Prof. P. Thiemann, L. Fennell, M. Geffken Winter Term 2014/15

Lecture: Concurrency Theory and Practise

`http://proglang.informatik.uni-freiburg.de/teaching/concurrency/2014ws/`

Exercise Sheet 2

November 28, 2014

I. Theory

I.1. Consistency

An informal definition of *quiescent consistency* can be found in the AMP book, Section 3.3. Additionally, we give a more formal definition below, which is presented in the sytle of the definition for *linearizability* and *sequential consistency* from the slides, such that quiescent consistency can be related to the aforementioned alternative concepts. First, we need two auxiliary definitions.

Definition 1. *Two histories H and G are* up-to-order-equivalent *if all threads see the same executions up to the order of executions. More formally, for every thread A, the thread projections $H|A$ and $G|A$ are equal up to the order of executions.*

Definition 2. *Given a history H and method executions m_o and m_1 in H, we say $m_o \rightarrow_H m_1$ if m_o precedes m_1 and the method calls are separated by a period of quiescence (i.e., at some point after the end of m_0 and before the start of m_1 there is no pending method call).*

Now we are ready to define quiescent consistency.

Definition 3. *History H is* quiescently consistent *if it can be extended to G by*

- *Appending zero or more responses to pending invocations*
- *Discarding other pending invocations*

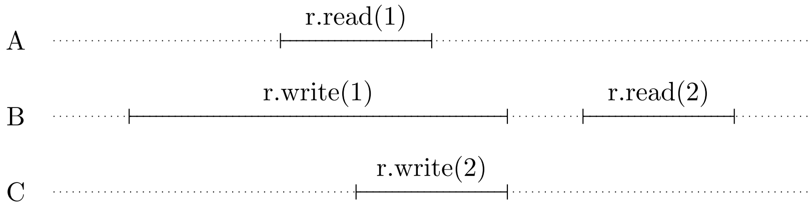
so that G is up-to-order-equivalent *to a legal sequential history S where $\rightarrow_G \subseteq \rightarrow_S$.*

Informally, the definition says that any time an object becomes quiescent, then the execution so far is equivalent to some sequential execution of the completed calls.

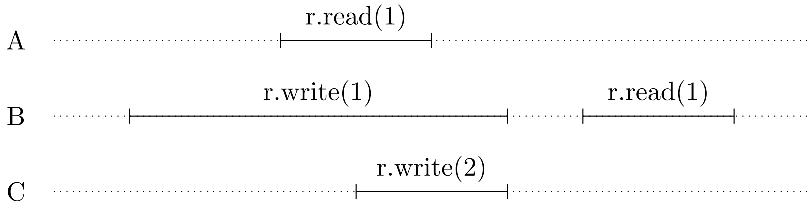
Exercise Give an example of an execution that is quiescently consistent but not sequentially consistent, and another that is sequentially consistent but not quiescently consistent.

I.2. Classifying histories

For each of the histories shown, are they quiescently consistent? Sequentially consistent? Linearizable? Justify your answer. History 1:



History 2:



I.3. Atomic Integers

The AtomicInteger class (in the `java.util.concurrent.atomic` package) is a container for an integer value. One of its methods is `boolean compareAndSet(int expect, int update)`. This method compares the object's current value to `expect`. If the values are equal, then it atomically replaces the object's value with `update` and returns true. Otherwise, it leaves the object's value unchanged, and returns false. This class also provides `int get()` which returns the object's actual value.

Consider the FIFO queue implementation:

```
class IQueue<T> {
  AtomicInteger head = new AtomicInteger(0);
  AtomicInteger tail = new AtomicInteger(0);
  T[] items = (T[]) new Object[Integer.MAX VALUE];
  public void enq(T x) {
    int slot ;
    do {
      slot = tail.get();
    } while (! tail.compareAndSet(slot , slot+1));
    items [slot] = x;
  }
  public T deq() throws EmptyException {
    T value;
    int slot;
    do {
      slot = head.get();
      value = items[slot];
      if (value == null)
        throw new EmptyException();
    } while (! head.compareAndSet(slot , slot+1));
    return value;
  }
}
```

It stores its items in an array `items`, which, for simplicity, we will assume has unbounded size. It has two `AtomicInteger` fields: `tail` is the index of the next slot from which to remove an item, and head is the index of the next slot in which to place an item. Give an example showing that this implementation is not linearizable.

I.4. Strange methods . . .

Definition 4. *A method is* bounded wait-free *if there is a bound on the number of steps a method call can take (from the AMP book, p. 57).*

Consider the following rather unusual implementation of a method m. In every history, the ith time a thread calls m, the call returns after 2^i steps. Is this method m wait-free, bounded wait-free, or neither?

II. Practice

II.1. Timestamp interface

Give Java code to implement the Timestamp interface of Fig. 2.10 in the AMP book using unbounded labels. For convenience, we repeat the interface definitions here.

```java
public interface Timestamp {
  /**
   * @return true if "this" is greater than "other".
   */
  boolean compare(Timestamp other);
}

/**
 * A TimestampSystem manages an array of Timestamps.
 */
public interface TimestampSystem {

  /**
   * @return the current array of timestamps
   */
  public Timestamp[] scan();

  /**
   * Update a timestamp.
   * @param timestamp The new timestamp.
   * @param i         The index into the Timestamp array to be updated.
   */
  public void label(Timestamp timestamp, int i);
}
```

Then, show how to replace the pseudocode of the Bakery lock of Fig. 2.9 in the AMP book (or on the slides) using your Timestamp Java code.

Submission

- Deadline: **27.11.2014, 23:59**
- Submit theory exercises in PDF format via email to `concurrency AT informatik.uni-freiburg.de`. Please name your single file with the scheme: `ex2-name(s).pdf`.
- Submit practical exercises as executable jar-files for each exercise. The file name should include the name of the exercise and your name (example: `philosophers-fennell.jar`). Make sure that you include all source files and libraries you use. Sources should always be documented!
- Late submissions may not be corrected.
- Do not forget to write your name(s) on the exercise sheet.
- You may submit in groups up to 2 people.