Programming Assignment for Concurrency & Multithreading 2022-2023

Overview

The programming assignment consists of implementing two data structures in Java using different concurrency mechanisms, empirically examining their performance, and writing a report about the implementations and experiments. The data structures that should be implemented are the singly linked list and the binary search tree. Examples of list data structures can be found in [2] or [4], information on binary search trees can be found in [1] or [3].

The assignment is split into two tasks:

- Implementing the data structures using coarse-grained locking algorithms
- Implementing the data structures using fine-grained locking algorithms

Details of these tasks, and a time schedule, can be found below. Note that the time schedule also includes a schedule for the feedback, so you can incorporate the feedback in the next task and your study of the material for the exam.

Prerequisites

First, decide whether you want to do the assignments alone or with someone else (you are encouraged to work in pairs). Then, if you don't already have an account on the DAS5 system, please send mail to `c.verstoep@vu.nl`, and we'll supply one. You'll need this, because you will be using a standard compute node of the DAS5 cluster to test and evaluate your solution.

On the Canvas course webpage, under "Files", you can find a framework for your implementation (`framework.zip`).

During development you can use your own computer, but in the end, your implementation should run on a DAS5 node [1]. Also keep in mind that different operating systems can have different performance characteristics, so be sure to double-check your intermediate results on a DAS5 node. Final measurements for the evaluation part of the assignment should always be done on a DAS5 node.

Using a repository

Please don't use a public repository. Doing so might invite others to copy/steal your work. You can, of course, use a private repository.

Accessing DAS5 from home

If you want to work from home and run on DAS5, you may notice that DAS5 (and actually, most of the VU network) is not directly accessible from home. You will have to go through the VU Stepstone machine, which you can reach using either ssh or putty, using your VUnet id. Say you have a directory "DataStructures" on your local machine. You can then first copy it to ssh.data.vu.nl:

```
scp -r DataStructures <your VUnet id>@ssh.data.vu.nl:
```

[1] `http://www.cs.vu.nl/das5/`

(or something similar with pscp if you use putty on Windows). Note the ':' at the end of the command! This probably asks for your VUnet id password.

Then, login on ssh.data.vu.nl, again using your VUnet id.

```
ssh <your VUnet id>@ssh.data.vu.nl
```

This may again require your VUnet id password. You can now copy the "DataStructures" directory to DAS5 using scp:

```
scp -r DataStructures <your DAS5 account>@fs0.das5.cs.vu.nl:
```

This probably asks for your DAS5 account password.
Then, you can login on DAS5 (from the ssh stepstone machine), and proceed from there:

```
ssh <your DAS5 account>@fs0.das5.cs.vu.nl
```

Structure

Each of the two tasks requires you to

- design and implement a synchronization scheme;
- create hypotheses about performance;
- evaluate the performance of your data structures;
- write a report.

Design and Implementation Implement your data structures in Java using the supplied framework. All data structures implement the `Sorted` interface which provides the methods `add(T t)`, `remove(T t)`, and `toArrayList()`. The data structures need to be sorted according to the ordering of `Comparable<T>`[2]. We provide a basic framework that uses the discussed data structures. It is your task to implement the `add(T t)`, `remove(T t)`, and `toArrayList()` methods according to the requirements (see below). Use a clear coding style.

There are two main differences between the data structures in [2] and the data structures required by the assignment. First, the list data structure presented in the book is sorted according to the hash value of the stored items, whereas we ask you to sort the list according to the `Comparable<T>` interface. Furthermore, the data structures in the book do not allow duplicate elements, whereas all data structures in this assignment should allow and represent duplicate elements.

The `toArrayList()` method should return an ArrayList containing the elements of the data structure, in the order indicated by the data structure. It does not have to be thread-safe. You should focus on implementing the `add()` and `remove()` methods. You may assume that `toArrayList()` will not be invoked during concurrent updates.

The framework also provides a test script that you can use to test your implementation, see below.

Hypotheses Hypothesize about the performance and scalability of your data structure. Try to qualitatively estimate the performance of the data structures by taking into account the number of elements in the data structure, the number of threads operating on the data structures, and the amount of work a thread is doing in comparison to inserting or removing an element. Relate to previous data structures if possible. Make sure to support your claims.

Evaluation Evaluate the performance of your data structures using a node of the DAS5 cluster. Intermediate testing of your implementation should mostly be done on your local machine, but you are encouraged to do test runs on a DAS5 node, because a DAS5 node may have more cores and that may affect the behaviour of your implementation. In the evaluation section of your report you assess the performance of the data structures and compare them to both your earlier stated hypotheses, and the performance of previous data structures. Your evaluation should be supported with quantitative measurements, for example using graphs and tables. Use reasonable element counts: a sequential run (a run with 1 thread) should take somewhere between a couple of seconds and a couple of minutes to allow you to measure speedups. Also use reasonable numbers of threads, considering the processor that you run your measurements on. Explain your results. Do the measurements correspond with your expectations? And if not, why not? Also, experiment with the workload: how do different workloads affect speedup?

[2] `http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html`

Report Your report should include:

- a short discussion of your design considerations;
- the hypotheses;
- the evaluation.

Please be concise: you don't have to explain what a linked list or a binary tree is. Your report for each task should be approximately 2-4 pages of text (not including raw measurement data and graphs).

Assessment Your final grade depends on the quality of your implementation, your hypotheses and their justification, and the thoroughness of your report, but also on the completeness of your submission for the first task. Please make sure that your code can be easily understood and uses a clear coding style. Note that we will test your implementation thoroughly, using the framework as provided. In your final grade, task 1 counts for 40%, task 2 counts for 60%. For each task, the code counts for 40%, the report counts for 60%.

Framework The provided framework takes care of most of the program, allowing you to concentrate on the implementation of the methods of the `Sorted` interface. The framework will first create a specified number of threads, have these threads add elements in random order, synchronize the threads using a barrier, and have the threads remove the same elements in a different order. After this, the data structure should be empty. The provided main program does 10 runs, and prints the time of each run and the average time.

In order to ease the debugging process, the framework seeds the random number generator based on the arguments you pass to it, ensuring that the same set of parameters always results in the same input to your algorithm.

We also provide a run script located in `bin/run_data_structures` to start the framework, with Integers. The script detects when it is run on the head node of the DAS5 cluster, and will in this case automatically submit the run to one of the DAS5 compute nodes instead. Executing the script without parameters displays a short manual. There also is a run script `bin/run_data_structures2`, that does the same, with Strings. Finally, there is a script `bin/test_data_structures` that exercises your implementations a bit.

Task 1: Coarse-grained data structures

For this task you are required to implement the two data structures using coarse-grained locking. Coarse-grained locking means that you first write a sequential implementation, and then ensure that only one thread can access your data structure simultaneously.

- Design and implement the coarse-grained data structures using the framework.
- Hypothesize on the performance of the coarse-grained data structures.
- Evaluate the performance of the coarse-grained data structures.

CoarseGrainedList This CoarseGrainedList needs to be a singly-linked list. The elements are ordered in a monotonically non-descending order. This means that the list may contain duplicate elements. Adding or removing an element should lock the complete data structure. Figures 9.4 and 9.5 in [2] show examples of update methods of a linked list with coarse-grained locks.

CoarseGrainedTree This tree is a binary search tree that allows duplicate elements. Elements are stored at internal nodes and leaf nodes of the tree. The binary search tree does not have to re-balance itself. Again, adding or removing an element should lock the complete datastructure.

Task 2: Fine-grained data structures

For this task you are required to implement the two data structures using fine-grained locking, See section 9.5 of [2]. Fine-grained locking is much more error-prone than coarse-grained locking, so test your implementation thoroughly! We will do so as well ...

- Design and implement the fine-grained data structures using the framework.
- Hypothesize on the performance of the fine-grained data structures.
- Evaluate the performance of the fine-grained data structures. Also compare with the performance of the coarse-grained data structures!

FineGrainedList This singly-linked list should not lock the complete data structure, but only parts of it to allow concurrent updates on different parts of the list. Figure 9.6 and 9.7 in [2] show examples of updates on a list with fine-grained locks.

FineGrainedTree This binary search tree is similar to a CoarseGrainedTree, but uses fine-grained locks, so locks at the nodes of the tree. Concurrent updates on different parts of the tree should be possible.

Timeline

See the table below. The tasks must be submitted before 23:59 on the day they are due. For the first assignment, you will receive feedback within a week, so you can take it into account for the second assignment. **Deadlines are strict!** If you are unable to finish the assignment in time, at least submit your unfinished work before the deadline.

#	**Task**	**Date**	**Feedback due**
1	Coarse-grained	September 30^{th}	October 7^{th}
2	Fine-grained	October 21^{th}	November 4^{th}

Submission procedure

Submissions should be done on Canvas. You have to submit both your code and your report. Create the report as a PDF file, put it in the pre-created "report" directory. Then use the command "./gradlew submit" to create a submit.zip file that you can submit via Canvas. Make sure that your name(s) and VU-net-ID(s) are on the report. If you work in pairs, one of you should submit.

Note that your program should run without any modification to the original framework (except for the implementation package, of course). In particular, you may not change the `Sorted` interface, since we will test your implementations in ways not provided in the framework. You are allowed to modify the framework for e.g. debugging purposes, but we will use the original framework for the testing of your code.

References

[1] Michael T Goodrich and Roberto Tamassia. *Algorithm Design: Foundation, Analysis and Internet Examples*. John Wiley & Sons, 2006.

[2] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear. *The Art of Multiprocessor Programming (second edition)*. Morgan Kaufmann Pub, 2021.

[3] Wikipedia. Binary Search Tree (`http://en.wikipedia.org/wiki/Binary_search_tree`). [Online; accessed 23-July-2020].

[4] Wikipedia. Linked List (`http://en.wikipedia.org/wiki/Linked_list`). [Online; accessed 23-July-2020].