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.

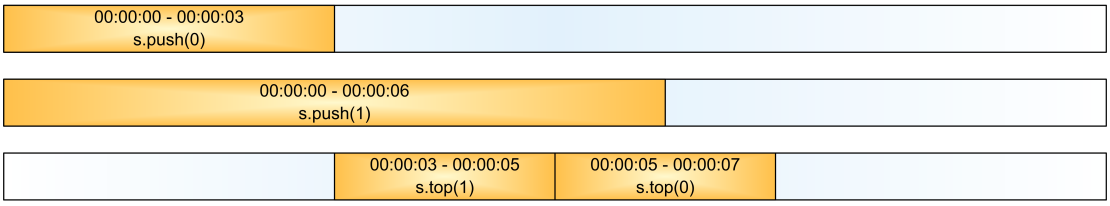

Figure 1: History

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.

1.3 Solution

1. The history is not linearizable. The call to s.push(1) must happen before s.top(1), because s.top(1) expects an element i on top of the stack. The call to s.push(0) must not happen before the call to s.top(1) took effect. The earliest point when s.top(1) can take effect is at the start of its time span. This is however already too late for s.push(0) to take effect.
2. The history is sequentially consistent. An equivalent legal sequential history is given by:
   (a) Thread 2 s.push(1)
   (b) Thread 2 s:void
   (c) Thread 3 s.top(1)
   (d) Thread 3 s:1
   (e) Thread 1 s.push(0)
   (f) Thread 1 s:void
   (g) Thread 3 s:top(0)
   (h) Thread 3 s:0

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.

2.3 Solution

The problem is that the two consecutive operations *tail*.*compareAndSet*(*slot*, *slot* + 1) and *items*[*slot*] = *x* do not happen atomically. The problem is illustrated in the following execution.

1. Thread 1 calls *q*.*enq*(*e1*) with a matching element *e1*.

2. Thread 1 executes $tail.compareAndSet(slot, slot + 1)$ and exits the loop.
3. Thread 2 calls $q.enq(e2)$ with a matching element $e2$.
4. Thread 2 executes $tail.compareAndSet(slot, slot + 1)$, exits the loop and executes $items[slot] = x$.
5. Thread 2 finishes the call to $q.enq(e2)$.
6. Thread 3 calls $q.deq()$.
7. Thread 3 executes $slot = head.get()$ and $value = items[slot]$. The slot is the one set by thread 1. Note that thread 1 did not set the value for this slot yet.
8. Thread 3 throws an exception because *value* is **null**.
9. Thread 1 executes $items[slot] = x$.
10. Thread 1 finished the call to $q.enq(e1)$.

The execution is depicted in the following history.

1. Thread 1 q.enq(e1)
2. Thread 2 q.enq(e2)
3. Thread 2 q:void
4. Thread 3 q.deq()
5. Thread 3 q:EmptyException()
6. Thread 1 q:void

The history is not linearizable. In the history above the call by thread 2 precedes the call by thread 3. This precedence relation must be preserved in any equivalent sequential history. Furthermore such a history can not have any other dequeue operations other than the one by thread 3. Therefore any such history would be invalid because a dequeue operation after a enqueue operation must not throw an empty exception in the absence of other dequeue operations.

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
10      left := Void
        right := Void
12      value := a_value
    end
14
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.

3.3 Solution

3.3.1 Task 1

Consider the following execution based on an entity n of type *BINARY_SEARCH_TREE*:

1. Thread 1: Calls **create** *n.make* (10) and finishes.
2. Thread 1: Calls *n.insert* (5). Continues until *left* := **create** {*BINARY_SEARCH_TREE*}.*make* (*a_new_value*).
3. Thread 2: Calls *n.insert* (4) and finishes.
4. Thread 1: Executes *left* := **create** {*BINARY_SEARCH_TREE*}.*make* (*a_new_value*). This overrides the value set by thread 2.
5. Thread 3: Executes *n.has* (4). Returns that the element could not be found.

The following history represents this execution:

1. Thread 1: create n.make (10)
2. Thread 1: n:void
3. Thread 1: n.insert (5)
4. Thread 2: n.insert (4)
5. Thread 2: n:void
6. Thread 1: n:void
7. Thread 3: n.has (4)
8. Thread 3: n:False

There are only two possible equivalent sequential histories that preserve the order relation. One of the histories lets the first thread insert its value first:

1. Thread 1: create n.make (10)
2. Thread 1: n:void
3. Thread 1: n.insert (5)
4. Thread 1: n:void
5. Thread 2: n.insert (4)
6. Thread 2: n:void
7. Thread 3: n.has (4)
8. Thread 3: n:False

The other one lets the second thread insert its value first:

1. Thread 1: create n.make (10)
2. Thread 1: n:void
3. Thread 2: n.insert (4)
4. Thread 2: n:void
5. Thread 1: n.insert (5)
6. Thread 1: n:void
7. Thread 3: n.has (4)
8. Thread 3: n:False

Both histories are illegal because the value 4 should not be reported as missing. Hence the implementation is not linearizable.

3.3.2 Task 2

Listing 2: Linearizable binary search tree

```
insert (a_new_value: INTEGER)
    -- Insert 'a_new_value' into the tree.
require
    tree_does_not_have_new_value: not has (a_new_value)
local
    l_cached_sub_tree : BINARY_SEARCH_TREE -- The cached sub tree.
do
    if a_new_value < Current.value then
        l_cached_sub_tree := left
        if not l_cached_sub_tree = Void then
            left.insert (a_new_value)
        else
            if not compare_and_swap ($left, l_cached_sub_tree, create {
                BINARY_SEARCH_TREE}.make (a_new_value)) then
                left.insert (a_new_value)
            end
        end
```

```
    else
18      l_cached_sub_tree := right
        if not l_cached_sub_tree = Void then
20        right.insert (a_new_value)
        else
22        if not compare_and_swap ($right, l_cached_sub_tree, create {
              BINARY_SEARCH_TREE}.make (a_new_value)) then
            right.insert (a_new_value)
24        end
        end
26    end
  end
```

The binary search tree is linearizable. When an insert operation runs in parallel to a number of insert operations, then the tree will contain the values of all these insert operations. When a has operation runs in parallel to a number of insert operations then the has operation might return true or false for the values of these insert operations. When the has operation runs after a number of insert operations then the has operation returns true for the values of these insert operations. For all other values the has operation returns false.

We now take a look at a history, by going from left to right; we look at the operations in the order in which they start. If multiple operations start at the same time, we look at these operations in a random order. As we go through the history, we construct a new sequential history, starting from an empty one.

- If the current operation is an insert operation then we add the insert operation to the end of the sequential history.
- If the current operation is a has operation that returns true for a value from an already added insert operation then we add the has operation to the end of the sequential history.
- If the current operation is a has operation that returns true for a value from a not yet added insert operation then the has operation must have run in parallel to a matching insert operation. In this case, we stall the has operation and add it after the insert operation as soon as we add the insert operation. This preserves the order relation because the insert operation ran in parallel to the has operation and hence we can choose the ordering between the two in the sequential history.
- If the current operation is a has operation that returns false and there is no insert operation that inserts this value then we add the has operation to the end of the sequential history.
- If the current operation is a has operation that returns false and there is an insert operation that inserts this value then either this insert operation must have run in parallel to the has operation or it must have run after the has operation.
  - In case the insert operation ran in parallel to the has operation, we have to distinguish two situations. One one hand, it could be that the insert operation has already been added, in which case we add the has operation in front of the insert operation. This preserves the order relation because the insert operation ran in parallel to the has operation and hence we can choose the ordering between the two in the sequential history. On the other hand, it could be that the insert operation has not been added, in which case we add the has operation to the end of the sequential history.
  - In case the insert operation ran after the has operation, we add the has operation to the end of the sequential history.

Because we add operations in the order in which they started with the only exception of operations that ran in parallel, the history produced in this way is an equivalent sequential history that preserves the order relation. The history is legal because no has operation returns true for a value that has not been added or false for a value that has been added. Hence, the binary search tree implementation is linearizable.

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:

```
1  a := 1
   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
```

4.3 Solution

Because the accesses to *enter1* and *enter2* are to independent locations, these can be reordered. For the first processor (similarly for the second) the program could be rewritten as

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

Now, both processes will calculate *tmp* as *True*, since initially both entry variables are *False*. Therefore, both will enter the critical section at the same time, violating mutual exclusion.

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?
```

```
34      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.

5.3 Solution

Consider the following execution based on an entity l of type *BAKERY_LOCK*:

1. Thread 1: Calls **create** $l.make$ (2) and finishes.
2. Thread 1: Calls $l.lock$ (1) and continues until it evaluated $1 + max(numbers)$. The evaluation results in 0.
3. Thread 2: Calls $l.lock$ (2) and continues until it evaluated $1 + max(numbers)$. The evaluation results in 0.
4. Thread 2: Assigns 1 as its number.
5. Thread 2: Passes the wait statement because thread 1 has 0 as its number.
6. Thread 2: Finishes $l.lock$ (2).
7. Thread 1: Assigns 1 as its number.
8. Thread 1: Passes the wait statement because both threads have the same number, and thread 1 has a lower identifier than thread 2.
9. Thread 1: Finishes $l.lock$ (1).
10. Thread 1: Calls $l.owns\_lock$ (1) and finishes it with a positive result.
11. Thread 2: Calls $l.owns\_lock$ (2) and finishes it with a positive result.
12. Thread 1: Calls $l.unlock$ (1) and finishes.
13. Thread 2: Calls $l.unlock$ (2) and finishes.

The following history represents this execution:

1. Thread 1: create l.make (2)
2. Thread 1: l:void
3. Thread 1: l.lock (1)
4. Thread 2: l.lock (2)
5. Thread 2: l:void
6. Thread 1: l:void
7. Thread 1: l.owns_lock (1)

8. Thread 1: l:True
9. Thread 2: l.owns_lock (2)
10. Thread 2: l:True
11. Thread 1: l.unlock (1)
12. Thread 1: l:void
13. Thread 2: l.unlock (2)
14. Thread 2: l:void

There are only a few possible equivalent sequential histories that preserve the order relation. In all of them, both lock checks must occur after both lock operations; furthermore, both unlock operations must occur after both lock check operations. None of these histories is valid because in all of them two different threads own the lock at the same time, and this is against the validity constraints of the lock.

The problem can be fixed by noting when a thread is choosing its number. A thread can then only check the other numbers if no other thread is currently choosing a number. This is shown in the following listing:

```
class BAKERY_LOCK create make

feature {NONE} -- Implementation and initialization
  numbers: ARRAY [INTEGER] -- The numbers.
  choosing: ARRAY [BOOLEAN] -- Which thread is choosing his number?
  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 choosing.make (1, n); numbers.fill_with (False)
      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
     choosing [i] := True
     numbers [i] := 1 + max (numbers)
     choosing [i] := False
     for all j /= i do
       await choosing [j] = False
       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
34      is_lock_owner [i] := False
    end
36
  owns_lock (i: INTEGER): BOOLEAN
38      -- Is thread 'i' the lock owner?
    do Result := is_lock_owner [i] end
40 end
```

References

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