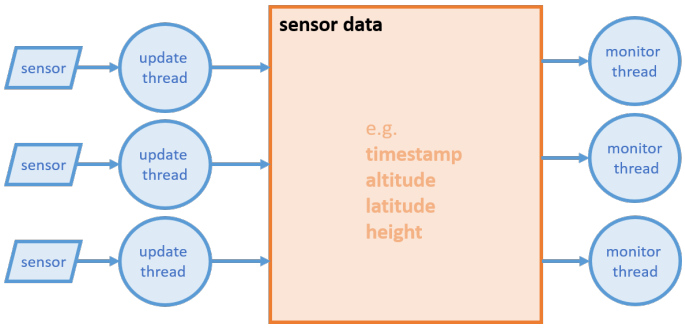
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

**Parallel Programming**
**Assignment 11: Advanced synchronization mechanisms**
**Spring Semester 2017**

Assigned on: **09.05.2017** Due by: **16.05.2017**

# Lock Free Sensor System

This exercise is about the lock-free implementation of a data record. We consider the following scenario: Assume you have a number of sensors that deliver complex data. It is considered clear from the beginning that retreiving the data from a sensor and in particular writing the data to some data structure cannot happen atomically. In this exercise we consider a sensor that delivers floating point data and an integer timestamp. You may want to think of a GPS sensor delivering, say, altitude, longitude and height together with the current time in milliseconds. In order to keep the system reliable and responsive, redundant data sources are installed, i.e. we have a lot of sensors that provide the same kind of data.




Now, we assume we have a real lot of concurrent reader threads that want to monitor the sensor data while we have a moderate number of writer threads. What we consider most important is that the reader threads do always read a consistent data set. Secondly, we require that only the newest sensor data are written to the sensor data object. This is where the timestamp becomes important.

## Implementation

Implement two versions of the senor data set:

**a)** One blocking version based on a readers-writers lock (*LockedSensors.java*).

**b)** A lock-free version (*LockFreeSensors.java*)

*Hints:*

**a)** Before you implement the readers-writers lock based version, start with a simple locked version in order to understand. Then try a readers-writers lock but be aware that the Java-implementation does not give fairness guarantees. What can this imply? In any case, you have the code from the lecture slides presenting a fair RW-Lock implementation.

**b)** The lock-free implementation solutions does NOT rely on mechanisms such as Double-Compare-And-Swap. Also it does not rely on a lazy update mechanism. Somehow you have to make sure that with a single reference update you change all data at once. How?

Compare the efficiency of the two solutions. Experiment with different numbers of readers and writers and different timings in order to find out under which conditions which version is better. Start with the template that we provide.

Note: When testing your implementation using the provided Unit Tests make sure that you run the tests multiple times (i.e. 5-10 times). This increases the chance of finding an error in your implementation as the Unit Tests can succeed even with an invalid implementation due to randomness of scheduling.

### Questions

**a)** Is your lock-free algorithm also wait-free? Please explain your answer.

**b)** Think about the solution you have provided: can you generalize this? Imagine, for example, the scenario of an expression tree that is updated by (a small amount of) writer threads sporadically but is used for evaluation by a huge number of reader threads. Can you use the same kind of mechanism?

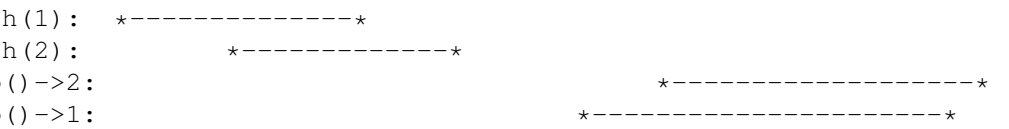
## Linearizability and Sequential Consistency

In the following questions let A, B and C denote threads operating on a shared stack object as specified in the listing below.

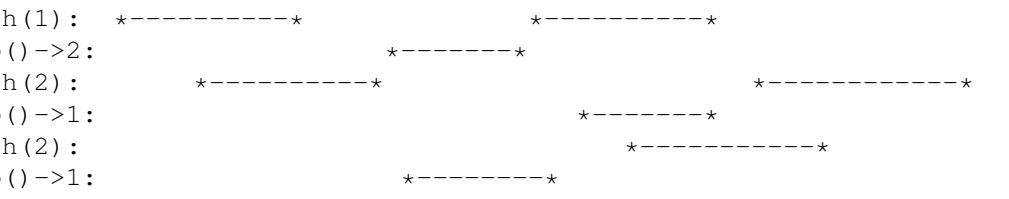
```
public interface Stack {
    /* pushes element v onto the stack */
    public void push(int v);

    /* removes the top element and returns it */
    public int pop();

    /* returns the top element */
    public int top();
}
```

Which of the following scenarios on the next page are lineraly consistent? Either mark the point of linearization or explain why it is not linearly consistent.

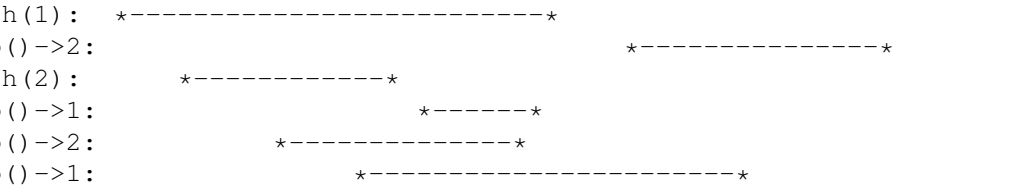
**a)**



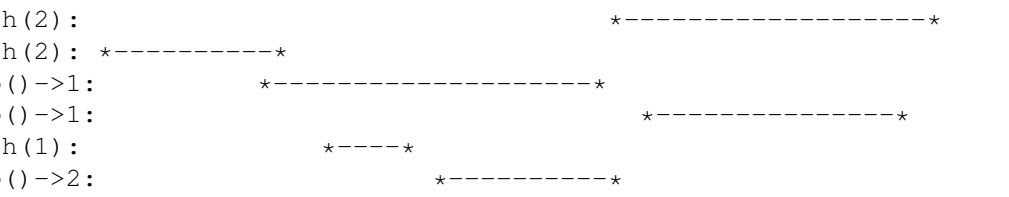

**b)**




**c)**




**d)**




# 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 `assignment11` 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 `assignment11`).

- **Check-in your project for the first time**
  - Right click your created project called **assignment11**.
  - 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 **assignment11**.
  - In the menu go to **Team**, then click **Commit**.
  - In the Comment field, enter a comment that summarizes your changes.
  - Then, click on **Ok**.