods. **Open-Address** hash tables have individual entries at each hash index, but resolve collisions by inserting items at nearby locations (*eg.* cuckoo hashing in Section 13.4 of the text and linear probing - *see the Wikipedia article on linear probing*). Please see Chapter 13 in the text for more detail.

All of the hash table designs use locks and while you're permitted to use your lock designs from ***pset4***, you may also build on the pthread MUTEX provided in the pthreads library. In general, we will use the STRIPEDHASHSET paradigm (Figure 13.6 in the text), allocating a lock bank of size $2^{\lceil \log_2 n \rceil}$ (the next larger power of two) for n threads. This allows us to map a hash index to a particular lock with a bitwise AND operation, which is cheap and convenient. Further, in practice it is generally accepted that if the number of locks is approximately equal to the number of threads (with a uniform distribution of accesses to each) you will largely mitigate lock contention. Feel free to demonstrate to yourself that this is true in our case, but it is not necessary for this assignment; you are free to assume it. In addition, we'll make the simplifying assumption that all hash tables have 2^k buckets for some k - after all, we're not savages.

Hash Tables We will build each of the following types of hash table, H:

- **Lock-based Closed-Address** (LOCKING): This is a standard hash table where ADD() and REMOVE() methods use a WRITELOCK() to make modifications to the list located at a bucket and CONTAINS() uses the corresponding READLOCK (see Section 8.3 in the text for a review of the READWRITELOCK concept). A RESIZE() method merely grabs all WRITELOCKS in sequential order (to avoid deadlock with a rival RESIZE() attempt) to halt activity during the course of the RESIZE() operation.
- **Lock-Free-Contains Closed-Address** (LOCK-FREEC): This design uses a regular REENTRANTLOCK and does not require a READWRITELOCK, since the CONTAINS() method does not ever make use of the READLOCK() functionality. This design has the additional requirement that if

a CONTAINS() and an ADD() or a REMOVE() method proceed concurrently that the result of the CONTAINS() is linearizable with the other call; i.e., you will have to argue that your design meets this requirement.

- **Linearly Probed Open-Address** (LINEAR-PROBE): This is a standard linear probing style hash table, where each entry in the table has a counter which defines the maximum number of steps necessary to find a previously added item - that is, if an ADD() method had to walk k steps to find a vacant spot, the counter would be set to the max of the previous value and k. This allows both the REMOVE() and CONTAINS() methods to limit their searches. Beware of the potential for deadlock when grabbing locks for multiple hash indices (as in ADD() and REMOVE()) in the event that the home index is not vacant.

- **Your Design!** (AWESOME): This is your opportunity to design a really fancy version of either LOCK-FREE or LINEAR PROBE (or try something completely different). In particular, you should extend the design in at least two non-trivial ways, demonstrating through experimentation of your own design why it is a good choice and providing your reasoning for its correctness. *Please include a section in your report where you concretely discuss the performance tradeoffs you made, your experiments to test those tradeoffs and your reasoning about the correctness of your algorithm.* You are free to pursue whatever design enhancements you like provided they are roughly as sophisticated as the following examples:

  - **Concurrent Resize:** Design your hash table such that all operations, including ADD() and REMOVE(), proceed concurrently with RESIZE() method calls.
  - **Distributed Resize:** When a single thread performs the resize operation, there is the potential that if the rate of all threads making ADD() calls exceeds the rate of a single thread copying items, that the maximum speedup is governed by the single thread performing the RESIZE() operation. Instead, your design could offload the work of copying items to the other threads. Use your engineering judgment to decide among the myriad ways there are to do this - be sure to include your justification for both performance and correctness in the report.
  - **Completely Lock-Free:** Can you build a concurrent hash table using only synchronization primitives like GETANDSET() and FETCHANDINCREMENT with no locks?
  - **Perfect Hashing:** One disadvantage of using a list for each bucket in a Closed-Address hash table is that the traversal of that list is inherently serial and takes time linear in the length of the list in the worst case. In the theoretical analysis of hash tables, there is a technique called **Perfect Hashing** which reduces the access time of a hash table to $O(1)$ Essentially, the technique amounts to another hash table (albeit much smaller) to hold items at each bucket, rather than a list: a hash table of hash tables.

Performance Tests We will test the concurrency of your hash table designs with a simple throughput test, taking packets from a HASHPACKETGENERATOR object which are encoded as ADD, REMOVE or CONTAINS requests. Then each packet is applied to the hash table according to this encoding, calculating the FINGERPRINT() - a proxy for generic packet processing work - as before. We will build three versions of this code:

- SERIAL: A single thread serially retrieves packets and makes the appropriate calls into a serial hash table implementation. This version of the code is configurable by the following parameters:
  - NUMMILLISECONDS (M) the time in milliseconds that the experiment should run.
  - FRACTIONADD (P_+) the percentage of packets that result in an ADD() call
  - FRACTIONREMOVE (P_-) the fraction of packets that result in a REMOVE() call. Note that the fraction of CONTAINS() packets is $1 - P_+ - P_-$.

  - HITRATE (ρ) the fraction of CONTAINS() calls that find data in the hash table.
  - MEAN (W) the expected amount of work per packet.
  - INITSIZE (s) the number of items to be preloaded into the table before the Dispatcher and Workers begin working (call GETADDPACKET() using the HASHPACKETGENERATOR s times).

- PARALLEL: As in the previous two assignments, a Dispatcher thread will solely retrieve packets from the HASHPACKETGENERATOR and write them into a bank of Lamport queues (QUEUEDEPTH = 16). Using your best load balancing strategy from the last assignment, the Worker threads will retrieve the packets from the queue bank and then make concurrent calls into the hash table. This version of the code is configurable by the same parameters as SERIAL plus:
  - NUMWORKERS (n) the number of Worker threads.
  - TABLETYPE (H) the type of hash table (*eg.* LOCKING, LOCK-FREEC etc.)

- PARALLELNOLOAD: This application is like PARALLEL, except that the Workers just drop the packets. The purpose of this application is to determine the maximum throughput of the Dispatcher. It is parameterized by the same parameters as PARALLEL minus H.

Experiment

You should create a directory and expand the accompanying utilities file into it. It will provide some basic components (*e.g.* utilities for timing code, a random packet generator etc.). You are free to modify or rewrite any of these except for HASHPACKETGENERATOR.

Next, you will perform the following set of experiments across various cross-products of these parameters. Each data point is a measurement taken on a dynamic system (a computer...) and is thus subject to noise. As a result, some care should be taken to extract representative data - we would propose running some reasonable number of experiments for each data point and selecting the median value as the representative. The median of 5 runs should suffice to get a reasonable estimate of the performance on any given configuration - please use your own engineering judgment to decide how many trials you require. Setting the experiment time M to 2000 (*ie.* 2 seconds) should suffice to warm up the caches for all of the following experiments. In each of the following experiments, we describe an experiment that you should perform and analyze. Please provide your hypotheses for the trends you see, supported by additional experiments of your own design if your hypotheses are not sufficiently substantiated by these experiments.

Hash Table Throughput Tests

1. **Dispatcher Rate** Run PARALLELNOLOAD with $n = C$ (where C is the number of cores in your test machine) and $W = 1$ to estimate the rate at which the Dispatcher retrieves packets and writes them into the queue bank. How does this rate compare with the same rate in the first programming assignment?

2. **Parallel Overhead** Run SERIAL and PARALLEL with $n = 1$ and $W = 4000$ on each hash table type, H ($\in$ {LOCKING, LOCK-FREEC, LINEAR-PROBE, AWESOME, }), with two load configurations:
   (a) **Mostly Reads** $P_+ = 0.09$, $P_- = 0.01$, $\rho = 0.9$
   (b) **Heavy Writes** $P_+ = 0.45$, $P_- = 0.05$, $\rho = 0.9$

3. **Speedup** Run SERIAL and PARALLEL with $n \in \{1, 2, 4, 8, 16, 32\}$, $W = 4000$, all H and under two load configurations:
   (a) **Mostly Reads** $P_+ = 0.09$, $P_- = 0.01$, $\rho \in \{0.5, 0.75, 0.9, 0.99\}$

(b) **Heavy Writes** $P_+ = 0.45,\ P_- = 0.05,\ \rho \in \{0.5, 0.75, 0.9, 0.99\}$

Make 8 plots (one for each load configuration) with speedup (relative to SERIAL) on the Y-axis and n on the X-axis. Analyze this data and present your most insightful observations in the report.

Writeup

Please submit a typeset (learn LaTeX if you haven't already!) report summarizing your results from the experiments and the conclusions you draw from them. Your report should include the experiments as specified above, any additional experimental data you used to draw conclusions and a self-contained report. That is, somebody should be able to read the report alone and understand what code you developed, what experiments you ran and how the data supports the conclusions you draw. In addition, your report should provide the reasoning (though, mercifully, not a formal proof) as to why your designs are correct. In particular, for each method in each design you should specify whether the method is wait-free, lock-free, deadlock-free or starvation-free and the reasoning for it. Finally, submit your development directory, containing all of the working code.