Assignment 7: Lock-free approaches

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.

| 00:00:00 - 00:00:03 |
| s.push(0) |

| 00:00:00 - 00:00:06 |
| s.push(1) |

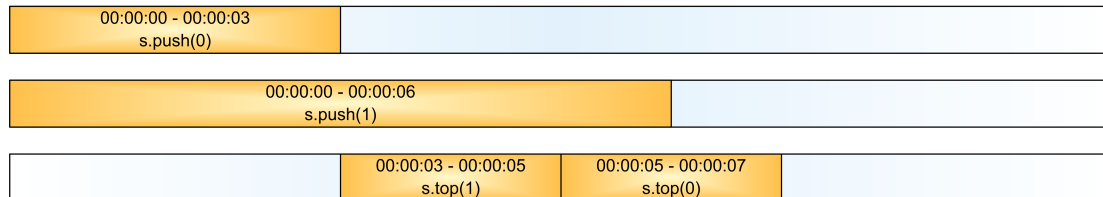| 00:00:03 - 00:00:05 | 00:00:05 - 00:00:07 |
| s.top(1) | s.top(0) |

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 [2]. 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);
```

```java
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

```eiffel
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
```

```eiffel
   feature −− Access
16    left : BINARY_SEARCH_TREE
           −− The left sub tree.
18    right : BINARY_SEARCH_TREE
           −− The right sub tree.
20    value: INTEGER
           −− The value.
22
   feature −− Basic operations
24    insert (a_new_value: INTEGER)
           −− Insert 'a_new_value' into the tree.
26      require
           tree_does_not_have_new_value: not has (a_new_value)
28      do
         if a_new_value < Current.value then
30         if not left = Void then
              left . insert (a_new_value)
32         else
              left := create {BINARY_SEARCH_TREE}.make (a_new_value)
34         end
         else
36         if not right = Void then
              right . insert (a_new_value)
38         else
              right := create {BINARY_SEARCH_TREE}.make (a_new_value)
40         end
         end
42      end

44    has (a_value: INTEGER): BOOLEAN
           −− Does the tree have 'a_value'?
46      do
         if a_value = Current.value then
48         Result := True
         else
50         if a_value < Current.value then
             if not left = Void then
52             Result := left.has (a_value)
             else
54             Result := False
             end
56         else
             if not right = Void then
58             Result := right.has (a_value)
             else
60             Result := False
             end
62         end
         end
64      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
2   b := 2
```

can be rewritten to

```
    b := 2
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
2   turn := 2
    if not enter2 or turn = 1 then
4     critical section
      enter1 := false
6   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
2   tmp2 := tmp1 = b
```

It may also help to review the proof of mutual exclusion given in slides for lecture 3.

5 Atomic update of multiple values

5.1 Background

A online game with thousands of players features a daily high score. The high score consists of the player's name and the score he or she achieved. Profiling determined that the current lock-based implementation is a bottleneck.

5.2 Task

You are asked to provide a prototype of a lock-free solution, pseudo-code is sufficient. You can use an integer for the score. Provide a routine to update the high score if the new score is better and a routine to retrieve the current high score. If you need additional data structures, describe them as well.

You are free to use the *compare_and_swap*-routine from task 3.2.

References

[1] CAS-Based Lock-Free Algorithm for Shared Deques. 9th Euro-Par Conference on Parallel Processing. Maged M. Michael 2003.

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

6 Money transaction system

6.1 Background

A money transaction system handles many transactions concurrently. A status monitor displays the total turnover and total number of transactions. Updating these numbers is a sequential bottleneck and should be done as quickly as possible, hence a lock-free solution is desired. Every query of these numbers should return a consistent result, so that the accurate average can be computed.

In finance, precision is very important. This can only be guaranteed by using a big decimal type for the total turnover. A BigDecimal is not a basic type, but supports all the common operations.

In order to avoid problems with an over-eager optimizing compiler or virtual machine, all variables that are modified concurrently should be marked with the **volatile** keyword.

6.2 Task

Your task is to implement this status monitor by providing the necessary data structure(s), a method *updateTotal* to add a transaction and a method *fetchLatest* to retrieve a consistent copy of the current state.

The framework provides you with compare-and-set routines for all (basic and reference) types of variables. This routine is implemented using the atomic operations of the processor and as such is always compiled inline. The following pseudocode demonstrates its function:

```
boolean cas(target, oldvalue, newvalue) {
    // target, oldvalue and newvalue have to be of the same type.
    synchronized {
        if ( target == oldvalue) {
            // This change is propagated outside the current scope.
```

```
            target = newvalue;
            return true;
        }
        return false;
    }
}
```