Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Solution 11: Advanced synchronization mechanisms
Spring Semester 2017

Assigned on: **09.05.2017** Due by: **16.05.2017**

Lock-Free Sensors

The trick used in the solution is that data is always copied for each update and only the reference to the object containing the data is used for atomic operations. This mechanism is known as "copy-on-write". It is, for example, also used in file-systems in order to guarantee consistency of the underlying file system structures at all times. It is also related to the "read-copy-update" mechanism used in the Linux kernel that allows fine-grained concurrent operations by readers and writers on complex data structures.

a) Is your lock-free algorithm also wait-free? Please explain your answer.
Solution: yes, the lock-free algorithm is wait free because of the time-stamps. Any trial to write a value to the sensor data object will fail the latest when there are newer data available. If there are no newer data available, it will fail within finite time also because no threads provide further values.

b) Think about the solution you have provided: can you generalize this? Imagine, for example, the scenario of an expression tree that is updated by (a small amount of) writer threads sporadically but is used for evaluation by a huge number of reader threads. Can you use the same kind of mechanism?
Solution: yes, the lock free algorithm from above was a so called copy-on-write algorithm. The basic trick is that for updating the data only a single reference was changed. This would in principle also work with the expression tree.

Solutions

1, 2, 3 are linearizable. 4 is not. (1 pt each) see next page.

a)
```
A s.push(1):  ⋆---x----------⋆
B s.push(2):         ⋆-----x-------⋆
A s.pop()->2:                                   ⋆---x---------------⋆
B s.pop()->1:                              ⋆---------------x------⋆
```

b)
```
A s.push(1):  ⋆---x------⋆              ⋆-x--------⋆
A s.pop()->2:                  ⋆-x-----⋆
B s.push(2):       ⋆---x------⋆                       ⋆----------x-⋆
B s.pop()->1:                              ⋆-----x-⋆
C s.push(2):                                  ⋆-------x---⋆
C s.pop()->1:                   ⋆----x---⋆
```

c)
```
A s.push(1):  ⋆--------------x-----------⋆
A s.pop()->2:                                  ⋆----------x----⋆
B s.push(2):      ⋆-------x----⋆
B s.pop()->1:                    ⋆--x---⋆
C s.top()->2:           ⋆-------------x⋆
D s.top()->1:                ⋆--x--------------------⋆
```

d)
```
C s.push(2):                                 ⋆-------------------⋆
A s.push(2): ⋆-----x----⋆
B s.top()->1:          ⋆-----------x--------⋆
A s.pop()->1:                                  ⋆---------------⋆
C s.push(1):               ⋆--x-⋆
D s.pop()->2:                     ⋆-------#--⋆
```

The operation with # cannot be satisfied assuming the operations in C and D do not overlap.