NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO DE
INFORMÁTICA

DEPARTAMENT OF
COMPUTER SCIENCE

Multiple Synchronization Strategies in a Concurrent Hash-Map

João M. Lourenço

May 05, 2021

Resumo

With this lab assignment you shall understand the tradeoffs between the costs and benefits of using different locking strategies.

1 Introduction

In this project you are given a running Java application, available at

```
git clone https://bitbucket.org/joaomlourenco/concurrent_hashmap.git
```

This application launches multiple threads (the number of threads is specified as the second command line argument) operating over a shared data structure (a hash map with collision lists), by inserting, removing and looking up for elements. After a certain time (specified in milliseconds as the first command line argument) the application prints some statistics and terminates.

You may import the project to Eclipse and work from there. If you prefer to work from the command line, go to the application "root directory" (you shall have three subdirectories: "bin", "src", and "resources") and run the command below.

```
find src -name '*.java' -print | xargs javac -d bin
```

To run the application, execute the command below, where the first argument is the time the application will run (in milliseconds) and the second is the number of threads. The output of the execution is typeset in italic.

```
java -cp bin cp/articlerep/Main 1000 1
```

Campus de Caparica
2829-516 CAPARICA | Portugal
T: +351 212 948 536
di.secretariado@fct.unl.pt
di.fct.unl.pt

And you will get an output similar to the following

```
Starting application...
Total operations: 4644330
Total successful puts: 585344 (13%)
Total successful removes: 575931 (12%)
Total successful gets: 1151131 (25%)
Throughput: 4644330 ops/sec
Finished application.
```

2 Lab Work

2.1 Add automatic validation

Nearby line 82 of the `Main.java` file you can find a *sanity check*, which implements some basic verification of the final state of out concurrent hash map. However, this validation is too weak and passing this validation does not mean that there were no concurrency conflicts.

Add a more elaborated automatic validation to your program. For this you must:

1. Think about the invariants of the hash map and of the application;
2. Identify which of those invariants may be broken when running the application with multiple threads;
3. Change the current *sanity check* to implement checks for the identified invariants. Depending on the additional invariants you plan to check, it may be necessary to change the interface and the implementation of the hash map to include an additional method "`validate()`" to check the invariants of the hash map data structure and report to the caller;

> **Before you start fixing the concurrency issues in the hash map, be sure that you develop a set of strong and solid tests, and that your program fails those tests frequently.**

2.2 Implement different locking strategies to fix the concurrency issues

Create four different implementations of the hash map, each one using a different locking strategy:

1. Create a solution that implements a **coarse grain locking strategy** using the **Java construct** `synchronized`. With this implementation there is a single global lock and only one thread at a time will be allowed to access the hash map.

2. Create a solution that implements a **coarse grain locking strategy** using a **single plain (exclusive) lock** from the `java.util.concurrent.locks` package. With this implementation there is also a single global lock and only one thread at a time will be allowed to access the hash map.

3. Create a solution that implements a **coarse grain locking strategy** using a **single read-write lock** from the `java.util.concurrent.locks` package. With this implementation, many lookup operations (`get`) may access the hash map simultaneously, but update operations (`put` and `remove`) must do it one at a time.

4. Create a solution that implements a **medium grain locking strategy** (i.e., one lock for each collision list of the hash map) using **plain (exclusive) locks** from the `java.util.concurrent.locks` package. With this implementation, many operations may access the hash map simultaneously as long as they are operating on different collision lists.

5. Create a solution that implements a **medium grain locking strategy** (i.e., one lock for each collision list of the hash map) **using read-write locks** from the `java.util.concurrent.locks` package. With this implementation, many operations may access the hash map simultaneously as long as they are operating on different collision lists or they are multiple `get` operations on the same collision list.

6. ***OPTIONAL:*** *Create a solution that implements a* ***fine grain locking strategy****, namely* ***hand-over-hand, optimistic and/or lazy sychronization****, to control the concurrent accesses to the hash map collision lists.*

2.3 Evaluate the effectiveness of each locking strategy

Run tests to evaluate the correctness of each of your solutions from above. If your implementations prove correct, then run ore tests to evaluate the scalability of the application when you increase the number of threads. Run experiments with $1, 2, 4, \ldots, N$ threads, where N is the double of the number of processors/cores you have available in your development computer.

NOTE: if you are using some kind of vitualized environment, please be sure that you have more than one processor assigned to your virtual execution environment.

2.4 Acquire a deeper understand of your Java program

Install the "`visualvm`" (`https://visualvm.github.io/features.html`) tool with

```
apt get install visualvm
```

and then use this tools to analyse the multiple versions of the concurrent hash map.

2.5 To Think About...

The sequential (unprotected) version is much faster than any lock-based solution. Will any of your solutions ever exhibit a speedup instead of a slowdown? How many processors do you need for that?