All the code can be found on: https://github.com/PatNei/PCPP-assignment-2

Group name: Spinning Jenny

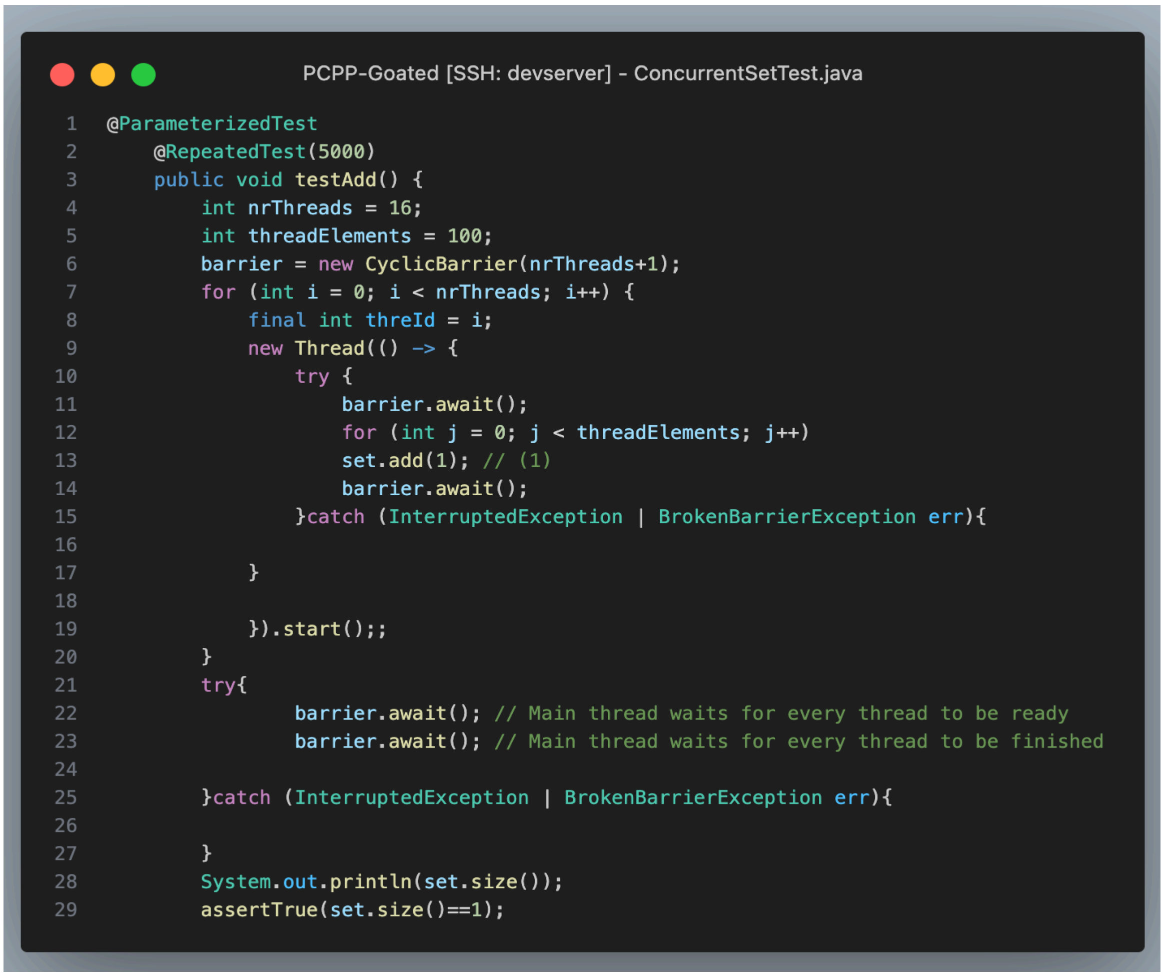
Group members:

Tobias Skovbæk Brund	tbru
Patrick Wittendorff Abarzua Neira	paab
Anders Arvesen	aarv

Exercise 4.1

Exercises 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.

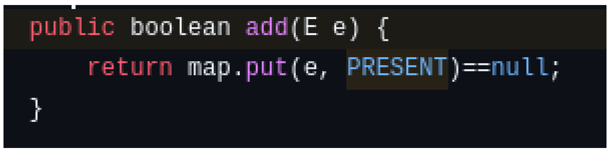


```java
@ParameterizedTest
    @RepeatedTest(5000)
    public void testAdd() {
        int nrThreads = 16;
        int threadElements = 100;
        barrier = new CyclicBarrier(nrThreads+1);
        for (int i = 0; i < nrThreads; i++) {
            final int threId = i;
            new Thread(() -> {
                try {
                    barrier.await();
                    for (int j = 0; j < threadElements; j++)
                    set.add(1); // (1)
                    barrier.await();
                }catch (InterruptedException | BrokenBarrierException err){

            }

            }).start();;
        }
        try{
                barrier.await(); // Main thread waits for every thread to be ready
                barrier.await(); // Main thread waits for every thread to be finished

        }catch (InterruptedException | BrokenBarrierException err){

        }
        System.out.println(set.size());
        assertTrue(set.size()==1);
```

We found that spawning N-threads all attempting to insert the same element to the `*ConcurrentIntegerSetBuggy*`doesn't lead to a size of 1 (as would be expected by the invariant of a set).

The interleaving that could lead to such an issue:
Since we don't have the actual implementation of the *HashSet*, but it makes use of a *HashMap.* An add call is really just a *map.put* which returns the old value compared to *null*. So:

- Let 1 be the *set.add* call in our code
- Let 2 be the *map.put* call from the *HashSet*.



```java
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

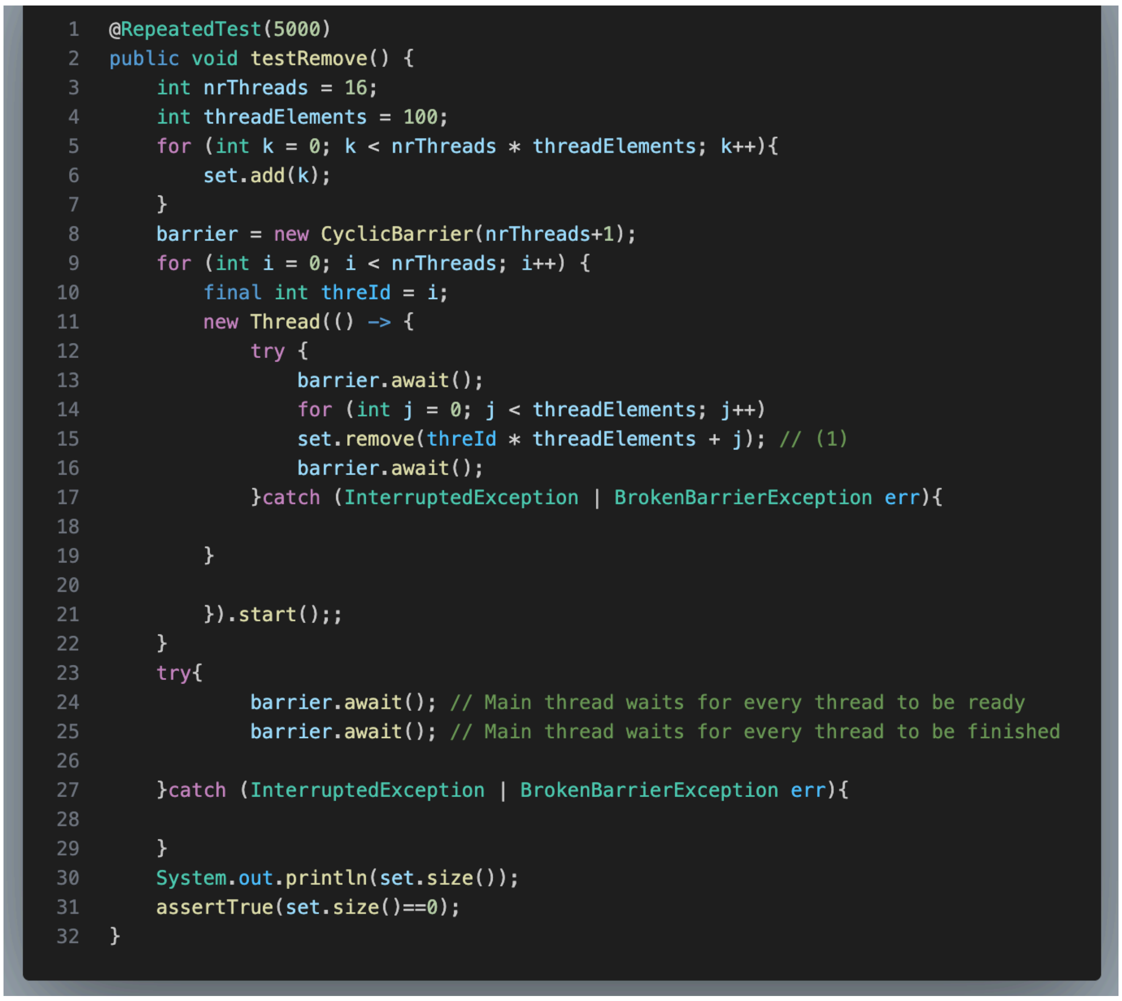
Note, that the *set.add* returns the value already there in the map, NULL if not there?
Imagine two threads **t1** and **t2**.

A problematic interleaving could then be:
t1(1), t2(1), t1(2), t2(2)

That is, when both threads read the underlying structure, they both put the SAME element to the same position. But both read a null because neither thread is done. Also, no happens-before, they could be writing to cached memory.

Exercises 4.1.2

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.



```java
@RepeatedTest(5000)
public void testRemove() {
    int nrThreads = 16;
    int threadElements = 100;
    for (int k = 0; k < nrThreads * threadElements; k++){
        set.add(k);
    }
    barrier = new CyclicBarrier(nrThreads+1);
    for (int i = 0; i < nrThreads; i++) {
        final int threId = i;
        new Thread(() -> {
            try {
                barrier.await();
                for (int j = 0; j < threadElements; j++)
                set.remove(threId * threadElements + j); // (1)
                barrier.await();
            }catch (InterruptedException | BrokenBarrierException err){

        }

        }).start();;
    }
    try{
            barrier.await(); // Main thread waits for every thread to be ready
            barrier.await(); // Main thread waits for every thread to be finished

    }catch (InterruptedException | BrokenBarrierException err){

    }
    System.out.println(set.size());
    assertTrue(set.size()==0);
}
```

- Above is the test, it first fills the set with a bunch of unique values, that are later to be removed by the testing threads.
- Each thread removes its own disjoint subset of elements/values. We'd expect it to go just fine and that by the end of the test, the set has a size of 0. (since all elements would have been removed)
- The test fails, not every time, but sometimes.

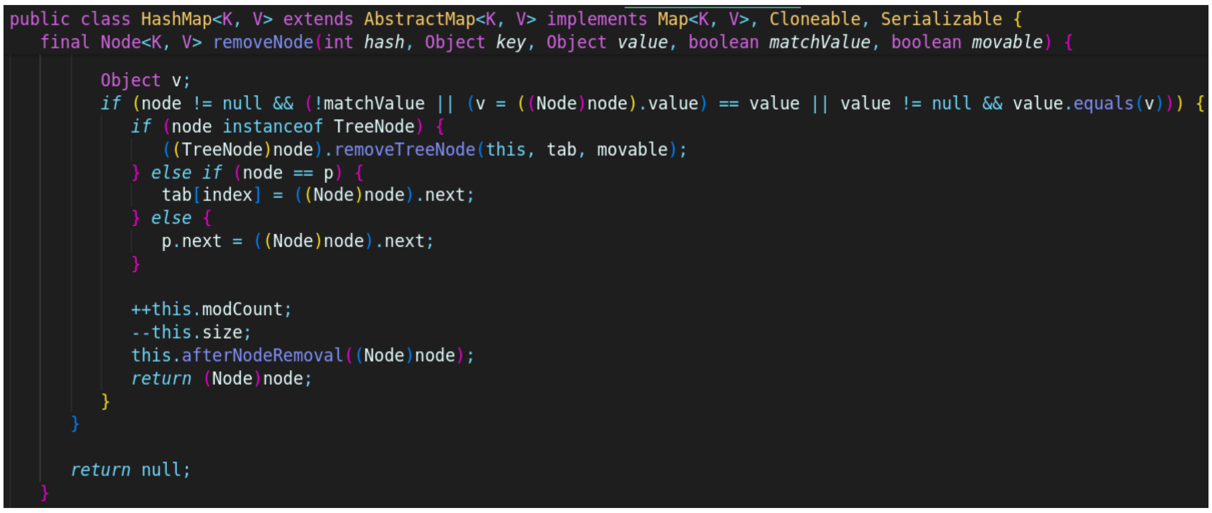
The interleaving that could lead to a non-zero size:

In our test we use 16 threads, we don't need that many for the interleaving. Imagine two threads *t1, t2*. And then the point 1) is the *set.remove* from the above screenshot, and 2) will be somewhere in the implementation of a HashMap (the underlying structure of hashset) that does the decrement of its size.

t1(1); t2(1); t2(2); t1(2)

Since there is no happens-before relation (no synchronization) between the threads, the decrement from *t2* is not necessarily visible to *t1,* the may see the same value, and both end up writing `*old-1*` back to the size field. Leaving the size 1 less when it should have been 2 less.

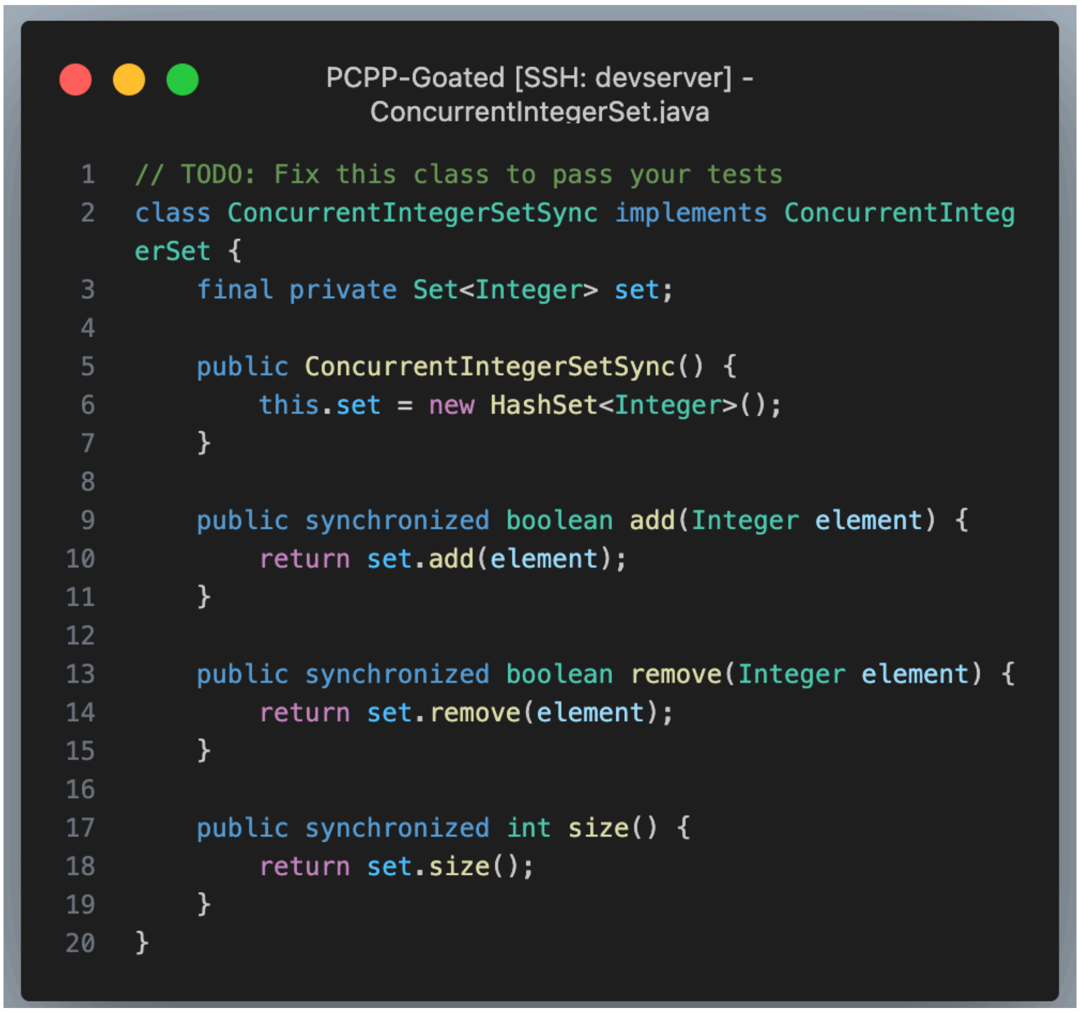
(The decrement is right there:)



```java
public class HashMap<K, V> extends AbstractMap<K, V> implements Map<K, V>, Cloneable, Serializable {
    final Node<K, V> removeNode(int hash, Object key, Object value, boolean matchValue, boolean movable) {

            Object v;
            if (node != null && (!matchValue || (v = ((Node)node).value) == value || value != null && value.equals(v))) {
                if (node instanceof TreeNode) {
                    ((TreeNode)node).removeTreeNode(this, tab, movable);
                } else if (node == p) {
                    tab[index] = ((Node)node).next;
                } else {
                    p.next = ((Node)node).next;
                }

                ++this.modCount;
                --this.size;
                this.afterNodeRemoval((Node)node);
                return (Node)node;
            }
        }

        return null;
    }
```

Exercises 4.1.3

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.


PCPP-Goated [SSH: devserver] - ConcurrentIntegerSet.java

```java
// TODO: Fix this class to pass your tests
class ConcurrentIntegerSetSync implements ConcurrentIntegerSet {
    final private Set<Integer> set;

    public ConcurrentIntegerSetSync() {
        this.set = new HashSet<Integer>();
    }

    public synchronized boolean add(Integer element) {
        return set.add(element);
    }

    public synchronized boolean remove(Integer element) {
        return set.remove(element);
    }

    public synchronized int size() {
        return set.size();
    }
}
```

We have made use of ***instance confinement***, by encapsulating access to the underlying *HashSet.*

- => The non-thread-safe class has been wrapped in a thread-safe class.
- => Every operation has become **atomic**.

Exercises 4.1.4

4. Run your tests on the `ConcurrentIntegerSetLibrary`. Discuss the results.

We live in an ideal world, we found no errors. We don't know if the *ConcurrentSkipListSet* really is thread-safe, but we didn't find any race conditions using our tests. This sheds some light on the difficulty of testing concurrent code as the give interleavings used for a test is non deterministic.

Exercises 4.1.5

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

Yes, credit to E.W. Dijkstra:
"Program testing can be used to show the presence of bugs, but never to show their absence!"
If one of the above tests fails, then we have "shown" that there exists an interleaving that causes race conditions.
We have not shown the counterexample itself (although JavaPathFinder could do it for us).

Exercises 4.1.6

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

No
E.W. Dijkstra:
"Program testing can be used to show the presence of bugs, but never to show their absence!"
If one of the above tests pass, then we have "shown" that there exists an interleaving that doesn't cause race conditions.
We have not shown the counterexample itself (although JavaPathFinder could do it for us).

Exercise 4.2

Exercises 4.2.1

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.

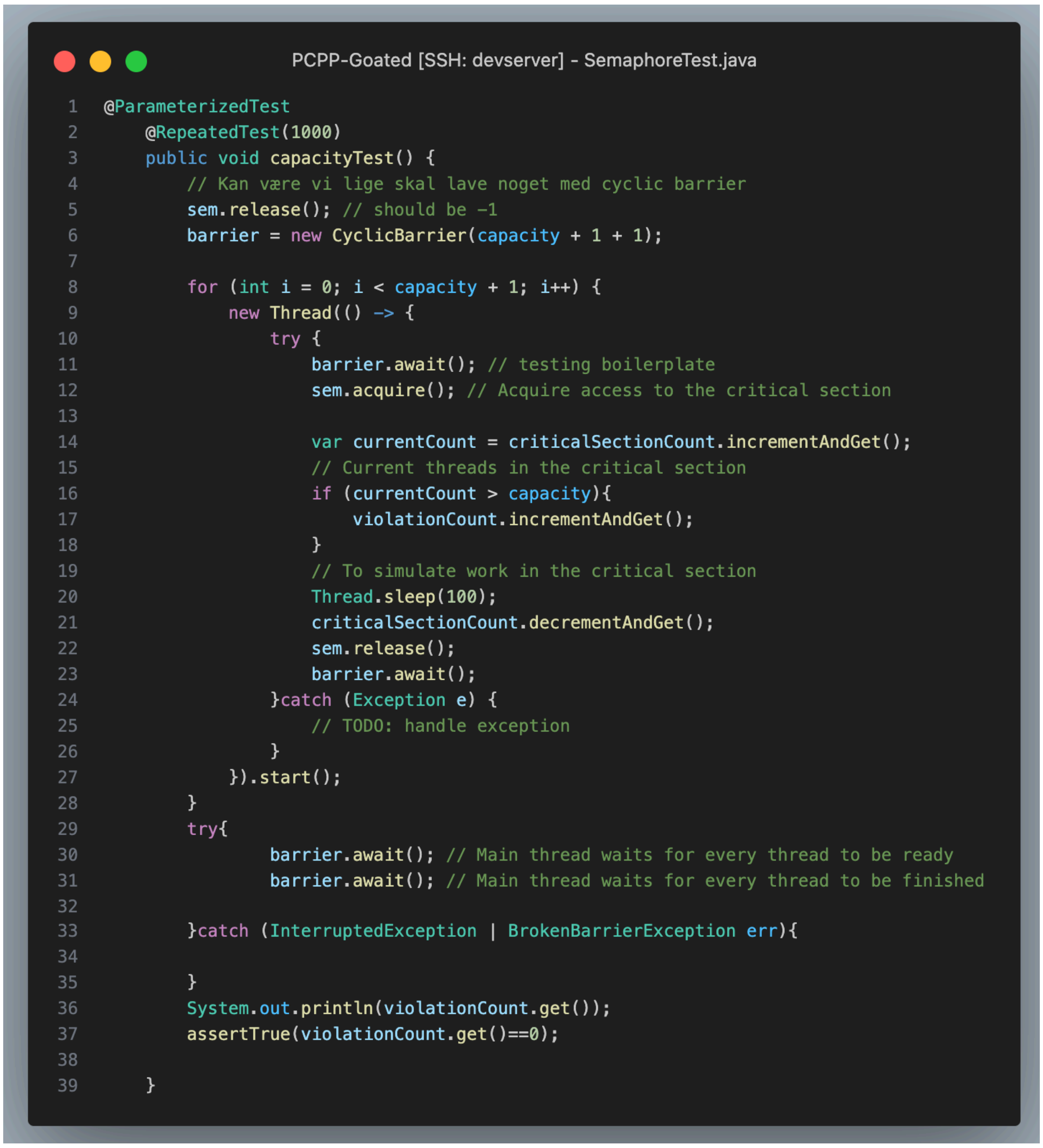
We have threads (mt), t1, t2, t3, t4.
We will call *release,* and *acquire*.
Also, imagine that **capacity=2**

mt(*release*); t1(*acquire*); t2(*acquire*); t3(*acquire*); t4(*acquire*) ← blocks;

There would be *3* threads in the critical section, but capacity was only 2 (3 > 2).

Exercises 4.2.2

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



PCPP-Goated [SSH: devserver] - SemaphoreTest.java

```java
@ParameterizedTest
    @RepeatedTest(1000)
    public void capacityTest() {
        // Kan være vi lige skal lave noget med cyclic barrier
        sem.release(); // should be -1
        barrier = new CyclicBarrier(capacity + 1 + 1);

        for (int i = 0; i < capacity + 1; i++) {
            new Thread(() -> {
                try {
                    barrier.await(); // testing boilerplate
                    sem.acquire(); // Acquire access to the critical section

                    var currentCount = criticalSectionCount.incrementAndGet();
                    // Current threads in the critical section
                    if (currentCount > capacity){
                        violationCount.incrementAndGet();
                    }
                    // To simulate work in the critical section
                    Thread.sleep(100);
                    criticalSectionCount.decrementAndGet();
                    sem.release();
                    barrier.await();
                }catch (Exception e) {
                    // TODO: handle exception
                }
            }).start();
        }
        try{
                barrier.await(); // Main thread waits for every thread to be ready
                barrier.await(); // Main thread waits for every thread to be finished

        }catch (InterruptedException | BrokenBarrierException err){

        }
        System.out.println(violationCount.get());
        assertTrue(violationCount.get()==0);

    }
```

Why does it trigger the interleaving?
Our test triggers the interleaving, because we initially call release() before starting any threads, which will decrement the state-field of the semaphore to -1, meaning that we can exceed the capacity, by calling acquire() the following amount of times: capacity + 1.