Homework 3 Wet

Due Date: 2/1/2017 23:00

Teaching assistant in charge:

● Gil Kupfer

Important: the Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the piazza forum. These notes are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:

● Read previous Q&A carefully before asking the question; repeated questions will probably go without answers
● Be polite, remember that course staff does this as a service for the students
● You’re not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour
● When posting questions regarding hw3, put them in the hw3 folder

Only Arie, the TA in charge, can authorize postponements. In case you need a postponement, contact him directly.
Introduction

In "Data Structures 1" you have studied several hash table implementations. However all of them had one thing in common – they could only be accessed sequentially by a single thread. In this assignment you will implement a concurrent hash table – a hash table that can be accessed and modified by several threads simultaneously.

Review – Hash Tables with Chaining

A hash table is a data structure designed to support three operations in average time O(1): lookup, insertion and deletion. The hash table itself is an array of m slots (also called buckets), and uses a hash function f() to map elements into those slots. When an element e is inserted into the hash table it is placed in the slot f(e). To look up whether an element e is in the table, we only need to compute f(e) and check whether the element is in that slot.

The problem with this very basic scheme is collisions. A collision happens when two elements, e1 and e2 are hashed into the same slot. There are several ways to resolve this problem, and in this exercise we will use "chaining". In this technique every slot in the hash table contains a linked list of elements.

For example, consider a hash table with 10 slots that contains integers, and the hash function \( f(e) = e \mod 10 \). Suppose the user inserts 5 elements with the values 1, 15, 31, 55 and 57. The resulting hash table will look like this:

![Diagram of hash table with chaining]
Detailed Description

Your hash table must support the following operations:

**Allocation:**

```c
hashtable_t* hash_alloc(int buckets, int (hash)(int, int))
```
This function allocates and initializes a new hash table and returns a pointer to the newly allocated structure.

The parameters passed to the function are:

- *buckets* – the initial number of buckets in the hash table.
- *hash* – the hash function the table will use. The first parameter to hash() is the current number of buckets, and the second the value to be hashed.

**Stop the world:**

```c
int hash_stop(hashtable_t* table)
```
This function prepares hash table for deallocation. hash_free is allowed to be called only after hash_stop was called on the same table.

You should deal gracefully with the situation when there are running operations on the list. All operations already in progress are allowed to finish. Once hash_stop() has started all new operations must fail.

Return values:

- 1: The table was stopped
- -1: Error

**Deallocation:**

```c
int hash_free(hashtable_t* table)
```
This function frees a previously allocated hash table.

Return values:

- 1: The table was freed
- 0: The table wasn't stopped (hash_stop)
- -1: Error

**Single-element insertion:**

```c
int hash_insert(hashtable_t* table, int key, void *val)
```
This function inserts a single element with the value val into the hash table at key. Note that if an element already exists in the hash table, it should not be added again.

Return values:

- 1: The element has been inserted
• 0: The element has not been inserted because it is already in the table
• -1: Error

**Single-element update:**

```c
int hash_update(hashtable_t* table, int key, void *val)
```
This function updates a single element with the key `key` that has the value `val`. Return values:
- 1: The element has been updated
- 0: The element has not been updated because it was not in the table
- -1: Error

**Single-element removal:**

```c
int hash_remove(hashtable_t* table, int key)
```
This function removes a single element with the key `key` from the hash table. Return values:
- 1: The element has been removed
- 0: The element has not been removed because it was not in the table
- -1: Error

**Single-element contains:**

```c
int hash_contains(hashtable_t* table, int key)
```
This function checks whether an element with the key `key` is in the hash table. Return values:
- 1: The element is in the hash table
- 0: The element is not in the hash table
- -1: Error

**Single-element computation:**

```c
int list_node_compute(hashtable_t* table, int key,
                      void *(*compute_func) (void *), void** result);
```
Find the node in the table according to the key and run user-provided function `compute_func` on node's data. Put the result in the user-provided result. Return values:
- 1: The function ran successfully
- 0: The element is not in the hash table
- -1: Error

**Bucket size query**

```c
int hash_getbucketsize(hashtable_t* table, int bucket)
```
This function returns the number of elements currently in bucket number \( b \), or -1 on error.

**Batch operations:**

\[
\text{void hash\_batch(hashtable\_t* table, int num\_ops, op\_t* ops)}
\]

Your hash table must support requests to perform several different operations concurrently. The additional parameters passed to the hash\_batch function are:

- \( \text{num\_ops} \) – the number of operations in the batch
- \( \text{ops} \) - an array of pointers to \( \text{op\_t} \) structures, where each \( \text{op\_t} \) structure represents a single-element operation. The definition of this structure is below:

\[
\text{typedef struct op\_t}
\]

\[
\{ 
\text{int key;}
\text{void *val;}
\text{enum \{INSERT, REMOVE, CONTAINS, UPDATE, COMPUTE\} op;}
\text{void *(*compute\_func) (void *);}
\text{int result;}
\}
\]

All operations in the batch must run concurrently. This means every operation must be executed by a different thread. All of those threads must be created as early as possible – you should not wait for some operations in a batch to complete before starting others.

The return value of each operation should be written to the result field in the corresponding \( \text{op\_t} \) structure. The hash\_batch function returns only after all operation in the batch complete. The header file hashtable.h that defines those required functions and data structures can be found on the course website.

**Implementation Requirements**

All of the above operations may be called concurrently. At any given moment, there might be several single-element operations concurrently with several batch operations, each of those batch operations itself running many single-element operations. Operations involving different elements should not wait for each other unless this is strictly necessary. **This means that you cannot use a single lock for the entire hash table.** Furthermore, since different values may be hashed into the same bucket, **you cannot use a single lock for every bucket.** Therefore you must have a separate lock for each element in the hash table. **Implementations that lock the entire table or bucket to insert or remove a value will not be accepted.**

You should use hand-over-hand locking instead. Hand-over-hand locking is a method of locking elements in a linked structure (such as a linked list), where a previous lock is kept while the next
You can find an illustration (and some sample code in java) at 
http://fileadmin.cs.lth.se/cs/education/eda015f/2013/herlihy4-5-presentation.pdf

You should develop and compile your code on T2 since the virtual machine you have is configured for single-core use.

Important Notes and Tips

- You must implement this exercise in C. Java/C++/Assembly are prohibited.
- Use only standard POSIX threads and synchronization functions.
- Do not change the provided header file.
- We only require you to submit the implementation of your hash table, but not any test programs that use the data structure. Of course, this does not mean you will not need a test program for debugging!
- Insertion and removal from a linked-list that has a lock for each element are not trivial to implement correctly. In particular, you might have to acquire several locks to perform an operation. Think before you start working!
- You should try to make your implementation as concurrent and as efficient as possible. The performance of your solution can affect your grade. However, efficiency must not come at the cost of correctness!
- Make sure your code is running on T2 rather than your virtual machine. We will check it only on T2.

Submission

- You should electronically submit a zip file that contains the source files and the Makefile. Its name should be Makefile. The Makefile will create a single executable named "main".
- You should not submit a printed version of the source code. However, you should document your source code!
- A file named submitters.txt which includes the ID, name and email of the participating students. The following format should be used:

  Linus Torvalds linus@gmail.com 234567890
  Ken Thompson ken@belllabs.com 345678901

Important Note: Make the outlined zip structure exactly. In particular, the zip should contain only the following files (no subdirectories):
If you missed a file and because of this, the exercise is not working, you will get 0 and resubmission will cost 10 points. In case you missed an important file (such as the file with all your logic) we may not accept it at all. In order to prevent it you should open the zip file in a new directory and try to build and test your code in the new directory, to see that it behaves as expected.

Have a Successful Journey,

The course staff