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

To implement a circular buffer we use an array of some size (the size is given in the constructor) to store the items in the buffer. Since we need to access the array from within transactions, we cannot use a normal Java array, instead we need to use an array type from ScalaSTM: TArray.

```
import scala.concurrent.stm.Ref;
import scala.concurrent.stm.TArray;
import scala.concurrent.stm.japi.STM;

public class CircularBufferSTM<E> implements CircularBuffer<E> {

    private TArray.View<E> items;

    CircularBufferSTM(int capacity) {
        items = STM.newTArray(capacity);
    }

}
```

To track where (at what array index) the next element should be put, and from where the oldest element should be taken, we add additional private fields to our CircularBufferSTM class. Since we also need to keep track of how many elements are in the queue, we also add a counter for the elements in the array. These are not "normal" integers, as we will also access them from within transactions, but rather references to integers tracked by STM. Thus we can declare them as final (since the reference will always point to the same object):

```
    private final Ref.View<Integer> count = STM.newRef(0);
    private final Ref.View<Integer> putIndex = STM.newRef(0);
    private final Ref.View<Integer> takeIndex = STM.newRef(0);
```

STM allows us to set the initial value of the underlying integer upon creation of the reference, we set all the variables to zero.

Now to implement the "put" method, which places an item into the CircularBufferSTM we need to do the folowing in a single atomic transaction:

- Check if the array is full. If yes, we retry the transaction (so for the caller this method blocks) until there is space available. Note that we do not need any back-off etc. to avoid livelocks — "retry" is smart enough to not actually do anything until one of the Ref's we read before calling it has changed.
- Place the item at "putIndex" into the Array.
- Incrementing "putIndex" (but we need to be careful, since this could "wrap-arround", so we increment and then use the remainder of the index divided by the size).
- Incrementing "count".

```
public boolean isFull() {
    return STM.atomic(new Callable<Boolean>() {
        public Boolean call() { return count.get() == items.length(); }
    });
}

private int next(int i) {
    return (i + 1) \% items.length();
}

public void put(final E item) {
    STM.atomic(new Runnable() {
        public void run() {
            if (isFull())
                STM.retry();
            items.update(putIndex.get(), item);
            putIndex.set(next(putIndex.get()));
            STM.increment(count, 1);
        }
    });
}
```

The implementation of "take" is analogous:

```
public E take() {
    return STM.atomic(new Callable<E>() {
        public E call() {
            if (isEmpty())
                STM.retry();
            E item = items.refViews().apply(takeIndex.get()).get();
            items.update(takeIndex.get(), null);
            takeIndex.set(next(takeIndex.get()));
            STM.increment(count, -1);
            return item;
        }
    });
}

public boolean isEmpty() {
    return STM.atomic(new Callable<Boolean>() {
        public Boolean call() { return count.get() == 0; }
    });
}
```