Operating Systems – 234123

Homework Exercise 4 – Wet

Teaching Assistant in charge:

Mohammad Agbarya

Assignment Subjects & Relevant Course material

Synchronization, Threads

Recitations 9-10, Lectures 8-9

Assignment Objectives

The objective of this assignment is to give you hands-on experience with parallel programming and various synchronization methods. You are going to implement a synchronization primitive (Barrier) in the first part and in the second part you are required to implement a thread-safe list using some synchronization primitives.

Part I - Barrier

Introduction

An N-process barrier is a synchronization mechanism that allows N threads, where N is a fixed number, to wait until all of them have reached a certain point. Once all threads have reached this certain point (the barrier), they may all continue. For example, we can use barriers to synchronize N people who watch a movie, if each person runs the following thread:

```
buy_popcorn();
buy_something_to_drink();
// wait for everyone else before we start the movie
barrier.wait();
watch_the_movie();
// wait until everyone leaves the cinema
barrier.wait();
close_the_door();
```

In the above example, each thread that calls `barrier.wait()` blocks until the last thread enters the synchronization barrier, and only then all threads are released. Note that the same barrier is used twice to synchronize all threads, so once a thread unblocks, it can re-enter the barrier and block again.

Implementation Requirements

Implement the following Barrier API, as defined in the provided `Barrier.h` header file:

1. **`Barrier(unsigned int num_of_threads);`**
    - This is the constructor of the Barrier. it receives one parameter `num_of_threads` - which is the number of threads that this barrier is supposed to handle.
    - Return value: A Barrier Object.

2. **`void Barrier::Wait();`**
    - Block and wait until N threads reach this point.
      If you're the N^{th} thread to call this method wake up the rest of the threads else, block.
    - Parameters: none.
    - Return value: none.

3. **`unsigned int Barrier::WaitingThreads();`**
    - Returns the number of threads which are currently waiting on this barrier.
    - Parameters: none.
    - Return value: number of threads waiting on the barrier.

4. **`~Barrier();`**
    - This is the deconstructor of the Barrier. destroys or releases any resources that it used (like semaphores and mutexes)
    - Parameters: none.
    - Return value: none.

Important Notes and Tips

- **Don't modify the signatures of the methods defined in the Barrier.h header file.** We will test your code according to this interface.
- You may add any class members you need (integers, Boolean, data structures, …), but **your implementation should use only the following synchronization primitives: semaphores and mutexes.**
  To put it another way, you are not allowed to use any of the following types and their associated functions: `pthread_cond_t`, and obviously not `pthread_barrier_t`.
- You can use as many semaphores as you need.
- In your implementation, you can ignore any errors that the semaphore and mutex functions may return. For instance, according to the man pages, `sem_init()` returns the value EINVAL when the initial value of the semaphore exceeds SEM_VALUE_MAX; we will not check such cases in our tests.
- Use only standard POSIX threads and synchronization functions. You are not allowed, of course, to use the C++11 thread support or the C++11 atomic operations libraries.
- 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 your virtual machine or Azure as was explained and done in previous homework.**

Part II – Thread Safe Linked List

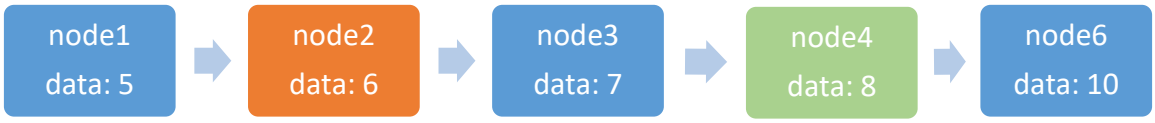
Introduction

In this part you will implement multi-threaded **sorted** linked list (aka thread safe sorted linked list). Each node of the list has a **unique** field **data** which can be considered as the node key, (a situation when two nodes have same data value is not allowed) that is used to keep the list sorted in an increasing order.
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. You must use **hand over hand locking** to achieve that goal.

What is hand over hand locking?

The insertion and removal operations may be called concurrently. Operations involving different elements should not wait for each other unless this is strictly necessary. For example, let's look at the following case:



Let's say we want to do the following operations:

- delete node 2 (orange) with data 6
- insert a new node **after** node 4 (green)

then we should be able to do both operations concurrently.

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 list.**
Implementation that locks the entire list to insert or remove a value will not be accepted.

Instead, you should use **hand-over-hand** locking. 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. Going back to our example above, Hand over hand locking would mean, while traversing the list, we should lock the current node and lock the next one before transitioning to the next node. And in removal of node 2 (orange) we should keep locking node 1 and lock node 2 until we free it and update node 1. (similar idea for insertion).

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
p.s. don't worry about the number of pages it's mostly (if not all) animation.

Implementation Requirements

Implement the following List API, as defined in the provided `ThreadSafeList.h` header file (your implementation should be added to `ThreadSafeList.h` header file and you should not add new source file that implements the header):

1. **`List();`**
   - This is the constructor of our thread safe sorted List.
   - If some operation fails print to std::cerr the following line and exit(-1)

     "<function name>: failed"

   - Return value: List object.

2. **`bool insert(const T& data);`**
   - Insert new node to list **while keeping the list ordered in an ascending order**. If there is already a node has the same data as @param data then return false (without adding it again), use hand over hand locking.
   - Parameters:
     i. data: the new data to be added to the list
   - Returns true if a new node was added and false otherwise

3. **`bool remove(const T& value);`**
   - Remove the node that its data equals to **value.**
   - Parameters
     i. value the data to lookup a node that has the same data to be removed
   - Returns true if a matched node was found and removed and false otherwise.

4. **`unsigned int getSize();`**
   - Returns the current size of the list.
   - You can't use global locking here either (coarse grained locking not allowed). (Hint: but you can maintain a counter)

5. **`~List();`**
   - Destructor to our list, destroy / free any resources that it took.
   - You can assume destroy operations are successful (print some error message to stderr for your own debugging in case destroy operations fail)
   - Parameters: none.
   - Return value: none.

6. **`void print();`**
   - Prints the list in unsafe way. This method is already implemented and provided to you as prat of the ThreadSafeList.h header file. This method will be used in some of ours tests to test your implementation (to print the final state of the list).

- You **<u>should not</u>** modify or remove this method from your submitted file.

Special methods:

```
7. virtual void __add_hook() {}
8. virtual void __remove_hook() {}
```

Those are Test hooks to be used for testing your code (you can take a look into testListHooks.cpp source file which uses them).

Very important Notes:

- You **<u>should not</u>** modify or remove these methods in your submitted file.
- You should call **`__add_hook()`** inside your insert method, after adding the new node and before releasing the lock(s) (i.e., should be called inside the critical section right before the unlock(s)).
- You should call **`__remove_hook()`** inside your remove method, after removing the given node and before releasing the lock(s)) (i.e., should be called inside the critical section right before the unlock(s)).
- Note that the **`__add_hook()`** and **`__remove_hook()`** should be called only if the add and remove methods succeed
- **Pseudo code example how to insert the hooks in your code:**

```
Insert(node){
    Lock()
    //insert new node
    __add_hook()
    Unlock()
}
```

Important Notes and Tips

- **Don't modify the signatures of the methods defined in the ThreadSafeList.h header file.** We will test your code according to this interface.
- Start implementing a sorted linked list (with no locks), test it, make sure that it is working as expected, and only then start applying the hand over hand locking.
- Use thread sanitizers to debug your multi-threaded implementation. You can read more about ThreadSanitizer for C++ here:

  https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
- You may add any class members you need (integers, Boolean, data structures, …), but **your implementation should use only <u>mutexes</u> as synchronization primitives.** To put it another

way, you are not allowed to use semaphores or any synchronization primitives (other than **`pthread_mutex_t`**) that are provided in the pthreads library.

- You can use as many mutexes as you need.
- In your implementation, you can ignore any errors that the pthreads functions may return; we will not check such cases in our tests.
- Use only standard POSIX threads and synchronization functions. You are not allowed, of course, to use the C++11 thread support or the C++11 atomic operations libraries.
- You should try to make your implementation as concurrent and as efficient as possible. You must use hand over hand locking mechanism when inserting or removing from the list. The performance of your solution can affect your grade. However, efficiency must not come at the cost of correctness!
- You should implement the list API mentioned above in the ThreadSafeList.h header file (and not in separated source file) since we are going to build your code with the following command line:
  `g++ --std=c++11 -pthread -Wall -pedantic -Werror -fPIC -pthread -std=c++11 testthreadSafeList.cpp -o tsl`
- **Make sure your code is running on either Azure or your virtual machine as you were instructed in HW0.**

Questions & Answers / Piazza

- The Q&A for the exercise will take place at a public forum Piazza **only**. Please **DO NOT** send questions to the private email addresses of any of the TAs.
- 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 stay updated.
- 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 must discuss such a matter, please come to the reception hours.
- When posting questions regarding **hw4**, put them in the **hw4** folder

Late Days

- Please **DO NOT** send postponement requests to the TA responsible for this assignment. Only the **TA in charge** can authorize postponements. In case you need a postponement, please fill out the attached form : https://forms.gle/bZYjnT2arRrtD6NF6

Submission

- You should edit and submit all source files (including headers) that were provided in the course website under the "HW4 Wet" section.
- You should electronically submit a zip file that contains the above files.
- We only require you to submit the implementation of your classes, but not any test programs that use these classes. Of course, this does not mean you will not need a test program for debugging!
- 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:** the zip should contain only the following files (no subdirectories):

```
zipfile -+
|
         +- Barrier.h
         +- Barrier.cpp
         +- ThreadSafeList.h
         +- submitters.txt
```

- You can create the zip by running the following command line:

```
zip ID1_ID2.zip Barrier.h Barrier.cpp ThreadSafeList.h 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.
- **Important Note:** when you submit, **retain your confirmation code and a copy of the file(s)**, in case of technical failure. Your confirmation code is **the only valid proof** that you submitted your assignment when you did.

Have a Successful Journey,
The course staff