1 Introduction

Big fat locks and fine-grained locking will only take you so far. In this homework, we experiment with two alternative approaches: read-copy-update, which allows lock-free reader access, and completely lock-free data structures, which use atomic compare-and-swap (CAS) operations to safely modify the data structure without the use of explicit locks.

2 Preliminaries

This homework requires access to a multi-core machine. Four servers are available for your use for this assignment: Kentsfield-1 and 2, and words. To access the kentsfields, which are hiding behind a firewall, use these commands:

Kentsfield-1	`ssh -p 8021 netid@bits-head.cs.uic.edu`	4 Intel Kentsfield threads
Kentsfield-2	`ssh -p 8034 netid@bits-head.cs.uic.edu`	4 Intel Kentsfield threads
words	`ssh -p 8190 netid@bits-head.cs.uic.edu`	16 Intel Westmere threads
lines	`ssh netid@lines.cs.uic.edu`	64 AMD Abu Dhabi threads

your UIC netid password applies. Watch out: if someone else is using the same machine as you at the same time, your performance results will be heavily affected. Before running a performance experiment, take a quick lock with `top`.

All four machines mount the same home directory, so your files will be visible everywhere. For generating performance plots, please run on lines. When you are just testing your code, better to use the kentsfields with up to 4 threads, or words with up to 16 threads, to leave room on the big iron.

Note that `lines` is an AMD machine - this implies a somewhat different architecture: different cache and memory latencies, different clock frequencies, and different instruction latencies and reciprocal throughput. This probably will not significantly impact your results in this homework, but it could be interesting to compare results from `words` and `lines`.

3 Assignment

The template for this assignment is available at `https://github.com/uicperformance/lockfree`. To build the programs, simply run `make` in the checked-out repository folder.

You are given the single-threaded list from hw4, and there are `test_` and `benchmark_` targets in the Makefile, as in hw4. The `test_` programs crash, hang in an infinite loop, or exit with an error report when run with `-n 2` or higher. This is due to a race condition: the programs work fine with one thread. If you completed hw4, you also have a hand-over-hand locking solution that's interesting to compare against.

3.1 Establish performance baseline, and prepare LaTeXreport

Using gnuplot, generate plots of the total throughput of benchmark_list_single_thread for a single thread, using the single-threaded list. Add a big fat spinlock to the list, and report the performance with 1, 2, 4, 8, 16, 32, 64 threads on the same plot. Label the lines accordingly. On the x-axis, vary the size of the list between 1–1,000,000 entries. On the y-axis, show the number of find operations performed per second. Make sure axes and lines are correctly labeled, with names and units as appropriate. Refer to hw4 for how to use gnuplot and latex.

The big fat lock performance is a likely lower bound on the performance of these data structures in a multi-threaded setting: compare the performance of a single threaded list vs. the bfl list with multiple threads, and provide a brief analysis.

3.2 A Read-Copy-Update (RCU) List

The homework template includes `qsbr.c,h`, an RCU implementation. Using qsbr, implement a new list, this time with a big fat lock for updaters, but lock-free reader access. Put your code in a file named `list_qsbr.c`: Matching Makefile test and benchmark targets are already provided.

There is no need to read `qsbr.c`, though it could be instructive. Read `qsbr.h` to learn about the QSBR interface. You will need one qsbr_pthread_data_t per pthread, which you need to reference whenever invoking qsbr functions. The ds_data field in the pthread_data_t struct is there for this purpose.

You will also need to initialize qsbr once globally, and once per thread using the appropriate methods. When freeing a chunk of memory, use qsbr_free_ptr instead of free(), and indicate to qsbr that the reader is in a quiescent period whenever it is done accessing the list.

Evaluate the performance of your RCU list with a big fat updater lock, using three plots: One with 50% updates, varying list size, and 1,2,4,8,16,32,64 threads. Another plot with fixed list size at 10,000 elements, varying update rate, and 1,2,4,8,16,32,64 threads. Finally, one with 50% updates, 10,000 elements, and varying thread count. Analyze your results and state your observations.

3.3 Fine-Grained locking with RCU

Instead of a big fat lock, create an solution that uses fine-grained locking. Traverse the list without locking, then before any update, acquire locks on any affected nodes, and use qsbr_free_ptr instead of free. `Note:` Since traversal is non-locking, you are not guaranteed that the state of the node remains the same after acquiring the lock, as it was before you acquired the lock. Double-check before proceeding. You may want to mark deleted nodes as such, to make checking easier.

You may use a separate C file, or preprocessor macros to support both BFL and fine-grained operation in the same file. Add appropriate Makefile targets.

Evaluate the performance of your fine-grained locking RCU list using three plots: One with 50% updates, varying list size, and 1,2,4,8,16,32,64 threads. Another plot with fixed list size at 10,000 elements, varying update rate, and 1,2,4,8,16,32,64 threads. Finally, one with 50% updates, 10,000 elements, and varying thread count. Analyze your results and state your observations.

3.4 An entirely lock-free list

Finally, eliminate the use of locks from your list entirely. Put this implementation a file called list_cas.c: Makefile targets are provided. Instead of locking, use atomic compare-and-swap (CAS) operations. CAS is defined sa a macro in util.h. Traverse the list in lock-free fashion, then use CAS to perform your updates. Restart from the beginning if the CAS fails. For insert, this is straightforward. Delete is more complicated: though many designs are possible, you may refer to the Tim Harris paper on a lock free linked list for some ideas. In principle, first mark the node for deletion by tagging its next-pointer, then remove the node from the list, both using CAS. If either operation fails, restart. Any tagged node needs to be ignored by other operations.

Evaluate the performance of your lock-free RCU list using three plots: One with 50% updates, varying list size, and 1,2,4,8,16,32,64 threads. Another plot with fixed list size at 10,000 elements, varying update rate, and 1,2,4,8,16,32,64 threads. Finally, one with 50% updates, 10,000 elements, and varying thread count. Analyze your results and state your observations.

3.5 turn-in instructions

Similar to the previous homework, turn-in is by git classroom, using this invitation link `https://classroom.github.com/a/2THtj6Zm`. Include `report.pdf` in the root folder of your submission.

3.6 Double-check your submission

To make sure that you submitted everything you think you submitted (`git` can be a little tricky until you get used to it), *git clone* your turn-in repository into a fresh folder, check that your `report.pdf` is in the submission folder, and that all the Makefile targets build correctly.