Assignment 8: Correctness conditions

ETH Zurich

1 Stack

1.1 Background

Figure 1 shows a history for three threads. Each time line corresponds to one thread. All the threads work on a single stack s. The query s.top (i) expects an element i to be on top of stack s. Note that s.top (i) does not remove the top item. The command s.push (i) pushes an element i on top of the stack s.

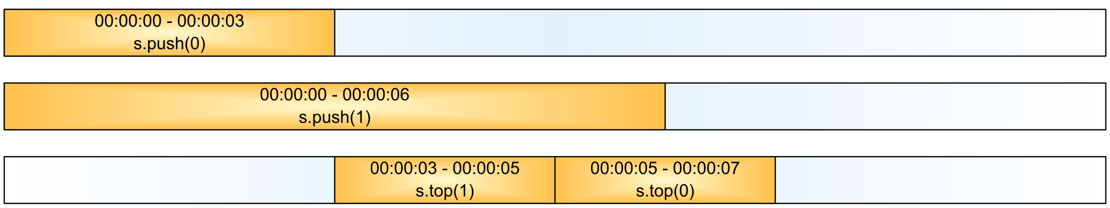

Figure 1: History

1.2 Task

1. Is the history shown in figure 1 linearizable? Justify your answer.
2. Is the history shown in figure 1 sequentially consistent? Justify your answer.

2 Non-linearizable queue

2.1 Background

This task has been adapted from [1]. The *AtomicInteger* class 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 following FIFO queue implementation. 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. *tail* is the index of the next slot in which to place an item.

```
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;
  }
}
```

2.2 Task

Give an example showing that this implementation is not linearizable.

3 Binary search tree

3.1 Background

Listing 1 shows the class of a binary search tree. The class defines a feature *insert* to add a value to a tree and a feature *has* to check whether the tree contains a value.

Listing 1: Non-linearizable binary search tree

```
class BINARY_SEARCH_TREE

create
  make

feature -- Initialization
  make (a_value: INTEGER)
      -- Initialize this node with 'a_value'.
    do
      left := Void
      right := Void
      value := a_value
    end

```

```
feature -- Access
  left: BINARY_SEARCH_TREE
      -- The left sub tree.
  right: BINARY_SEARCH_TREE
      -- The right sub tree.
  value: INTEGER
      -- The value.

feature -- Basic operations
  insert (a_new_value: INTEGER)
      -- Insert 'a_new_value' into the tree.
    require
      tree_does_not_have_new_value: not has (a_new_value)
    do
      if a_new_value < Current.value then
        if not left = Void then
          left.insert (a_new_value)
        else
          left := create {BINARY_SEARCH_TREE}.make (a_new_value)
        end
      else
        if not right = Void then
          right.insert (a_new_value)
        else
          right := create {BINARY_SEARCH_TREE}.make (a_new_value)
        end
      end
   end

  has (a_value: INTEGER): BOOLEAN
      -- Does the tree have 'a_value'?
    do
      if a_value = Current.value then
        Result := True
      else
        if a_value < Current.value then
          if not left = Void then
            Result := left.has (a_value)
          else
            Result := False
          end
        else
          if not right = Void then
            Result := right.has (a_value)
          else
            Result := False
          end
        end
      end
    end
end
```

3.2 Task

1. Devise an execution sequence that demonstrates that the binary search tree from Listing 1 is not linearizable; provide a corresponding history and explain why this history is non-linearizable.

2. Using the feature *compare_and_swap*, develop a linearizable version of the binary search tree class. Provide only the changed features.

   The feature *compare_and_swap* (*$entity*, *test_value* , *new_value*) sets the value of an entity to *new_value* if and only if the entity currently has the value *test_value*; the feature call returns whether or not the test was successful. Here, the *$* operator returns the address of the entity.

4 Practical sequential consistency

4.1 Background

One of the implicit simplifying assumptions behind many of the example programs presented in the course has been that sequential consistency is being followed. Recall that sequential consistency essentially means that the relative ordering of operations between threads does not have to be maintained, but the per-thread ordering of operations should be kept. However, this assumption is invalidated quite easily by both compilers and hardware without careful attention.

Compilers are free to reorder the instructions given in the program text, given that it does not change the output of the sequential program.

For example:

```
a := 1
b := 2
```

can be rewritten to

```
b := 2
a := 1
```

if the compiler thinks it would be faster, as the output of the sequential program is the same in either case.

4.2 Task

Consider this one-shot Peterson locking algorithm:

```
enter1 := true
turn := 2
if not enter2 or turn = 1 then
  critical section
  enter1 := false
end
```

How does this locking algorithm break if the compiler (or CPU) can reorder reads and writes to independent variables? To see how, it may help to rewrite the algorithm so that intermediate expressions are computed and stored into temporary variables, for example, turning $a + 1 = b$ into

```
tmp1 := a + 1
tmp2 := tmp1 = b
```

5 Linearizability

5.1 Background

The following listing shows the class for a lock that is based on a faulty implementation of the bakery algorithm. The class defines a feature *lock* to acquire the lock and a feature *unlock* to release the lock. The feature *owns_lock* returns whether a thread currently owns the lock. The feature *make* creates the lock for threads $1 \ldots n$.

```
class BAKERY_LOCK create make

feature {NONE} -- Implementation and initialization
  numbers: ARRAY [INTEGER] -- The numbers.
  is_lock_owner: ARRAY [BOOLEAN] -- Which thread is the lock owner?

  make (n: INTEGER)
      -- Initialize this lock for threads 1 to 'n'.
    do
      create numbers.make (1, n); numbers.fill_with (0)
      create is_lock_owner.make (1, n); is_lock_owner.fill_with (False)
    end

feature -- Basic operations
  lock (i: INTEGER)
      -- Acquire this lock for thread 'i'.
   do
     numbers [i] := 1 + max (numbers)
     for all j /= i do
       await numbers [j] = 0 or (numbers [i], i) < (numbers [j], j)
     end
     is_lock_owner [i] := True
   end

  unlock (i: INTEGER)
      -- Release this lock for thread 'i'.
    do
      number [i] := 0
      is_lock_owner [i] := False
    end

  owns_lock (i: INTEGER): BOOLEAN
      -- Is thread 'i' the lock owner?
    do Result := is_lock_owner [i] end
end
```

5.2 Task

1. Devise an execution sequence that demonstrates that the bakery lock from the listing is non-linearizable; provide a corresponding history and explain why this history is non-linearizable.

2. Suggest a fix to make the bakery lock linearizable. Explain why the fix solves the issue.

References

[1] Maurice Herlihy und Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.