Programming Assignment #2: Concurrent Database

Due Monday, May 15

In this assignment, you will implement a BTree database, supporting concurrent search, insert, and delete. The goal is to learn about and experiment with concurrent access to data structures. For this assignment, you will be using pthreads. This is a relatively new assignment, so modifications may be made.

Make sure you read the syllabus regarding late days, honor code, etc. Note: you may work alone or in groups of two.

Deliverables

1. Code (25% of grade): To submit your code, run

```
/afs/ir/class/cs315a/bin/submit pa2
```

from the directory containing your submission. All files in the current directory and any subdirectories will be copied to our repository. Please **do not submit our database file** back to us, but **do submit your output databases**. Please include a `README` with the names and SUNet IDs of all group members and a description of the directory structure of your submission (what answers are where, etc.). You must **submit by 11:59 p.m.** on the due date.

2. Writeup (75%): Answer the questions in this document, making appropriate graphs. A significant part of parallel programming is the analysis of program performance, so after you write correct code, you usually need to spend just as long tuning the code for proper performance. We will rely upon your written report to grade your performance-tuning efforts and your analysis of the causes of any problems that may occur. For later programming assignments, much of this analysis should come from your insights about the hardware architecture as well as from the program code itself.

When reporting timing information, **include the input filename and machine name**.

Submit your writeup on paper in the homework box by Darlene's office, Gates 408 **(before 5 p.m.)** or electronically along with your code (PDF, PS, or HTML; no Word documents please) **(before 11:59 p.m.)**. Please include names and SUNet IDs of all group members.

The Environment

You must compile and run your applications on Sun multiprocessor machines using either pthreads or OpenMP. At Sweet Hall, there are many such machines available: the `saga` (1–22) and `elaine` (1–43) machines all have two processors, the two `tree` machines have eight, and `junior` has 16. In order to keep the load on the larger machines reasonable as deadlines approach, we recommend that you compile and debug your applications on the 2-processor machines before moving to the larger machines only for final performance runs. `junior` has been reserved for exclusive use by another class from 6 p.m.–6 a.m., so you can only use it during the day.

Timing Runs

After debugging your programs on the smaller machines (`saga` and `elaine`), you'll need to acquire timed runs to compute speedups. Make sure you use the same machine for *all* runs of one problem. You may vary the machine between problems.

Use `junior` or `bigpun` for timing runs. `bigpun` has a batch system so only one program will be running at once; this makes the reported times more accurate. Please follow the batch submission instructions expressed on the newsgroup: namely, please specify both memory requirements and processor count or your jobs will be killed.

You may need to modify the `Makefile` provided. In order to use the Sun compiler, you must set the appropriate environment variables by executing `source pa.csh` in csh or `. pa.bash` in bash, located in `/afs/ir/class/cs315a/PA`. If you are interested in reading more about the Sun compiler, we refer you to the man page or the manual, which is available on the internet:

Sweet Hall: `http://docs.sun.com/app/docs/coll/771.6`

If you choose to use other machines, with other compilers, the exact method of using these features may vary considerably. If you choose to use a different kind of machine, we will not support you with any unusual compiler setup that may be necessary.

The Problem

A simple database can be implemented using what's called a BTree—a balanced, *n*-ary tree stored in memory and/or disk containing keys and data. For more information on BTrees, see the "White Book" (Cormen, Leiserson, Rivest, and Stein), Chapter 18, or numerous other available resources. Three key operations on BTrees are search, insert, and delete. Insert and delete may invoke re-balancing operations on the tree (moving nodes to maintain a tree of height O(log(n))).

We provide a pre-built database full of strings, `defrhymes.db`, and three inputs, each containing search, insert, and delete transactions:

- **`random.trans`:** A random collection of searches, inserts, and deletes.
- **`amazon.trans`:** Mostly searches, and few inserts and deletes.
- **`science.trans`:** A lot of inserts, some searches, and few deletes (like in a laboratory collecting a lot of data).

We provide a sequential version, `PA2/pa2.cpp`, which reads an input database and a list of transactions. **The program modifies in the database in place**, so you may have to recopy your input each time you run your program. Results from searches will be printed to `stdout`. To check correctness, diff `stdout` and a traversal of your database with those generated by the sequential version.

You may want to create your own database and input files for testing. `create.cpp` was the code used to create `defrhymes.db` from a file with words on each line. You should be able to discern the fileformat for the input without trouble.

Execute the program by typing

```
./pa2 <# procs> <database.db> <input.trans>
```

You can copy the necessary files from the following directory on the Leland system:

```
/afs/ir/class/cs315a/PA/PA2/
```

Problem 1. Simplistic Parallelization

To begin, augment the BTree with per-node locks to support concurrent operations and implement an input queue: take the commands from the transaction input file and distribute them to your threads, perhaps using your work queue from PA1.

1A) Obviously, holding an exclusive lock on all nodes touched during a traversal will always be correct, but completely serializes (since you'll lock the root node). Instead, implement operations by using "spider locks" (a.k.a., "hand-over-hand locks"), which lock only those nodes required for correctness. For example, when traversing down the tree, you'll need to ensure that a delete does not remove a node out from under you. So, lock the current node, lock the child you'll be going to, then unlock the current node and move to the child. Similar scheme will be required for inserts and deletes.

Discuss your implementation and provide performance results for the `amazon.trans` input with 2, 4, and 8 CPUs. What is the impact of lock overhead/contention?

1B) Since real databases tend to have many more searches than inserts and deletes, explore using read/write locking and discuss and measure your implementation with `amazon.trans` on 2, 4 and 8 CPUs. How does it compare to your results in 1A? Do inserts and deletes need to be handled differently than before? Did you make implementation decisions based on the ratio of searches vs. inserts/deletes?

Problem 2. Do Some Research

There are *many* of ways to get better concurrency than what was suggested in Problem 1. The key will be avoiding global locking and protecting only what absolutely needs to be. For example, what happens on an insert if a node is full/not full? What happens if the tree needs to be rebalanced? We will discuss several techniques in the review session, but we encourage you to read the following papers and/or the many textbooks that cover this material (like DB textbooks). Options range from lock-free techniques, to locking order heuristics, to reference counting, etc.

```
/afs/ir/class/cs315a/PA/PA2/p354-kung.pdf
/afs/ir/class/cs315a/PA/PA2/p650-lehman.pdf
```

2A) Discuss **2** different strategies and implement **one** of them. Report and analyze speedups on 8 and 16 CPUs. When you discuss a strategy, make sure to include what changes must be made to the input queues, output queues, and search/insert/delete algorithms. What assumptions have you made under your strategy? Defend them. Report results for all three inputs.