Homework 3 Wet

Due Date: 11/6/2017 23:30

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 this homework you will implement multi-threaded sorted linked list. Each node of the list has a unique field key, (a situation when two nodes have same key is not allowed) that is used to keep the list sorted in an increasing order. Each node also includes opaque pointer data which points to the user defined data.
In your implementation of the list you can add additional fields to the node structure.

In order to keep high concurrency, you will use fine-grained locking. Multiple operations should be able to run simultaneously but are expected to run normally and do not disturb each other.

Detailed Description

Your list should support the following operations:

**linked_list_t** list_alloc();
Create a new list.

**void** list_free(linked_list_t* list);
Free your list.
You should deal gracefully with the situation when there are running operations on the list. All operations already in progress (operations that was started before list_free was called) are allowed to finish and list_free should wait for them to finish.
Once list_free() was called all new operations must fail (especially during the waiting).

**int** list_split(linked_list_t* list, int n, linked_list_t** arr);
Split the list into n new lists alternately. (Assuming n==2: the first element will go to the first list, the second element to the second list, the third element again to the first list and so on.)
The new lists will be stored in arr, assuming it is big enough and the original list will be freed.
If there are fewer than n elements in the original list, list_split should generate empty lists up to a total of n lists.

Like list_free, all operations already in progress are allowed to finish but all new operations must fail.

Returns 0 upon success, !=0 in case of failure.

**int** list_insert(linked_list_t* list, int key, void* data);
Insert new node with data to the list according to the key.
Returns 0 upon success, !=0 in case of failure.

Pay attention that the list should be sorted by the keys of the nodes.
In addition, the user is the responsible for memory allocation of his data.

If the list already has an item with the same key, list_insert should fail.
int list_remove(linked_list_t* list, int key);
Remove the list node with key from the list.
Returns 0 upon success, ≠0 in case of failure.

int list_find(linked_list_t* list, int key);
Check if the list contains node with specific key.
Returns 1 if such node exists, 0 if not, and any other number in case of failure.

int list_size(linked_list_t* list);
Calculate the number of nodes in the list.

int list_update(linked_list_t* list, int key, void* data);
Update data in the list node according to the key.
Returns 0 upon success, ≠0 in case of failure.

int list_compute(linked_list_t* list, int key, int (*compute_func) (void *), int* result);
Find the node in the list according to the key and run user-provided function compute_func on node's data. Put the result in the user-provided result.

Pay attention: compute_func is allowed to change the pointed data, but not the pointer itself.
(list_update does the pointer updates.)

Returns 0 upon success, ≠0 in case of failure.

void list_batch(linked_list_t* list, int num_ops, op_t* ops);
Perform several different operations on the list.
The additional parameters passed to the function are:
● num_ops is the number of operations in the batch
● ops is the array of pointers to the op_t structures. Each op_t structure describes operation on the single list node.

Below is the definition of this structure:
typedef struct op_t {
    int key;
    void* data;
    enum {INSERT, REMOVE, FIND, CONTAINS, UPDATE, COMPUTE} op;
    int (*compute_func) (void *);
    int result;
} op_t;

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 op_t structure. In the case of COMPUTE, the result of compute_func should be written into data (with the appropriate casting) and the result of list_compute into result. The list_batch() function
returns only after all operation in the batch complete. The header file mylist.h that defines those required functions and data structures can be found on the course website.

Implementation Requirements

You should implement your list in one file named “my_list.c”.

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 list. Therefore, you must have a separate lock for each element in the hash table list.

Implementations that lock the list 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 one is being acquired.

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 list, 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 implement your list in one file named “my_list.c”, there is no need for a `makefile` file.
- You should electronically submit a zip file that contains the source file (there is no need to submit the header file).
- 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):

```
zipfile +-  
  |   +- my_list.c 
  |   +- submitters.txt
```

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