TU WIEN Informatics Parallel Computing

Research Group for Parallel Computing
Faculty of Informatics
TU Wien

Advanced Multiprocessor Programming

Summer term 2023
Theory exercise 2 of 2

Issue date: 2023-04-23
Due date: 2023-05-08 (23:59)

Total points: 42

1 Consistency Criteria

Ex 1.1 (2 points)

Give an example of a sequentially-consistent execution that is not safe.

Ex 1.2 (2 points)

Consider a *memory object* that encompasses two register components. We know that if both registers are quiescently consistent, then so is the memory. Does the converse hold? If the memory is quiescently consistent, are the individual registers quiescently consistent? Outline a proof, or give a counterexample.

Ex 1.3 (4 points)

Suppose x, y, z are registers with initial values 0. Is the history in Figure 1 quiescently consistent, sequentially consistent, linearizable? For each consistency criterion, either provide a consistent execution or a proof sketch.

Does there exist an initial valuation for the registers x, y, z such that your above result w.r.t. linearizability changes? (Meaning if you found the history to be linearizable, can you find initial values for x, y, z s.t. it no longer is linearizable; conversely if you found the history to be not linearizable, can you find initial values s.t. it becomes linearizable?) If so, provide the initial values and either a proof sketch or an example execution.

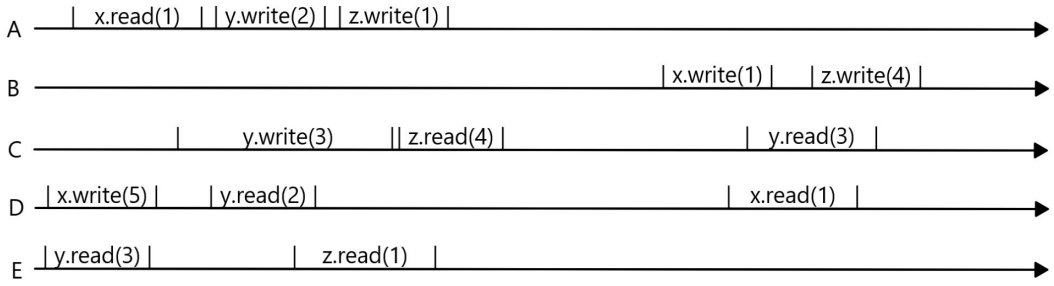

Figure 1: History with five threads and three registers

Ex 1.4 (4 points)

Suppose x, y, z are stacks and w is a FIFO queue. Initially the stacks and the queue are all empty. Is the history in Figure 2 quiescently consistent, sequentially consistent, linearizable? For each consistency criterion, either provide a consistent execution or a proof sketch.

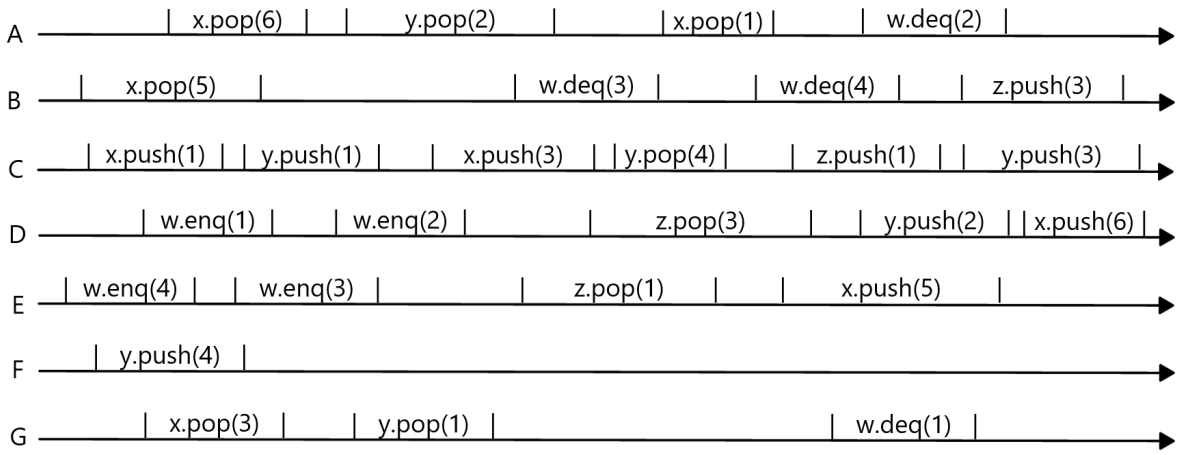

Figure 2: History with seven threads three stacks and one queue

Ex 1.5 (2 points)

The `AtomicInteger` class (in **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 shown in Figure 3. It stores its items in an array `items`, which for simplicity we will assume has unbounded size. It has two `AtomicInteger` fields: `head` is the index of the next slot from which to remove an item and tail is the index of the next

slot in which to place an item. Give an example execution showing that this implementation is *not* linearizable and provide a short explanation as to why.

```
1  class IQueue<T> {
2    AtomicInteger head = new AtomicInteger(0);
3    AtomicInteger tail = new AtomicInteger(0);
4    T[] items = (T[]) new Object[Integer.MAX_VALUE];
5    public void enq(T x) {
6      int slot;
7      do {
8        slot = tail.get();
9      } while (! tail.compareAndSet(slot, slot+1));
10     items[slot] = x;
11   }
12   public T deq() throws EmptyException {
13     T value;
14     int slot;
15     do {
16       slot = head.get();
17       value = items[slot];
18       if (value == null)
19         throw new EmptyException();
20     } while (! head.compareAndSet(slot, slot+1));
21     return value;
22   }
23 }
```

Figure 3: Non-linearizable queue implementation

Ex 1.6 (4 points)

This exercise examines a queue implementation that can be seen in Figure 4, whose `enq()` method does not have a linearization point.

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail;
4    static final int CAPACITY = 1024;
5
6    public HWQueue() {
7      items =(AtomicReference<T>[])Array.newInstance(AtomicReference.class,
8          CAPACITY);
9      for (int i = 0; i < items.length; i++) {
10       items[i] = new AtomicReference<T>(null);
11     }
12     tail = new AtomicInteger(0);
13   }
14   public void enq(T x) {
15     int i = tail.getAndIncrement();
16     items[i].set(x);
17   }
18   public T deq() {
19     while (true) {
20       int range = tail.get();
21       for (int i = 0; i < range; i++) {
22         T value = items[i].getAndSet(null);
23         if (value != null) {
24           return value;
25         }
26       }
27     }
28   }
29 }
```

Figure 4: Queue implementation

The queue stores its items in an `items` array, which for simplicity we will assume is unbounded. The `tail` field is an `AtomicInteger`, initially zero. The `enq()` method reserves a slot by incrementing `tail` and then stores the item at that location. Note that these two steps are not atomic: there is an interval after `tail` has been incremented but before the item has been stored in the array.

The `deq()` method reads the value of `tail`, and then traverses the array in ascending order from slot zero to the tail. For each slot, it swaps *null* with the current contents, returning the first non-*null* item it finds. If all slots are *null*, the procedure is restarted.

Give an example execution showing that the linearization point for `enq()` cannot occur at line 15. (*hint: give an execution, where two* `enq()` *calls are not linearized in the order they execute line 15*)

Give another example execution showing that the linearization point for `enq()` cannot occur at line 16.

Since these are the only two memory accesses in `enq()`, we must conclude that `enq()` has no single linearization point. Does this mean `enq()` is not linearizable?

2 Consensus

Ex 2.1 (6 points)

Show that with sufficiently many n-thread binary consensus objects and atomic registers one can implement n-thread consensus over n values.

Ex 2.2 (6 points)

The `Stack` class provides two methods: `push(x)` pushes a value onto the top of the stack and `pop()` removes and returns the most recently pushed value. Prove that the `Stack` class has consensus number *exactly(!)* two.

Ex 2.3 (2 points)

Suppose we augment the FIFO `Queue` class with a `peek()` method that returns but does not remove the first element in the queue. Show that the augmented queue has infinite consensus number.

Ex 2.4 (2 points)

Consider three threads, A, B, C, each of which has a MRSW register, X_A, X_B, X_C, that it alone can write and the others can read.

In addition each pair shares a `RMWRegister` register that provides only a `compareAndSet()` method: A and B share R_{AB}, B and C share R_{BC}, A and C share R_{AC}. Only the threads that share a register can call that register's `compareAndSet()` method – which is the only way to interact with said register.

Your mission: either give a consensus protocol and prove that it works or sketch an impossibility proof.

Ex 2.5 (6 points)

Consider the situation described in Ex 2.4 except that A, B, C can apply a `doubleCompareAndSet()` operation to both of their shared registers at once. Regarding the semantics and signature of this new operation consider two variations [1]:

a) Suppose the signature is

`bool doubleCompareAndSet(RMWRegister shReg1, RMWRegister shReg2, int expect, int update)`.

For a call of `doubleCompareAndSet(X, Y, exp, up)`, atomically the following is done: it checks if the value in `X` and `Y` is equal to `exp`. If both are equal to `exp`, `X` and `Y` are updated with `up` and `true` is returned. Otherwise no update is conducted and it returns `false`. The equivalent description in code can be seen in Listing 1.

b) Suppose the signature is

`TwoBool doubleCompareAndSet(RMWRegister shReg1, RMWRegister shReg2, int expect, int update)`,

where `TwoBool` is a simple class consisting of two (publicly accessible) `bool` values.

[1] If any of the `doubleCompareAndSet()` variations is called by a thread for at least one register, which it does not have access to, we assume the method throws an `IllegalAccess` exception with no further side effects.

```
class TwoBool { public bool val1; public bool val2; }
```

Consider the semantics given by Listing 2.

Listing 1: doubleCompareAndSet() - variation 1

```
1  bool doubleCompareAndSet(RMWRegister shReg1, RMWRegister shReg2, int expect, int update)
2  {
3      bool retVal = false;
4      atomic
5      {
6          if(shReg1 == expect && shReg2 == expect)
7          {
8              shReg1 = update;
9              shReg2 = update;
10             retVal = true;
11         }
12     }
13     return retVal;
14 }
```

Listing 2: doubleCompareAndSet() - variation 2

```
1  TwoBool doubleCompareAndSet(RMWRegister shReg1, RMWRegister shReg2, int expect, int update)
2  {
3      TwoBool retVal;
4      atomic
5      {
6          retVal.val1 = shReg1.compareAndSet(expect, update); // the 'usual' CAS operation on a
7          retVal.val2 = shReg2.compareAndSet(expect, update); // single register
8      }
9      return retVal;
10 }
```

For both semantics of the `doubleCompareAndSet()` operation either provide a consensus protocol and prove that it works or sketch an impossibility proof. (Note that threads can still use their shared register's normal `compareAndSet()` method!)

Ex 2.6 (2 points)

The `SetAgree` class, like the `Consensus` class provides a `decide()` method whose call returns a value that was the input of some thread's `decide()` call. However unlike the `Consensus` class, the values returned by `decide()` calls are not required to agree. Instead these calls may return no more than k distinct values. (When k is 1, `SetAgree` is the same as consensus.)

What is the consensus number of the `SetAgree` class when $k > 1$? Sketch a proof!