Hash Hash Hash 1.0.3

Jon Eyolfson
May 10, 2021
Due: May 21, 2021 at 8 PM PST

In this lab you'll be making a hash table implementation safe to use concurrently. You'll be given a serial hash table implementation, and two additional hash table implementations to modify. You're expected to implement two locking strategies and compare them with the base implementation. The hash table implementation uses separate chaining to resolve collisions. Each cell of the hash table is a singly linked list of key/value pairs. You are not to change the algorithm, only add mutex locks. Note that this is basically the implementation of Java concurrent hash tables, except they have an optimization that doesn't create a linked list if there's only one entry at a hash location.

Additional APIs. Similar to Lab 2, the base implementation uses a linked list, but instead of `TAILQ`, it uses `SLIST`. You should note that the `SLIST_` functions modify the `pointers` field of `struct list_entry`. For your implementation you should only use `pthread_mutex_t`, and the associated init/lock/unlock/destroy functions. You will have to add the proper `#include` yourself.

Starting the lab. Run the following command to get the skeleton for Lab 3: `git pull upstream main`. You should be able to run `make` in the `lab-03` directory to create a `hash-table-tester` executable, and then `make clean` to remove all binary files. The executable takes two command link arguments: `-t` changes the number of threads to use (default 4), and `-s` changes the number of hash table entries to add per thread (default 25,000). For example you can run: `./hash-table-tester -t 8 -s 50000`.

Files to modify. You should only be modifying `hash-table-v1.c`, `hash-table-v2.c`, and `README.md` in the `lab-03` directory.

Tester Code. The tester code generates consistent entries in serial such that every run with the same `-t` and `-s` flags will receive the same data. All hash tables have room for 4096 entries, so for any sufficiently large number of additions, there will be collisions. The tester code runs the base hash table in serial for timing comparsions, and the other two versions with the specified number of threads. For each version it reports the number of μs per implementation. It then runs a sanity check, in serial, that each hash table contains all the elements it put in. By default your hash tables should run `-t` times faster (assuming you have that number of cores). However, you should have missing entries (we made it fail faster!). Correct implementations should *at least* have no entries missing in the hash table. However, just because you have no entries missing, you still may have issues with your implementation (concurrent programming is significantly harder).

Your task. Using only `pthread_mutex_*`, you should create two thread safe versions of the hash table "add entry" functions. You only have to add locking calls to `hash_table_v1_add_entry` and `hash_table_v2_add_entry`, all other functions are called serially, mainly for sanity checks. By default there is a data race finding and adding entries to the list. You'll need to fill in your `README.md` completely. Most sections are what you expect from previous labs, however there is more to add for this lab (explained below).

For the first version, v1, you should only be concerned with correctness. Create a **single** mutex, only for v1, and make `hash_table_v1_add_entry` thread safe by adding the proper locking calls. Remember, you should only modify code in `hash-table-v1.c`. You'll have to explain why your implementation is correct in your `README.md`. You should test it versus the base hash table implementation and also add your findings to `README.md`.

For the second version, v2, you should be concerned with correctness and performance. You can now create as many mutexes as you like in `hash-table-v2.c`. Make `hash_table_v2_add_entry` thread safe by adding the proper locking calls. Similar to the first version, you'll need to explain why your implementation is correct, test it's performance against the previous implementations, and add your findings to `README.md`.

In both cases you may add fields to any hash table `struct`: the hash table, `hash_table_entry`, or `list_entry`. You code changes should not modify `contains` or `get_value`. Any other code modifications are okay. However, you should not change any functionality of the hash tables.

Errors. You will need to check for errors for any `pthread_mutex_*` functions you use. You may simply `exit` with the proper error code. You are expected to destroy any locks you create. The given code passes `valgrind` with no memory leaks, you should not create any.

Tips. Since this is a lab about concurrency and parallelism, you may want to significantly increase the number of cores given to your virtual machine, or run your code on a Linux machine with more cores.

Example output. You should be able run:

```
> ./hash-table-tester -t 8 -s 50000
Generation: 130,340 usec
Hash table base: 1,581,974 usec
  - 0 missing
Hash table v1: 359,149 usec
  - 28 missing
Hash table v2: 396,051 usec
  - 24 missing
```

Submission. Simply push your code using `git push origin main` (or simply `git push`). For late days will we look at the timestamp on our server. We will never use your commit times as proof of submission, only when you push your code to the course Git server.