Lab 4: Parallel Hash Tables `1.0.1`

ECE 353: Systems Software
Jonathan Eyolfson
March 6, 2023 **Due: March 13, 2023 @ 11:59 PM ET**

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.

Lab setup. Ensure you're in the repository (`cd ~/ece353-labs`) directory. Make sure you have the latest skeleton code from us by running: `git pull upstream main`.

This will create a merge, which you should be able to do cleanly. If you don't know how to do this read Pro Git. **Be sure to read chapter 3.1 and 3.2 fully.** This is how software developers coordinate and work together in large projects. For this course, you should always merge to maintain proper history between yourself and the provided repository. **You should never rebase in this course, and in general you should never rebase unless you have a good reason to.** It will be important to keep your code up-to-date during this lab as the test cases may change with your help.

You can finally run: `cd pht` to begin the lab.

Additional APIs. Similar to the suggestion in Lab 3, 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.

Building. First, make sure you're in the `pht` directory if you're not already. After, run the following commands:

```
meson setup build
meson compile -C build
```

Whenever you make changes, you can run the compile command again. You should only need to run setup once.

Starting the lab. After you build you'll have a `build/pht-tester` executable. 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: `build/pht-tester -t 8 -s 50000`.

Files to modify. You should only be modifying `src/hash-table-v1.c`, `src/hash-table-v2.c`, and `README.md` in the `pht` 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 comparisons, 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. Only `hash_table_v1_add_entry` and `hash_table_v2_add_entry` need locking calls. 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.

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 its 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 to run:

```
> build/pht-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`) to submit it. *You need to create your own commits to push, you can use as many as you'd like.* You'll need to use the `git add` and `git commit` commands. Push as many commits as you want, your latest commit that modifies the lab files counts as your submission. For submission time we will *only* look at the timestamp on our server. We will never use your commit times (or file access times) as proof of submission, only when you push your code to the course Git server.