Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Assignment 10: Advanced synchronization mechanisms
Spring Semester 2017

Assigned on: **02.05.2017** Due by: **09.05.2017**

Overview

In this exercise, we take a closer look on how to implement efficient parallel data-structures. We simulate a common scenario in data processing, a parallel producer consumer queue [1]. Based on your knowledge about locks, you will implement the queue using different lock strategies, (e.g., coarse and fine grained locking, optimistic locking, lazy locking) and measure the performance of your implementation.

Getting Prepared

- Download the ZIP file named `assignment10.zip` on the course website.
- Import the project in Eclipse: Click on *File* in the top-menu, then select *Import*. In the dialog, select *Existing Projects into Workspace* under the *General* directory, then click on Next. In the new dialog, select the radiobox in front of *Select archive file* to import a ZIP file. Then, click Browse on the right side of the text-box to select the ZIP file you just downloaded from the website (`assignment10.zip`). After that, you should see assignment10 as a project under *Projects*. Click Finish.
- If you have done everything correctly, you should now have a project named assignment10 in your *Package Explorer*.

Look at the two source files `SortedListInterface` and `SequentialList` in the `assignment10` package. The `SequentialList` is a simple and non thread-safe implementation of `SortedListInterface` which specifies an API for a list that keeps it's elements always in sorted order. The tests in `BenchmarkTest` simulate a producer/consumer scenario with different threads that produce items (by adding them to the list) and consume items (by removing them from the list) as well as doing many read-only accesses to the list by checking if an item is in the list.

Note that for the purpose of this exercise you are not supposed to rely on existing List, Map implementations from the Java standard library for your implementation, but rather you are supposed to implement your own. However, you are of course allowed to study and re-use ideas from the source code of those implementations.

Your task is to implement four thread-safe versions of `SortedListInterface` using different lock strategies:

[1] http://en.wikipedia.org/wiki/Producer- consumer_problem

Coarse Grained Locking

The first implementation is supposed to be a simple and thread-safe version of the `SortedListInterface`, using a single, global lock to make sure update operations do not interfere with each other. Implement the provided template class `CoarseGrainedLockList`.

Coarse grained locking method locks the whole list when a thread requires to *add*, *remove* or check if the list *contains* the method. This is the simplest way of implementing the thread safe linked list, but gives the worst performance.

Fine Grained Locking

As you know, having a big global lock that locks the whole data-structure is simple to implement, but usually tends to hurt performance. In your next implementation of `SortedListInterface`, you will address this by using individual locks for each queue element.

Implement the class `FineGrainedLockList` that uses an individual lock per list element. For your implementation, you should be careful to lock any nodes that you need in order to update the list, but not too many! You need to be sure that between comparison of the list values and changing the list content no other threads can access the data. Therefore, hand-over-hand locking is a good choice.

Optimistic Locking

One way to reduce synchronization costs further is to take a chance: search without acquiring locks, lock the nodes found, and then confirm that the locked nodes are correct. If a synchronization conflict causes the wrong nodes to be locked, then release the locks and start over. Normally, this kind of conflict is rare, which is why we call this technique optimistic synchronization.

Implement the class `OptimisticLockList` that still uses an individual lock per list element. This method typically requires a re-validation if the data-structure has been modified after the lock was acquired. Think about what could go wrong here. *Hint: What can happen between value comparison and locking of the nodes? How can you check if the data was modified?*

Lazy Locking

The `OptimisticLockList` implementation works best if the cost of traversing the list twice without locking is significantly less than the cost of traversing the list once with locking. One drawback of this particular algorithm is that `contains()` acquires locks, which is unattractive if `contains()` calls are much more common than calls to other methods.

Implement `LazyLockList` so that `contains()` calls are wait-free, and `add()` and `remove()` methods, while still blocking, traverse the list only once (in the absence of contention). The trick to implementing lazy locking is to keep track of deleted nodes by adding a Boolean marked field to each node. We maintain the invariant that every unmarked node is reachable which allows us to avoid re-traversing the whole list. If a node is marked, then that node is considered not reachable and therefore considered as not contained in the list. To remove an element, we take two steps: First, logically remove the target node by marking it as removed, and second, physically remove it by redirecting the reference in the predecessor's next field.

Implementation performance

Once you are confident that your implementations are correct, run the `Main` method of the `BenchmarkTest` class, which will benchmark the amount of operations per second for each of your implementation by running a pre-defined workload. In your report, list the numbers you achieve for each of the implementation and briefly discuss your results.

You can also play with the initial parameters `numOps`, `initialElems`, `insertPercentage` in `BenchmarkTest` to see if they impact your results.

It is very difficult to write a JUnit test which detects all possible race conditions with a high probability. `SortedLinkedListTest` tries to detect some errors by adding and removing elements to and from the list with many threads in parallel. You can use the test to find bugs in your implementation but you should not rely on it to find all errors. Feel free to adjust the parameters of the JUnit test or modify it if you have a good idea on how to test your implementation.

Submission

In order for us to grade your exercises and give you feedback, you need to submit your code to the Subversion repository. You will find detailed instructions on how to install and set-up Eclipse for use with Subversion in Exercise 1.

Once you have completed the skeleton, commit it to SVN in a directory named `assignment10` by following the steps described below. The questions that require written answers should all be recorded in a single file named `report.pdf` and placed in the base directory of your project (i.e., in folder `assignment10`).

- **Check-in your project for the first time**
  - Right click your created project called **assignment10**.
  - In the menu go to **Team**, then click **Share Project**.
  - In the dialog that now appears, select **SVN** as a repository type, then click **Next**.
  - In case you have submitted Exercise 1, choose **Use existing repository location** and select the pre-defined URL in the dialog that should look like this
    https://svn.inf.ethz.ch/svn/vechev/pprog17/students/NETHZ_USERNAME
    Click Finish. Otherwise follow the steps in Exercise 1 to set-up a repository location.
- **Commit changes in your project**
  - Now that your project is connected with the SVN server, you need to make sure that every time you change your code or your report, at the end you submit it to the SVN server as well.
  - Right click your project called **assignment10**.
  - In the menu go to **Team**, then click **Commit**.
  - In the Comment field, enter a comment that summarizes your changes.
  - Then, click on **Ok**.