Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Assignment 13: Transactional Memory
Spring Semester 2020

Assigned on: **18.05.2020** Due by: **25.05.2020**

Overview

This week's assignment is about Transactional Memory. Transactional Memory tries to simplify synchronization for the programmer and place it in a hardware or software system. This assignment is intended to give you a hands-on introduction on how to use transactional memory, which you learned about during the lecture. We will be using using ScalaSTM, which is a library-based implementation of software transactional memory. (Note: despite the name you will not program in Scala, but rather still write code using the Java API.)

Exercise 1 – Circular Buffer with STM

For this exercise you are asked to implement a bounded queue, similar to `ArrayBlockingQueue` from the Java concurrency library which was used in the Bakery Simulator scenario. You will need to implement a few functions as listed below:

```
interface CircularBuffer<E> {
    // Adds item to the queue.
    // If full, this blocks until a slot becomes free.
    public void put(E item);

    // Retrieves the oldest item from the queue.
    // If empty, this blocks until an item is present.
    public E take();

    public boolean isEmpty();
    public boolean isFull();
}
```

We now briefly explain how to use the ScalaSTM framework. The main idea is that you declare the critical sections in your code as atomic blocks:

```
class Account {
  ...
    void withdraw(int amount) {
        STM.atomic(new Runnable() { public void run() {
            // critical section -> executed atomically
            }
        });
    }
  ...
}
```

What the STM implementation does is that it tracks all memory accesses (read/write) done inside the atomic block. Once the transaction completes, the framework checks for interference, makes sure all values are consistent and either applies them all atomically (*commit*) or rolls back the entire block and retries again later (*abort*). For this reason, all shared data objects should be either immutable or they need to be wrapped in a `Ref`. An example of how to create this in Java is:

```
Ref.View<Integer> count = STM.newRef(0);
```

Warning: it is important you do not access a `Ref` from outside a transaction (i.e. only from *within* an atomic block). In Scala, this can be enforced by the compiler but the same is not available for Java.

There are many operations you can invoke on a `Ref`. The ones you will need for the exercise are: `.get()`, `.set()` and `STM.increment`. To represent the buffer of items in the queue you can use:

```
TArray.View<E> items = STM.newTArray(capacity);

// ... and to access an element at index 'i' use:
Ref.View<E> item = items.refViews().apply(i);
```

If you need to wait, for instance because the queue is empty or full, call `STM.retry()`. This will block the calling thread until the reader/writer set has changed, at which point the transaction will be restarted automatically.

For more information have a look at the API documentation:

https://nbronson.github.io/scala-stm/api/0.8/scala/concurrent/stm/japi/STM$.html

Submission

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

Once you have completed the skeleton, commit it to Git by following the steps described below. For the questions that require written answers, please write them on paper and bring them to the next exercise session where the solutions will be discussed.

- **Check-in your project for the first time**
  - Right click your created project called **assignment13**.
  - In the menu go to **Team**, then click **Share Project**.
  - You should see a dialog "Configure Git Repository". Here, next to the Repository input field click on **Create...**
  - Select a root git directory or your projects that you have created in Execise 1. Note for all your assignments you should use the same directory.
  - click **Finish**.
- **Commit changes in your project**
  - Now that your project is connected to your git repository, you need to make sure that every time you change your code or your report, at the end you commit your changes and send (push) them to the git server.
  - Right click your project called **assignment13**.
  - In the menu go to **Team**, then click **Commit...**.

  - In the Comment field, enter a comment that summarizes your changes.
  - In the Files list, select all the files that you changed and want them to be committed. This typically includes all the Java files but not necessarily all the files (e.g., you dont have to commit setting files of our eclipse installation).
  - Then, click on **Commit** to store the changes locally or **Commit and Push** to also upload them to the server. Note that in order to submit your solution you need to **both** commit and push your changes to the server.

- **Push changes to the git server**
  - Right click your project called **assignment13**.
  - In the menu go to **Team**, then click **Push Branch 'master'**. Note if this is not your fist push you can also use **Push to Upstream** to speed up the process.
  - A new dialog appears, now fill in for the URL field:
    `https://gitlab.inf.ethz.ch/COURSE-PPROG20/<nethz-username>.git`
  - Click **Next**
  - Keep the default values and click **Next**
  - An authentication dialog should appear. Fill in your nethz username and password and click **OK**.
  - Click **Finish** to confirm your changes. Note that eclipse might ask for authentication again.

- **Browse your repository online**
  - you can access and browse the files in your repository online on GitLab at:
    `https://gitlab.inf.ethz.ch/COURSE-PPROG20/<nethz-username>`