Group 1

# Assignment 2

## Week 3

**3.1.1)** *Implement a class BoundedBuffer as described above using only Java Semaphore for synchronization— i.e., Java Lock or intrinsic locks (synchronized) cannot be used.*

See BoundedBuffer.java

**3.1.2).** *Explain why your implementation of BoundedBuffer is thread-safe.*

A) In this class, we have five states: LinkedList<Object> buffer, int cnt, Semaphore empt, Semaphore full, Semaphore mutex.
B) All the states are declared as private and no method returns a reference to the state. So states will **not escape**.
C) All states are declared as final or volatile, so it is **safely published**.
D) All states are mutable, the Semaphore object itself is thread-safe. Semaphore mutex is used to ensure that access to both buffer and cnt is **mutually exclusive**.

**3.1.3)** *Is it possible to implement BoundedBuffer using Barriers? Explain your answer.*

No. Barriers are synchronization primitives used to wait until several threads reach some point in their computation. If we allow multiple threads to calculate simultaneously, we lose the mutual exclusion guarantee.

**3.1.4)** *One of the two constructors to Semaphore has an extra parameter named fair. Explain what it does, and explain if it matters in this example. If it does not matter in this example, find an example where it does matter.*

Java semaphores have a fair flag so that their entry queue prioritizes the longest waiting thread.

**3.2.1)** *Implement a thread-safe version of Person using Java intrinsic locks (synchronized). Hint: The Person class may include more attributes than those stated above; including static attributes*

See Person.java

Group 1

**3.2.2)**. *Explain why your implementation of the Person constructor is thread-safe, and why subsequent accesses to a created object will never refer to partially created objects.*

a) In this class, we have six states: long idCounter, boolean isFirstInstance, long id, String name, int zip, String address.
b) All the states are declared as private and no method returns a reference to the state. So states will **not escape**.
c) idCounter and isFirstInstance are class variable. They are initialized before instance initialization. id is defined as final, All other variables are initialized with default values, so it is **safely published**. This also means that there will be no reordering during construction, and the created object will never refer to partially created objects.
d) id is final (immutable), for all other variables, We use the synchronized keyword for both reading and writing to ensure **mutual exclusion** access.

**3.2.3)** *Implement a main thread that starting several threads that create and use instances of the Person class.*

See Person.java

**3.2.4)** *Assuming that you did not find any errors when running 3. Is your experiment in 3 sufficient to prove that your implementation is thread-safe?*

No, to make a sufficient experiment to prove thread-safety of the Person class, we would have to set up an experiment where we call both the read and write methods of the class on the same instance from multiple threads.

Group 1

# Assignment 2

## Week 4

**4.1.1)** *Implement a functional correctness test that finds concurrency errors in the add(Integer element) method in ConcurrentIntegerSetBuggy. Describe the interleaving that your test finds.*

T1(Check for 1), T2(Check for 1), T1(Add 1), T2(Add 1)

Because add is not atomic it can create an interleaving where two threads tries to add the same element at the same time, where they both see the element is not in the set and proceeded to add the element, creating a set with multiples of the same integer.

**4.1.2) .** *Implement a functional correctness test that finds concurrency errors in the remove(Integer element) method in ConcurrentIntegerSetBuggy. Describe the interleaving that your test finds.*

T1(check for j), T2(check for j), T1(Remove j), T1(decrement size), T2(Remove j), T2(decrement size)

Because remove is not atomic it can create an interleaving where multiple threads tries to remove the same integer, decrementing the size of the set twice while only removing 1 element, so size method no longer matches the actual size of the set.

**4.1.3)** *In the class ConcurrentIntegerSetSync, implement fixes to the errors you found in the previous exercises. Run the tests again to increase your confidence that your updates fixed the problems. In addition, explain why your solution fixes the problems discovered by your tests.*

By adding the synchronization keyword to the methods use the objects intrinsic lock, ensuring we can never have an interleaving where multiple threads can access the critical sections of the methods simultaneously.

**4.1.4)** *Run your tests on the ConcurrentIntegerSetLibrary. Discuss the results.*

When running the test with ConcurrentIntegerSetLibrary, it passes the test. This is due to the type ConcurrentSkipListSet class used in this case, which is designed to be thread-safe for concurrent access by multiple threads.

**4.1.5).** *Do a failure on your tests above prove that the tested collection is not thread-safe? Explain your answer.*

Yes, failure on the tests will mean the current structure of the set after the operations does not match the specification of a hashset.

**4.1.6)** *Does passing your tests above prove that the tested collection is thread-safe (when only using add() and remove())? Explain your answer.*

No, the test can still be passed while not being thread-safe if it happens to execute in a certain interleaving. Which is why we have to repeat the test many times to get the problematic interleavings. It is also missing a test that both adds and removes from the collection simultaneously from multiple threads.

**4.2.1)** *Let capacity denote the final field capacity in SemaphoreImp. Then, the property above does not hold for SemaphoreImp. Your task is to provide an interleaving showing a counterexample of the property, and explain why the interleaving violates the property?*

T1(release), T2(acquire), T3(acquire), T4(acquire), T5(acquire)

It violates the property of only allowing *c* number of threads in the critical section, as we can have an interleaving that starts with a release, decrementing the current state to -1, which then allows the SemaphoreImp to have c+1 threads in the critical section. This hold for any interleaving where there is a section with more calls to release than to acquire.

**4.2.2)** *Write a functional correctness test that can trigger the interleaving you describe in 1. Explain why your test triggers the interlaving.*

The test demonstrates this interleaving by by first calling the release method once, before initiallizing N number of threads, and then checking if the count of intialized threads is equal to N. If it is the property of the semaphore holds, if not it means that the initial call to release decremented state to -1 and allowed the semaphore to initiate N+1 threads which violates its property.