# PCPP IT University of Copenhagen, F2025

# Exercises week 7

Last update: 2025/10/05

## Goal of the exercises

The exercises aim to give you practical experience on:

- Reasoning about correctness of concurrent objects using linearizability.
- Identifying and reasoning about linearization points in implementations of non-blocking data structures.
- Reasoning about non-blocking progress notions.

## Syntax for concurrent executions, sequential executions and linearizations

For the answers in the exercises below, you may use the mathematical notation for histories in Herlihy, Chapter 3. However, to avoid getting into the details of the mathematical notation, you may use the following syntax to provide concurrent executions, sequential executions and linearizations. We describe the syntax by example. This syntax is also used in the Linearizability section of the concurrency note.

**Concurrent executions** Consider the concurrent execution at the top of slide 47 in lecture 7. The following syntax represents the execution:

```
A: -|  q.enq(x)  |-------|  p.enq(y)  |---->
B: -----|  q.deq(x)  |------|  p.enq(y)  |->
```

The text in between `|` `|` denotes the method call. The left `|` denotes the point in time when the method was invoked. The right `|` denotes the point int time when the response of the method call is received (in other words, when the method call finished). The width of `|` `|` denotes the duration of the method call. On the left hand side, `A:` and `B:` are thread names. In this case, the upper execution corresponds to thread `A` and the lower execution to thread `B`. The dashed line represents real time.

**Sequential execution** In slide 47 (lecture 7), we provide a (possible) sequential execution for the concurrent execution above. The sequential execution can be written using the following syntax:

```
<q.enq(x),p.enq(y),q.deq(x),p.deq(y)>
```

The symbol < denotes the beginning of the execution, and the symbol > denotes the end of the execution. In a sequential execution, the sequence of method calls are listed in the (sequential) order of execution. Recall that real-time is irrelevant for sequential executions. We are only interested on whether a method call happens before another.

**Linearizations** Linearizations have the same syntax as sequential executions. Remember that the process of finding a linearization involves setting linearization points and map them to a sequential execution. For instance, the linearization of slide 52 (lecture 7) for the concurrent execution above is written as:

```
<q.enq(x),q.deq(x),p.enq(y),p.deq(y)>
```

Ula

**Exercise 7.1** In this exercise, we look into several concurrent executions of a FIFO queue. Your task is to determine whether they are sequentially consistent or linearizable (according to the standard specification of a sequential FIFO queue).

*Mandatory*

1. Is this execution sequentially consistent? If so, provide a sequential execution that satisfies the standard specification of a sequential FIFO queue. Otherwise, explain why it is not sequentially consistent.

```
A: ---------------|q.enq(x)|--|q.enq(y)|->
B: ---|q.deq(x)|------------------------->
```

2. Is this execution (same as above) linearizable? If so, provide a linearization that satisfies the standard specification of a sequential FIFO queue. Otherwise, explain why it is not linearizable.

```
A: ---------------|q.enq(x)|--|q.enq(y)|->
B: ---|q.deq(x)|------------------------->
```

3. Is this execution linearizable? If so, provide a linearization that satisfies the standard specification of a sequential FIFO queue. Otherwise, explain why it is not linearizable.

```
A: ---|       q.enq(x)           |-->
B: ------|q.deq(x)|---------------->
```

4. Is this execution linearizable? If so, provide a linearization that satisfies the standard specification of a sequential FIFO queue. Otherwise, explain why it is not linearizable.

```
A: ---|q.enq(x)|-----|q.enq(y)|-->
B: --|        q.deq(y)           |->
```

*Challenging*

5. Is this execution sequentially consistent? If so, provide a sequential execution that satisfies the standard specification of a sequential FIFO queue. Otherwise, explain why it is not sequentially consistent.

```
A: -|p.enq(x)|------------|q.enq(x)|------------|p.deq(y)|------------>
B: ------------|q.enq(y)|------------|p.enq(y)|------------|q.deq(x)|->
```

**Exercise 7.2** In this exercise, we look into the Treiber Stack. Your task is to reason about its correctness using linearization and testing.

Blanka

*Mandatory*

1. Define linearization points for the `push` and `pop` methods in the Treiber Stack code provided in `app/src/main/java/exercises07/LockFreeStack.java`. Explain why those linearization points show that the implementation of the Treiber Stack is correct. For the explanation, follow a similar approach as we did in lecture 7 for the MS Queue (slides 60 and 61) or the explanation in the concurrency note (section Linearizability).

2. Write a JUnit 5 functional correctness test for the `push` method of the Treiber Stack. Consider a stack of integers. The test must assert that after $n$ threads push integers $x_1, x_2, \ldots, x_n$, respectively, the total sum of the elements in the stack equals $\sum_{i=1}^{n} x_i$. Write your test in the test skeleton file `app/src/test/java/exercises07/TestLockFreeStack.java`

3. Write a JUnit 5 functional correctness test for the `pop` method of the Treiber Stack. As before, consider a stack of integers. Given a stack with $n$ elements $x_1, x_2, \ldots, x_n$ already pushed, the test must assert the following: after $n$ threads pop one element $y_i$ each, the sum of popped elements equals the sum of elements originally in the stack $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$. Write your test in the test skeleton file `app/src/test/java/exercises07/TestLockFreeStack.java`

4. Do the tests in part 2. and 3. cover all linearization points in the Treiber Stack? Explain your answer. If you answered that not all linearization points were covered, then add additional concurrent functional tests to cover all linearization points.

Blanka

**Exercise 7.3** In this exercise, we revisit the progress notions for non-blocking computation that we discussed in lecture 6.

*Mandatory*

1. Consider the reader-writer locks exercise from week 6. There are four methods included in this type of locks: `writerTryLock`, `writerUnlock`, `readerTryLock` and `readerUnlock`. State, for each method, whether they are *wait-free*, *lock-free* or *obstruction-free* and explain your answers.

**Exercise 7.4** This exercise is a slightly modified version of exercise 3.7 in Herlihy page 71. This exercise concerns the following lock-free queue implementation:

```
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)); // E4
    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)); // D8
    return value;
  }
}
```

*Challenging*

1. In the code, we have marked two linearization points `E4` and `D8` for `enq` and `deq`, respectively. Show that, with these linearization points, the implementation is *not* linearizable (according to the standard specification of a sequential FIFO queue).