CS 111: Operating System Principles
Lab 4

Hash Hash Hash 3.0.0

Jui-Nan Yen, Medha Kini
Derivative document by: Jonathan Eyolfson
July 8, 2025
Due: August 8, 2025 @ 11:59 PM PT

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 s `truct l ist_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. Download the lab3 skeleton code from the course BruinLearn page under the assignments tab.

You should be able to run `make` in the `lab3` directory to create a `hash-table-tester` executable, and then `make clean` to remove all binary files. T he e xecutable t akes t wo c ommand l ink a rguments: – t c hanges 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 `lab3` 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 d ata. A ll h ash t ables h ave r oom f or 4 096 e ntries, s o f or a ny s ufficiently l arge 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 n umber o f t hreads. F or e ach v ersion i t r reports t he n umber o f 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 i n y our R `EADME.md` c ompletely. M ost s ections a re w hat y ou e xpect from previous labs, however there is more to add for this lab (explained below).

For the first v ersion, `v1`, you s hould o nly b e c oncerned w ith c orrectness. C reate a **single** m utex, o nly f or `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 t o `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`. Your 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
```

Testing. There are a set of basic test cases given to you. We'll withhold more advanced tests which we'll use for grading. Part of programming is coming up with tests yourself. To run the provided test cases please run the following command in your lab directory:

```
python -m unittest
```

Submission.

1. All lab submissions will now take place on BruinLearn. You will find submission links for all labs under the assignment page.
2. The submission format is a single .tar.gz file. This archive should include all files that were given to you in the skeleton(create a tar file from your lab3 directory). Do not inclue any executable, pycache directory etc that are not inclued in the skeleton code directory. You should only modify the skeleton code in this directory. The name of the file should be your student ID with no separators (eg: 4051238888.tar.gz).
3. Your submission will be graded solely based on your last submission to BruinLearn, without any exceptions. Please double check your submission to make sure you submit the correct version of your implementation.

Grading. The breakdown is as follows:

1. 70% code implementation
2. 30% documentation in README.md

Detailed Rubrics

Performance check [50 pts]

Approach for performance/correctness grading:

1. Run the same test for 3 times.
2. If it does not miss any element during all 3 runs, give the correctness grade.

3. If the implementation misses an element even in one of the runs, no credit for either performance or correctness.
4. If the implementation reaches the expected performance in one of the runs, give the correct performance grade.

V1 grading

1. Must be slower than base implementation.
2. Test 1 - High number of elements = 15pts (10pts performance, 5pts correctness).

V2 grading

1. Test 1 - High number of elements, high performance criteria (v2 $\leq$ base/(num cores - 1)) = 55pts (45pts performance, 10pts correctness).
2. If correctness is OK but performance criteria do not match:
3. Try Test 2 - High number of elements, weak performance criteria (v2 $\leq$ base/(num cores / 2)) = 45pts (35pts performance, 10pts correctness).
4. If correctness is OK but performance criteria do not match:
5. Test 3 - Low number of elements, weak performance criteria = 35pts (25pts performance, 10pts correctness).
6. Test-1, Test-2, and Test-3 are mutually exclusive. For example, V2 grading will be done via one of these three test groups.
7. For instance, if correctness was achieved in TestX but performance criteria were not, then we tried TestX+1, but this time the implementation missed some elements, so no credit for correctness.

Documentation [30 pts]:

1. The explanation of using the mutex is reasonable (5 pts for v1; 10 pts for v2, as long as it's reasonable and faster than v0) and consistent with the implementation (5 pts + 5 pts for v1 and v2 respectively). [25 pts]
2. V1 slowness explanation: students must mention thread definition-related overhead. [5 pts]

Code implementation [20 pts]:

1. Wrong locking/unlocking place [5 pts]
2. Forgetting to unlock in the case of existing list entry – causing an unlockable hash table entry [5 pts]
3. No error handling on mutex calls [5 pts]
4. No mutex cleanup [5 pts]