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 \( s.\text{top}(i) \) expects an element \( i \) to be on top of stack \( s \). Note that \( s.\text{top}(i) \) does not remove the top item. The command \( s.\text{push}(i) \) pushes an element \( i \) on top of the stack \( s \).

![Figure 1: History](image)

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 \( \text{AtomicInteger} \) class is a container for an integer value. One of its methods is \( \text{boolean compareAndSet}(\text{int} \ \text{expect}, \ \text{int} \ \text{update}) \). This method compares the object’s current value to \( \text{expect} \). If the values are equal, then it atomically replaces the object’s value with \( \text{update} \) and returns \( \text{true} \). Otherwise, it leaves the object’s value unchanged, and returns \( \text{false} \). This class also provides \( \text{int get()} \) which returns the object’s actual value.

Consider the following FIFO queue implementation. It stores its items in an array \( \text{items} \), which, for simplicity, we will assume has unbounded size. It has two \( \text{AtomicInteger} \) fields. \( \text{head} \) is the index of the next slot from which to remove an item. \( \text{tail} \) is the index of the next slot in which to place an item.

```java
class IQueue<\text{T}> { 
    \text{AtomicInteger head = new AtomicInteger(0);} 
    \text{AtomicInteger tail = new AtomicInteger(0);} 
```
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

```java
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
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
     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
42
44  has (a_value: INTEGER): BOOLEAN
     -- Does the tree have 'a_value'?
46  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
62 end
64
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
```

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