Practical Concurrent and Parallel Programming IV

Testing & Verification

Raúl Pardo

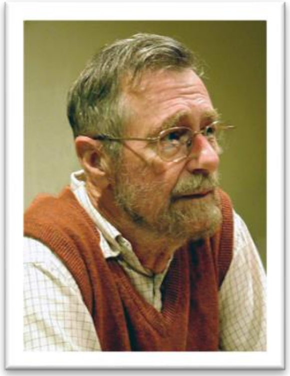
Oral feedback week 1

- Apologies for the situation with oral feedback
- Please note that this was an exceptional and unforeseen situation

Remember

*“Program **testing** can be used to **show the presence of bugs**, but **never to show their absence**!”*

Edsger W. Dijkstra



Agenda

- Intro to concurrency properties
- Testing
  - Intro to JUnit 5
  - Counter
  - Bounded Buffer
  - Deadlocks
- Formal Verification
  - Java Path Finder

Concurrency properties

- Traditionally, properties of concurrent programs are split into:
  - Safety – *“Something bad never happens”*

    Ex. 1: Two intersection traffic lights are never green at the same time
    Ex. 2: The field size of a collection is never less than 0

  - Liveness – *“Something good will eventually happen”*

    Ex. 1: The traffic light will eventually switch to red
    Ex. 2: It should always be possible to eventually add elements to the collection
    Ex. 3: If an actor sends message to another actor, the latter will eventually receive the message

Interleavings


```
// shared variable
int counter = 0;

// two threads
for(int i=0; i<2; i++){
  new Thread(() -> {
    while(true) {
      int temp = counter;     (1)
      temp = counter + 1;     (2)
      counter = temp;         (3)
    }
  }).start();
}
```

What are possible interleavings for this program? (assuming that (1),(2) and (3) are atomic operations)

Interleavings


```
// shared variable
int counter = 0;

// two threads
for(int i=0; i<2; i++){
  new Thread(() -> {
    while(true) {
      int temp = counter;    (1)
      temp = counter + 1;    (2)
      counter = temp;        (3)
    }
  }).start();
}
```

Assuming that (1), (2) and (3) are atomic.

Some interleavings are

1. (1), (2), (3), (1), (2), (3),...
2. (1), (2), (3), (1), (2), (3),...
3. (1), (2), (3), (1), (2), (3),...
4. (1), (2), (3), (1), (2), (3),...

But we also have

1. (1), (1), (2), (2), (3), (3),...
2. (1), (2), (1), (2), (3), (3),...

These produce race conditions

Testing concurrent programs

- Testing concurrent programs is about writing tests to find undesired interleavings (if any)
    - These are commonly known as *counterexamples*
    - They show an interleaving that violates a property

But we also have

1. (1), (1), (2), (2), (3), (3),…
2. (1), (2), (1), (2), (3), (3),…

These produce race conditions

- Since concurrent execution is non-deterministic, it is not guaranteed that tests will trigger undesired interleavings

- Today we will see strategies to design tests to find interleavings that violate a property

Structure of counterexamples

- The type of counterexample we are looking for, depends on the type of property
  - Safety
  - Liveness

Counterexamples in safety properties

- Safety property
  - A counterexample is a ***finite*** interleaving where the property does not hold

- Ex. 1: Two intersection traffic lights are never green at the same time
  - *Counterexample*: a finite interleaving that result in having two green lights at the same time

- Ex. 2: The field size of a collection is never less than 0
  - *Counterexample*:

Can you give a counterexample for this property?

Counterexamples in safety properties

- Safety property
  - A counterexample is a <u>*finite*</u> interleaving where the property does not hold

- <u>Ex. 1</u>: Two intersecting traffic lights are never green at the same time
  - *Counterexample*: a finite interleaving that result in having two green lights at the same time

- <u>Ex. 2</u>: The field size of a collection is never less than 0
  - *Counterexample*: a finite interleaving that results in size less than 0, e.g., -1

Structure of counterexamples

- Liveness property
  - A counterexample is an ***infinite*** interleaving where the property never holds

- Ex. 1: The traffic light will eventually switch to red
  - *Counterexample:* an infinite interleaving when lights are never red

- Ex. 2: It should always be possible to eventually add elements to the collection
  - *Counterexample:* an infinite interleaving when a thread can never add an element to the collection

Can you give a counterexample for this property?

Structure of counterexamples

- Liveness property
  - A counterexample is an ***infinite*** interleaving where the property never holds

- Ex. 1: The traffic light will eventually switch to red
  - *Counterexample:* an infinite interleaving when lights are never red. E.g., an interleaving where lights are always green.

- Ex. 2: It should always be possible to eventually add elements to the collection
  - *Counterexample:* an infinite interleaving when a thread can never add an element to the collection.
    E.g., an interleaving where other threads are always chosen to add elements, and one thread is never scheduled.

Structure of counterexamples

- The type of counterexample we are looking for, depends on the type of property
  - **Safety**
  - Liveness

Today we focus only on safety properties

Testing Concurrent Programs (Counter)

Types of concurrency tests

- **Functional Correctness tests**
  - These tests focus on testing that program behaves (functions) correctly when executed concurrently
  - For instance, data structures
  - *This lecture focuses on this type of tests*

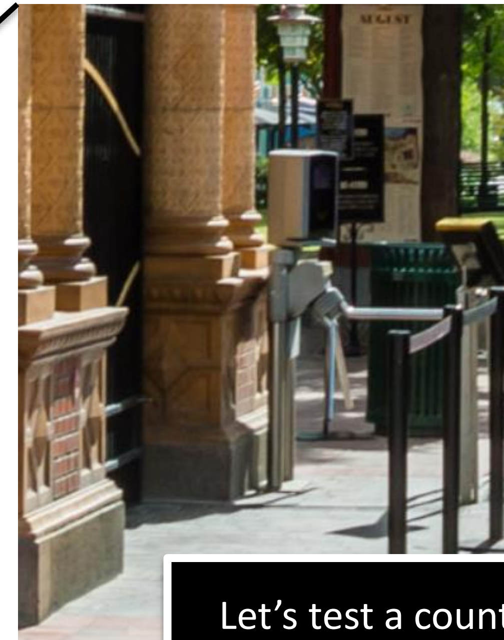
- Performance tests (next week with Jørgen)
  - These tests focus on measuring the execution performance of concurrent programs
  - We will see saw in week 5 a more accurate (and statistically stronger) method to measure performance than the book

JUnit 5

- JUnit is a popular unit test framework for Java programs
- It makes it easy to implement and run tests
- Some useful features are:
  - Execute initialization tasks
  - Running tests repeatedly
  - Define and automatically execute sets of input parameters for a test
  - Compatibility with build tools, such as Gradle
  - ...

Threads in Java – Example
Tivoli entrance turnstile

Another déjà vu from lecture 1



Let’s test a counter class that counts the number of visitors that cross the turnstile

Sequential tests in JUnit 5

Class to test

```
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
        count = 0;
    }

    public void inc() {
        count++;
    }

    public int get() {
        return count;
    }
}
```

Test Class

```
// several imports

public class CounterTest {

    private Counter count;

    @BeforeEach
    public void initialize() {
      count = new CounterDR();
    }

    @RepeatedTest(10)
    @DisplayName("Counter Sequential")
    public void testingCounterSequential()
    {
      int localSum = 0;
      for (int i = 0; i < 10_000; i++) {
        count.inc();
        localSum++;
      }
      assertTrue(count.get()==localSum);
    }
    …
    // other tests
}
```

Sequential tests in JUnit 5

Class to test

```
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
        count = 0;
    }

    public void inc() {
        count++;
    }

    public int get() {
        return count;
    }
}
```

Counter variable that will be used in the tests

Test Class

```
// several imports

public class CounterTest {

    private Counter count;

    @BeforeEach
    public void initialize() {
      count = new CounterDR();
    }

    @RepeatedTest(10)
    @DisplayName("Counter Sequential")
    public void testingCounterSequential()
    {
      int localSum = 0;
      for (int i = 0; i < 10_000; i++) {
        count.inc();
        localSum++;
      }
      assertTrue(count.get()==localSum);
    }
    …
    // other tests
}
```

Sequential tests in JUnit 5

Class to test

```
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
        count = 0;
    }

    public void inc() {
        count++;
    }

    public int get() {
        return count;
    }
}
```

Counter variable that will be used in the tests

This method is executed before each test. It is useful to initialize the objects to test

Test Class

```
// several imports

public class CounterTest {

    private Counter count;

    @BeforeEach
    public void initialize() {
      count = new CounterDR();
    }

    @RepeatedTest(10)
    @DisplayName("Counter Sequential")
    public void testingCounterSequential()
    {
      int localSum = 0;
      for (int i = 0; i < 10_000; i++) {
        count.inc();
        localSum++;
      }
      assertTrue(count.get()==localSum);
    }
    …
    // other tests
}
```

Sequential tests in JUnit 5

Class to test

```
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
        count = 0;
    }

    public void inc() {
        count++;
    }

    pu

    }
}
```

Counter variable that will be used in the tests

This method is executed before each test. It is useful to initialize the objects to test

First, we define the type of test. One might use @Test (regular test), @RepeatedTest(X) the test is executed X times, or @ParameterizedTest with an input generator (see next slides)

Test Class

```
// several imports

public class CounterTest {

    private Counter count;

    @BeforeEach
    public void initialize() {
      count = new CounterDR();
    }

    @RepeatedTest(10)
    @DisplayName("Counter Sequential")
    public void testingCounterSequential()
    {
      int localSum = 0;
      for (int i = 0; i < 10_000; i++) {
        count.inc();
        localSum++;
      }
      assertTrue(count.get()==localSum);
    }
    …
    // other tests
}
```

Sequential tests in JUnit 5

Class to test

```
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
        count = 0;
    }

    public void inc() {
        count++;
    }

    pu

    }
}
```

Counter variable that will be used in the tests

This method is executed before each test. It is useful to initialize the objects to test

First, we define the type of test. One might use @Test (regular test), @RepeatedTest(X) the test is executed X times, or @ParameterizedTest with an input generator (see next slides)

Test Class

```
// several imports

public class CounterTest {

    private Counter count;

    @BeforeEach
    public void initialize() {
      count = new CounterDR();
    }

    @RepeatedTest(10)
    @DisplayName("Counter Sequential")
    public void testingCounterSequential()
    {
      int localSum = 0;
      for (int i = 0; i < 10_000; i++) {
        count.inc();
        localSum++;
      }
      assertTrue(count.get()==localSum);
    }
    …
    // other tests
}
```

Some text to display when printing the result of the test

Sequential tests in JUnit 5

Class to test

```
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
        count = 0;
    }

    public void inc() {
        count++;
    }

    pu

    }
}
```

Counter variable that will be used in the tests

This method is executed before each test. It is useful to initialize the objects to test

First, we define the type of test. One might use @Test (regular test), @RepeatedTest(X) the test is executed X times, or @ParameterizedTest with an input generator (see next slides)

Body of the test. In this case we execute inc() 10000 times

Test Class

```
// several imports

public class CounterTest {

    private Counter count;

    @BeforeEach
    public void initialize() {
      count = new CounterDR();
    }

    @RepeatedTest(10)
    @DisplayName("Counter Sequential")
    public void testingCounterSequential()
    {
      int localSum = 0;
      for (int i = 0; i < 10_000; i++) {
        count.inc();
        localSum++;
      }
      assertTrue(count.get()==localSum);
    }
    …
    // other tests
}
```

Some text to display when printing the result of the test

Sequential tests in JUnit 5

Class to test

```java
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
        count = 0;
    }

    public void inc() {
        count++;
    }

    public int get() {

    }
}
```

Test

```java
// several imports

public class CounterTest {

    private Counter count;

    @BeforeEach
    public void initialize() {
      count = new CounterDR();
    }

    @RepeatedTest(10)
    @DisplayName("Counter Sequential")
    public void testingCounterSequential()
    {
      int localSum = 0;
      for (int i = 0; i < 10_000; i++) {
        count.inc();
        localSum++;
      }
      assertTrue(count.get()==localSum);
    }
    …
    // other tests
}
```

The test finishes with some assertions. Here we check that the final value of count equals our local sum.

You may also add assertions during the execution of the test.

Sequential tests in JUnit 5

Class to test

```
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
        count = 0;
    }

    public void inc() {
        count++;
    }
```

To run tests in gradle we use:

```
$ gradle cleanTest test --tests <package>.<test_class>
```

In this example,

```
$ gradle cleanTest test --tests testingconcurrency.CounterTest
```

(`cleanTest` ensures a fresh environment for running the test, it is not always necessary)

Test

```
// several imports

public class CounterTest {

    private Counter count;

    @BeforeEach
    public void initialize() {
      count = new CounterDR();
    }

    @RepeatedTest(10)
    @DisplayName("Counter Sequential")
    public void testingCounterSequential()
    {
      int localSum = 0;
      for (int i = 0; i < 10_000; i++) {
        count.inc();
        localSum++;
      }
      assertTrue(count.get()==localSum);
    }
    …
    // other tests
}
```

Concurrent Correctness Test – Counter

- Now we extend the test to multiple threads (or turnstiles)

Concurrent Correctness Test – Counter

- Some strategies to take into account when developing a test:

1. Precisely define the property you want to test

2. If you are going to test multiple implementations, it is useful to define an *interface* for the class you are testing

3. Concurrent tests require a setup for starting and running multiple threads
   - Maximize contention to avoid a sequential execution of the threads
   - You may need to define thread classes

4. Run the tests multiple times and with different setups to try to maximize the number of interleavings tested

Concurrent Correctness Test – Counter

- Precisely define the property you want to test
- *“after N threads execute inc() X times, the value of the counter must be equal to N*X”*

```
Class CounterTest {

 Counter count;
 …
 public void testingCounterParallel(int nrThreads, int N) {
     // body of the test
     assert(N*nrThreads == count.get());
 }
 …
}
```

Concurrent Correctness Test – Counter

- Precisely define the property you want to test
- *“after N threads execute inc() X times, the value of the counter must be equal to N*X”*

```
Class CounterTest {

 Counter count;
 …
 public void testingCounterParallel(int nrThreads, int N) {
     // body of the test
     assert(N*nrThreads == count.get());
 }
 …
}
```

Is this a safety or liveness property?

Concurrent Correctness Test – Counter

- If you are going to test multiple implementations, it is useful to define an *interface* for the class you are testing

```
public interface Counter {
    public void inc();
    public int get();
}
```

```
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
      count = 0;
    }

    public void inc() {
      count++;
    }

    public int get() {
      return count;
    }
}
```

```
class CounterSync implements Counter {

    private int count;

    public CounterSync() {
      count = 0;
    }

    public synchronized void inc() {
      count++;
    }

    public int get() {
      return count;
    }
}
```

```
class CounterAto implements Counter {

    private AtomicInteger count;

    public CounterAto() {
      count = new AtomicInteger(0);
    }

    public void inc() {
      count.incrementAndGet();
    }

    public int get() {
      return count.get();
    }}
```

Concurrent Correctness Test – Counter

- If you are going to test multiple implementations, it is useful to define an *interface* for the class you are testing

```
public interface Counter {
    public void inc();
    public int get();
}
```

A thread-safe interger class, with methods to increase and decrete the intenger

```
class CounterDR implements Counter {

    private int count;

    public CounterDR() {
      count = 0;
    }

    public void inc() {
      count++;
    }

    public int get() {
      return count;
    }
}
```

```
class CounterSync implements Counter {

    private int count;

    public CounterSync() {
      count = 0;
    }

    public synchronized void inc() {
      count++;
    }

    public int get() {
      return count;
    }
}
```

```
class CounterAto   ements Counter {

    private AtomicInteger count;

    public CounterAto() {
      count = new AtomicInteger(0);
    }

    public void inc() {
      count.incrementAndGet();
    }

    public int get() {
      return count.get();
    }}
```

Concurrent Correctness Test – Counter

- Maximize contention (e.g., using a cyclic barrier) to avoid a sequential execution of the threads
  - Use a cyclic barrier to reduce the likelihood that threads are executed in a sequence

```
class TestCounter {

  // Shared variable for the tests
  CyclicBarrier barrier;

  …

 public void testingCounterParallel(int nrThreads, int N) {
     …

     // init barrier
     barrier = new CyclicBarrier(nrThreads + 1);

     for (int i = 0; i < nrThreads; i++) {
       new Thread(() -> {
        barrier.await(); // wait until all threads are ready
        // thread execution
        barrier.await(); // wait until all threads are finished
      }).start();
     }

     try {
         barrier.await();
         barrier.await();
     } catch (InterruptedException | BrokenBarrierException e) {
         e.printStackTrace();
     }
     …
}
```

Concurrent Correctness Test – Counter

- Maximize contention (e.g., using a cyclic barrier) to avoid a sequential execution of the threads
  - Use a cyclic barrier to reduce the likelihood that threads are executed in a sequence

```
class TestCounter {

  // Shared variable for the tes
  CyclicBarrier barrier;

  …

 public void testingCounterParal
     …

     // init barrier
     barrier = new CyclicBarrier(nrThreads + 1);

     for (int i = 0; i < nrThreads; i++) {
       new Thread(() -> {
        barrier.await(); // wait until all threads are ready
        // thread execution
        barrier.await(); // wait until all threads are finished
      }).start();
     }

     try {
          barrier.await();
          barrier.await();
     } catch (InterruptedException | BrokenBarrierException e) {
          e.printStackTrace();
     }
     …
}
```

Why do we need this +1?

Concurrent Correctness Test – Counter

- You may need to define thread classes

```
class TestCounter {

 Counter count;
 …
 public class Turnstile extends Thread {

     private final int N;

     public Turnstile(int N) { this.N = N; }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < N; i++) {
           count.inc();
       }
       barrier.await();
         } catch (InterruptedException | BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
   …
}
```

Note that the thread includes the barrier.await()s

Concurrent Correctness Test – Counter

- You may need to define thread classes

```
class TestCounter {

 Counter count;
 …
 public class Turnstile extends Thread {

     private final int N;

     public Turnstile(int N) { this.N = N; }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < N; i++) {
            count.inc();
       }
       barrier.await();
         } catch (InterruptedException | BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
   …
}
```

Note that the thread includes the barrier.await()s

```
class TestCounter {

  // Shared variable for the tests
  CyclicBarrier barrier;

  …

 public void testingCounterParallel(int nrThreads,
                                    int N) {
     …

     // init barrier
     barrier = new CyclicBarrier(nrThreads + 1);

     for (int i = 0; i < nrThreads; i++) {
       new Turnstile(N).start();
     }

     try {
         barrier.awai
         barrier.awai
     } catch (Interru
     BrokenBarrierException e) {
         e.printStackTrace();
     }
     …
}
```

Now we can simply start the thread in the test

Concurrent Correctness Test – Counter

- You may need to define thread classes

```
class TestCounter {

 Counter count;
 …
 public class Turnstile extends Thread {

     private final int N;

     public Turnstile(int N) { this.N = N; }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < N; i++) {
           count.inc();
       }
       barrier.await();
         } catch (InterruptedException | BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
   …
}
```

Note that the thread includes the barrier.await()s

```
class TestCounter {

  // Shared variable for the tests
  CyclicBarrier barrier;
   private final static ExecutorService pool
     = Executors.newCachedThreadPool();
  …
 public void testingCounterParallel(int nrThreads,
                                    int N) {
     …

     // init barrier
     barrier = new CyclicBarrier(nrThreads + 1);

     for (int i = 0; i < nrThreads; i++) {
       pool.execute(new Turnstile(N));
     }

     try {
         barrier.awai
         barrier.awai
     } catch (Interru
     BrokenBarrierExce
         e.printStack
     }
     …
}
```

Alternatively, we can use a thread pool as in Goetz
We will cover ThreadPools in two weeks

Concurrent Correctness Test – Counter

- Optionally (though encouraged), one may generate input parameters using JUnit (@ParameterizedTest)
  - Note that the test method takes as input two integer parameters
  - Using @MethodSoucer we can specify a method that provides a collection of parameters (known as arguments)

```
class TestCounter {
 …
 @ParameterizedTest
 @MethodSource("argsGeneration")
 public void testingCounterParallel(int nrThreads,
                                    int N) {
     //body of the test
 }
 …
}
```

```
private static List<Arguments> argsGeneration() {

    // Max number of increments
    final int I = 50_000;
    final int iInit = 10_000;
    final int iIncrement = 10_000;

    // Max exponent number of threads (2^J)
    final int J = 6;
    final int jInit = 1;
    final int jIncrement = 1;

    // List to add each parameters entry
    List<Arguments> list = new
    ArrayList<Arguments>();

    // Loop to generate each parameter entry
    // (2^j, i) for i \in {10_000,20_000,...,J}
    //              and j \in {1,..,I}
    for (int i = iInit; i <= I; i += iIncrement) {
        for (int j = jInit; j < J; j += jIncrement) {
      list.add(Arguments.of((int) Math.pow(2,j),i));
        }
    }

    // Return the list
    return list;
 }
```

Concurrent Correctness Test – Counter

- Optionally (though encouraged), one may generate input parameters using JUnit (@ParameterizedTest)
  - Note that the test method takes as input two integer parameters
  - Using @MethodSoucer we can specify a method that provides a collection of parameters (known as arguments)

```
class TestCounter {
 …
 @ParameterizedTest
 @MethodSource("argsGeneration")
 public void testingCounterParallel(int nrThreads,
                                    int N) {
     //body of the test
 }
 …
}
```

```
private static List<Arguments> argsGeneration() {

    // Max number of increments
    final int I = 50_000;
    final int iInit = 10_000;
    final int iIncrement = 10_000;

    // Max exponent number of threads (2^J)
    final int J = 6;
    final int jInit = 1;
    final int jIncrement = 1;

    // List to add each parameters entry
    List<Arguments> list = new
    ArrayList<Arguments>();

    // Loop to generate each parameter entry
    // (2^j, i) for i \in {10_000,20_000,...,J}
    //          and j \in {1,..,I}
    for (int i = iInit; i <= I; i += iIncrement) {
        for (int j = jInit; j < J; j += jIncrement) {
      list.add(Arguments.of((int) Math.pow(2,j),i));
        }
    }

    // Return the list
    return list;
 }
```

Arguments is a JUnit class that can be seen as a collection of objects of different type

Concurrent Correctness Test – Counter

- Let’s look at all together in code-lecture directory
- Note that Gradle requires test classes to be placed in the folder app/src/test/java/<package>/
- We look at three different implementations
  - CounterDR
  - CounterSync
  - CounterCAS
- JUnit produces a nice HTML report in build/reports/tests/test/classes/<package>.<class>.html
  - It includes outputs and running times

Concurrent Correctness Test – Counter

- Remember that some interleavings are difficult to trigger
- Let's look at the test `testingCounterParallelConstant()`
  - It is hard to find the interleavings that violate our property
  - Executing the test multiple times increases your chances of triggering the interleavings you are looking for
  – Remember @RepeatedTest()

Testing a Bounded Buffer

Concurrent Correctness Test – BoundedBuffer

- Now we turn our attention to a Bounded Buffer

Consumers

Producers

Bounded buffer

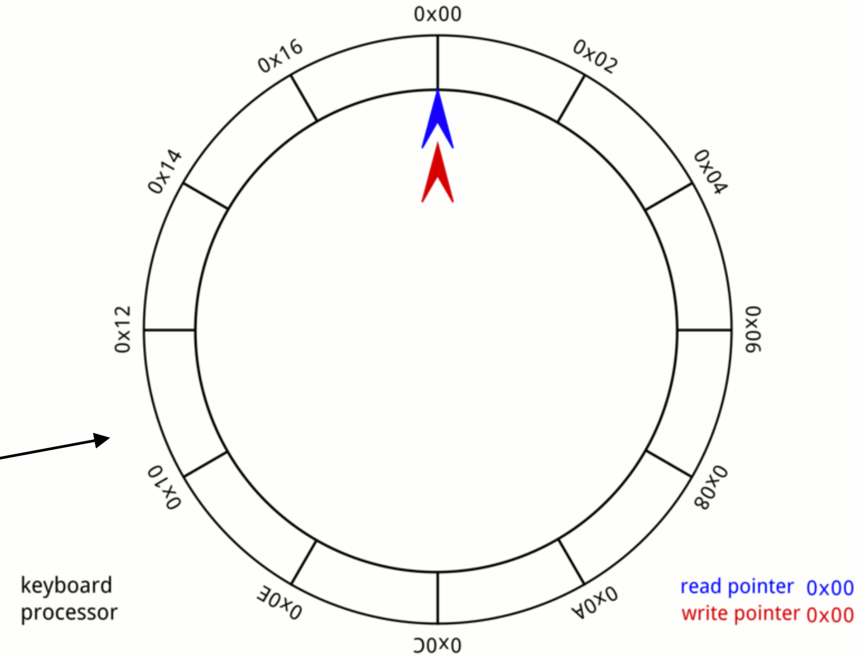
Concurrent Correctness Test – Bounded Buffer

- We study a functional correctness property of a bounded buffer that may be accessed by producers and consumers concurrently

- Producers may put elements in the buffer as long as there is space. Otherwise they must wait.

- Consumers can take elements from the buffer as long as it is not empty. Otherwise they must wait.

- The buffer is implemented as a *circular buffer*



- Synchronization is implemented using semaphores

Concurrent Correctness Test – Bounded Buffer


```
@ThreadSafe
public class BoundedBuffer<E> {
    private final Semaphore availableItems, availableSpaces;
    @GuardedBy("this") private final E[] items;
    @GuardedBy("this") private int putPosition = 0, takePosition = 0;

    public BoundedBuffer(int capacity) {
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }
    public boolean isEmpty() {
        return availableItems.availablePermits() == 0;
    }
    public boolean isFull() {
        return availableSpaces.availablePermits() == 0;
    }

    public void put(E x) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(x);
        availableItems.release();
    }
    public E take() throws InterruptedException {
        availableItems.acquire();
        E item = doExtract();
        availableSpaces.release();
        return item;
    }
```

- Here is the implementation in Goetz p. 249

```
    private synchronized void doInsert(E x) {
        int i = putPosition;
        items[i] = x;
        putPosition = (++i == items.length)? 0 : i;
    }
    private synchronized E doExtract() {
        int i = takePosition;
        E x = items[i];
        items[i] = null;
        takePosition = (++i == items.length)? 0 : i;
        return x;
    }
}
```

Concurrent Correctness Test – Bounded Buffer


```java
@ThreadSafe
public class BoundedBuffer<E> {
    private final Semaphore availableItems, availableSpaces;
    @GuardedBy("this") private final E[] items;
    @GuardedBy("this") private int putPosition = 0, takePosition = 0;

    public BoundedBuffer(int capacity) {
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }
    public boolean isEmpty() {
        return availableItems.availablePermits() == 0;
    }
    public boolean isFull() {
        return availableSpaces.availablePermits() == 0;
    }

    public void put(E x) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(x);
        availableItems.release();
    }
    public E take() throws InterruptedException {
        availableItems.acquire();
        E item = doExtract();
        availableSpaces.release();
        return item;
    }
```

- Here is the implementation in Goetz p. 249
- It uses two semaphores to block threads when the buffer is full or empty

```java
    private synchronized void doInsert(E x) {
        int i = putPosition;
        items[i] = x;
        putPosition = (++i == items.length)? 0 : i;
    }
    private synchronized E doExtract() {
        int i = takePosition;
        E x = items[i];
        items[i] = null;
        takePosition = (++i == items.length)? 0 : i;
        return x;
    }
}
```

Concurrent Correctness Test – Bounded Buffer


```
@ThreadSafe
public class BoundedBuffer<E> {
    private final Semaphore availableItems, availableSpaces;
    @GuardedBy("this") private final E[] items;
    @GuardedBy("this") private int putPosition = 0, takePosition = 0;

    public BoundedBuffer(int capacity) {
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }
    public boolean isEmpty() {
        return availableItems.availablePermits() == 0;
    }
    public boolean isFull() {
        return availableSpaces.availablePermits() == 0;
    }

    public void put(E x) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(x);
        availableItems.release();
    }
    public E take() throws InterruptedException {
        availableItems.acquire();
        E item = doExtract();
        availableSpaces.release();
        return item;
    }
```

- Here is the implementation in Goetz p. 249
- It uses two semaphores to block threads when the buffer is full or empty
- It uses intrinsic locks to ensure mutual exclusion when accessing the buffer

```
    private synchronized void doInsert(E x) {
        int i = putPosition;
        items[i] = x;
        putPosition = (++i == items.length)? 0 : i;
    }
    private synchronized E doExtract() {
        int i = takePosition;
        E x = items[i];
        items[i] = null;
        takePosition = (++i == items.length)? 0 : i;
        return x;
    }
}
```

Concurrent Correctness Test – Bounded Buffer

1. Property to check

- *"after several producers put integers $x_1, \ldots, x_N$ to the buffer and several consumers take integers $y_1, \ldots, y_N$ from the buffer, it must hold that $\sum_{i=1}^{N} x_i = \sum_{i=1}^{N} y_i$"*
- More informally: *"If several threads put and take the same number of elements, the sum of the put elements and the sum of the taken elements must be equal"*
- A producer may add more than one integer in the buffer and a consumer may take more than one integer
  - *The only constraint is that the combined number of puts and takes is the same for all producers and consumers*

Concurrent Correctness Test – Bounded Buffer

1. Property to check

- *“after several producers put integers $x_1, \dots, x_N$ to the buffer and several consumers take integers $y_1, \dots, y_N$ from the buffer, it must hold that $\sum_{i=1}^{N} x_i = \sum_{i=1}^{N} y_i$”*
- More informally: *“If several threads put and take the same number of elements, the sum of the put elements and the sum of the taken elements must be equal”*

er and a consumer

ts and takes is the

What should go wrong in the buffer for this property to be violated?

Or, assume a wrongly implemented buffer, can you give an interleaving that violates the property?

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (producer)

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

 class Producer extends Thread {
     int nrTrials;
     int localSum;

     public Producer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           Random r  = new Random();
           int toPut = r.nextInt();
           buffer.put(toPut);
           localSum += toPut;
       }
       putSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (producer)

We have use two global AtomicIntegers to keep track of the global sum of put/remove

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

 class Producer extends Thread {
     int nrTrials;
     int localSum;

     public Producer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           Random r  = new Random();
           int toPut = r.nextInt();
           buffer.put(toPut);
           localSum += toPut;
       }
       putSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (producer)

We have use two global AtomicIntegers to keep track of the global sum of put/remove

The producer is initialized with the number of integers it should put in the buffer. It also has a local sum of put numbers.

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

 class Producer extends Thread {
     int nrTrials;
     int localSum;

     public Producer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           Random r  = new Random();
           int toPut = r.nextInt();
           buffer.put(toPut);
           localSum += toPut;
       }
       putSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (producer)

We have use two global AtomicIntegers to keep track of the global sum of put/remove

The producer is initialized with the number of integers it should put in the buffer. It also has a local sum of put numbers.

The producer generates a local random number to puts it in the buffer. Then it updates the local sum of put numbers

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

 class Producer extends Thread {
     int nrTrials;
     int localSum;

     public Producer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           Random r  = new Random();
           int toPut = r.nextInt();
           buffer.put(toPut);
           localSum += toPut;
       }
       putSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
   }
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (producer)

We have use two global AtomicIntegers to keep track of the global sum of put/remove

The producer is initialized with the number of integers it should put in the buffer. It also has a local sum of put numbers.

The producer generates a local random number to puts it in the buffer. Then it updates the local sum of put numbers

Finally, the global put sum is updated with the local sum of the producer.

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

 class Producer extends Thread {
     int nrTrials;
     int localSum;

     public Producer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           Random r  = new Random();
           int toPut = r.nextInt();
           buffer.put(toPut);
           localSum += toPut;
       }
       putSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (producer)

We have use two global AtomicIntegers to keep track of the global sum of put/remove

The producer is initialized with the number of integers it should put in the buffer. It also has a local sum of put numbers.

The producer generates a local random number to puts it in the buffer. Then it updates the local sum of put numbers

Finally, the global put sum is updated with the local sum of the producer.

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

 class Producer extends Thread {
     int nrTrials;
     int localSum;

     public Producer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           Random r  = new Random();
           int toPut = r.nextInt();
           buffer.put(toPut);
           localSum += toPut;
       }
       putSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
   }
```

As expected, we use a barrier to maximize contention.

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (consumer)

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

  class Consumer extends Thread {
     int nrTrials;
     int localSum;

     public Consumer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           localSum += buffer.take();
       }
       takeSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
}
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (consumer)

The consumer is initialized with the number of integers it should take from the buffer. I also has a local sum of taken numbers.

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

  class Consumer extends Thread {
     int nrTrials;
     int localSum;

     public Consumer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           localSum += buffer.take();
       }
       takeSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
}
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (consumer)

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

  class Consumer extends Thread {
     int nrTrials;
     int localSum;

     public Consumer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           localSum += buffer.take();
       }
       takeSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
}
```

The consumer is initialized with the number of integers it should take from the buffer. I also has a local sum of taken numbers.

The consumer takes an element from the buffer and it updates the local sum of taken integers

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (consumer)

The consumer is initialized with the number of integers it should take from the buffer. I also has a local sum of taken numbers.

The consumer takes an element from the buffer and it updates the local sum of taken integers

Finally, the global taken sum is updated with the local sum of the consumer.

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

  class Consumer extends Thread {
     int nrTrials;
     int localSum;

     public Consumer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           localSum += buffer.take();
       }
       takeSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
}
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (consumer)

The consumer is initialized with the number of integers it should take from the buffer. I also has a local sum of taken numbers.

The consumer takes an element from the buffer and it updates the local sum of taken integers

Finally, the global taken sum is updated with the local sum of the consumer.

```
Class BoundedBufferTest {

 BoundedBuffer buffer;
 AtomicInteger putSum; // global sum of put numbers
 AtomicInteger takeSum; // global sum of taken numbers

  class Consumer extends Thread {
     int nrTrials;
     int localSum;

     public Consumer(int nrTrials) {
         this.nrTrials = nrTrials;
         this.localSum = 0;
     }

     public void run() {
         try {
       barrier.await();
       for (int i = 0; i < nrTrials; i++) {
           localSum += buffer.take();
       }
       takeSum.addAndGet(localSum);
       barrier.await();
         } catch (InterruptedException |
     BrokenBarrierException e) {
       e.printStackTrace();
         }
     }
    }
}
```

As expected, we use a barrier to maximize contention.

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (test)

```
public void putTakeTest(int nrThreads,
                        int nrTrials,
                        int bufferSize) {


    // init buffer
    buffer  = new BoundedBufferSemaphore<Integer>(bufferSize);
    // init barrier
    barrier = new CyclicBarrier((nrThreads*2) + 1);

    for (int i = 0; i < nrThreads; i++) {
        pool.execute(new Producer(nrTrials));
        pool.execute(new Consumer(nrTrials));
    }

    try {
        barrier.await();
        barrier.await();
    } catch (InterruptedException | BrokenBarrierException e)
    {
        e.printStackTrace();
    }
    assert(putSum.get() == takeSum.get());
   }
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (test)

The test has 3 parameters: the number of pairs of producer consumers, the number of put/take that each producer/consumer must perform and the size of the buffer

```
public void putTakeTest(int nrThreads,
                        int nrTrials,
                        int bufferSize) {

    // init buffer
    buffer  = new BoundedBufferSemaphore<Integer>(bufferSize);
    // init barrier
    barrier = new CyclicBarrier((nrThreads*2) + 1);

    for (int i = 0; i < nrThreads; i++) {
        pool.execute(new Producer(nrTrials));
        pool.execute(new Consumer(nrTrials));
    }

    try {
        barrier.await();
        barrier.await();
    } catch (InterruptedException | BrokenBarrierException e)
    {
        e.printStackTrace();
    }
    assert(putSum.get() == takeSum.get());
   }
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (test)

The test has 3 parameters: the number of pairs of producer consumers, the number of put/take that each producer/consumer must perform and the size of the buffer

We initialize the buffer and the barrier

```
public void putTakeTest(int nrThreads,
                        int nrTrials,
                        int bufferSize) {

      // init buffer
      buffer  = new BoundedBufferSemaphore<Integer>(bufferSize);
      // init barrier
      barrier = new CyclicBarrier((nrThreads*2) + 1);

      for (int i = 0; i < nrThreads; i++) {
          pool.execute(new Producer(nrTrials));
          pool.execute(new Consumer(nrTrials));
      }

      try {
          barrier.await();
          barrier.await();
      } catch (InterruptedException | BrokenBarrierException e)
      {
          e.printStackTrace();
      }
      assert(putSum.get() == takeSum.get());
    }
```

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (test)

The test has 3 parameters: the number of pairs of producer consumers, the number of put/take that each producer/consumer must perform and the size of the buffer

We initialize the buffer and the barrier

```
public void putTakeTest(int nrThreads,
                        int nrTrials,
                        int bufferSize) {

    // init buffer
    buffer  = new BoundedBufferSemaphore<Integer>(bufferSize);
    // init barrier
    barrier = new CyclicBarrier((nrThreads*2) + 1);

    for (int i = 0; i < nrThreads; i++) {
        pool.execute(new Producer(nrTrials));
        pool.execute(new Consumer(nrTrials));
    }

    try {
        barrier.await();
        barrier.await();
    } catch (InterruptedException | BrokenBarrierException e)
    {
        e.printStackTrace();
    }
    assert(putSum.get() == takeSum.get());
  }
```

We execute a producer and consumer in each iteration

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (test)

The test has 3 parameters: the number of pairs of producer consumers, the number of put/take that each producer/consumer must perform and the size of the buffer

We initialize the buffer and the barrier

As a reminder, the first await is to maximize contention, and the second to wait for all threads to finish execution

```
public void putTakeTest(int nrThreads,
                        int nrTrials,
                        int bufferSize) {

    // init buffer
    buffer  = new BoundedBufferSemaphore<Integer>(bufferSize);
    // init barrier
    barrier = new CyclicBarrier((nrThreads*2) + 1);

    for (int i = 0; i < nrThreads; i++) {
        pool.execute(new Producer(nrTrials));
        pool.execute(new Consumer(nrTrials));
    }

    try {
        barrier.await();
        barrier.await();
    } catch (InterruptedException | BrokenBarrierException e)
    {
        e.printStackTrace();
    }
    assert(putSum.get() == takeSum.get());
  }
```

We execute a producer and consumer in each iteration

Concurrent Correctness Test – Bounded Buffer

2. Testing setup (test)

```
public void putTakeTest(int nrThreads,
                        int nrTrials,
                        int bufferSize) {

    // init buffer
    buffer  = new BoundedBufferSemaphore<Integer>(bufferSize);
    // init barrier
    barrier = new CyclicBarrier((nrThreads*2) + 1);

    for (int i = 0; i < nrThreads; i++) {
        pool.execute(new Producer(nrTrials));
        pool.execute(new Consumer(nrTrials));
    }

    try {
        barrier.await();
        barrier.await();
    } catch (InterruptedException | BrokenBarrierException e)
    {
        e.printStackTrace();
    }
    assert(putSum.get() == takeSum.get());
}
```

The test has 3 parameters: the number of pairs of producer consumers, the number of put/take that each producer/consumer must perform and the size of the buffer

We initialize the buffer and the barrier

We execute a producer and consumer in each iteration

As a reminder, the first await is to maximize contention, and the second to wait for all threads to finish execution

Finally, we check that our property holds after executing the test. The test relies on the correctness of AtomicInteger.

2. Testing setup (test)

```
public void putTakeTest(int nrThreads,
                        int nrTrials,
                        int bufferSize) {

    // init buffer
    buffer  = new BoundedBufferSemaphore<Integer>(bufferSize);
    // init barrier
    barrier = new CyclicBarrier((nrThreads*2) + 1);

    for (int i = 0; i < nrThreads; i++) {
        pool.execute(new Producer(nrTrials));
        pool.execute(new Consumer(nrTrials));
    }

    try {
        barrier.await();
        barrier.await();
    } catch (InterruptedException | BrokenBarrierException e)
    {
        e.printStackTrace();
    }
    assert(putSum.get() == takeSum.get());
}
```

The test has 3 parameters: the number of pairs of producer consumers, the number of put/take that each producer/consumer must perform and the size of the buffer

We initialize the buffer and the barrier

We execute a producer and consumer in each iteration

As a reminder, the first await is to maximize contention, and the second to wait for all threads to finish execution

Finally, we check that our property holds after executing the test. The test relies on the correctness of AtomicInteger.

Let's run the tests!

Concurrent Correctness Test – Bounded Buffer

- Here is the implementation in Goetz p. 249

```
@ThreadSafe
public class BoundedBuffer<E> {
    private final Semaphore availableItems, availableSpaces;
    @GuardedBy("this") private final E[] items;
    @GuardedBy("this") private int putPosition = 0, takePosition = 0;

    public BoundedBuffer(int capacity) {
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }
    public boolean isEmpty() {
        return availableItems.availablePermits() == 0;
    }
    public boolean isFull() {
        return availableSpaces.availablePermits() == 0;
    }

    public void put(E x) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(x);
        availableItems.release();
    }
    public E take() throws InterruptedException {
        availableItems.acquire();
        E item = doExtract();
        availableSpaces.release();
        return item;
    }

    private synchronized void doInsert(E x) {
        int i = putPosition;
        items[i] = x;
        putPosition = (++i == items.length)? 0 : i;
    }
    private synchronized E doExtract() {
        int i = takePosition;
        E x = items[i];
        items[i] = null;
        takePosition = (++i == items.length)? 0 : i;
        return x;
    }
}
```

Concurrent Correctness Test – Bounded Buffer


```
@ThreadSafe
public class BoundedBuffer<E> {
    private final Semaphore availableItems, availableSpaces;
    @GuardedBy("this") private final E[] items;
    @GuardedBy("this") private int putPosition = 0, takePosition = 0;

    public BoundedBuffer(int capacity) {
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }
    public boolean isEmpty() {
        return availableItems.availablePermits() == 0;
    }
    public boolean isFull() {
        return availableSpaces.availablePermits() == 0;
    }

    public void put(E x) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(x);
        availableItems.release();
    }
    public E take() throws InterruptedException {
        availableItems.acquire();
        E item = doExtract();
        availableSpaces.release();
        return item;
    }
```

- Here is the implementation in Goetz p. 249

Do you think these methods are thread-safe?
(the question is a bit ambiguous on purpose, you can try asking me to be more precise if you will)

```
    }
    private synchronized E doExtract() {
        int i = takePosition;
        E x = items[i];
        items[i] = null;
        takePosition = (++i == items.length)? 0 : i;
        return x;
    }
}
```

Deadlocks

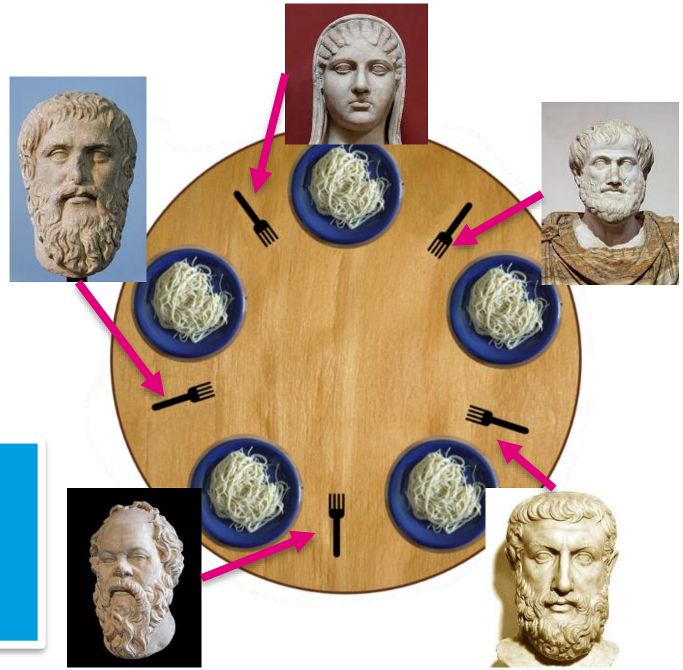
- A deadlock occurs when all threads are waiting locks hold by other threads
  - *which will never happen as all threads are waiting*

- Standard (but not very realistic) example:
  - Dinning philosophers by E.W. Dijkstra
  - Philosophers only think and eat
  - A philosopher must pick both left and right forks to start eating

Deadlocks

- A deadlock occurs when all threads are waiting locks hold by other threads
  - *which will never happen as all threads are waiting*

- Standard (but not very realistic) example:
  - Dinning philosophers by E.W. Dijkstra
  - Philosophers only think and eat
  - A philosopher must pick both left and right forks to start eating



What happens if we reach a state where all philosophers have grabbed their right fork?

Testing deadlocks

- Testing for deadlocks is not really possible

Testing deadlocks

- Testing for deadlocks is not really possible

Why?

Testing deadlocks

- Testing for deadlocks is not really possible

- We should define a maximum duration for an operation, after which, we deem the interleaving as deadlocked

- If, when a running a test, we observe that the program doesn’t terminate for a long time, it might be due to deadlocks
  - Let’s run the previous test on an implementation of a bounded buffer with deadlocks

Formal Verification

Limitations of testing

- Testing is a extremely useful technique, which is the de-facto approach in industry
  - You should extensively tests all your programs!
- However, it cannot be used to prove the absence bugs (remember the first slides)
- Tests can be seen as random interleaving generators ☺
  - They stimulate the system to produce different interleavings
  - For most systems, it is virtually impossible to write a set of tests that cover all possible interleavings in the system

Formal Verification

- Formal verification is a technology that aims to *prove* that a program satisfy a specification (properties)

- It treats programs and properties as mathematical objects

- Using mathematical reasoning it is possible to prove that programs satisfy their specifications (i.e., for all possible interleavings)
  - Manually: Proof assistants (Coq, Isabelle, etc.)
  - Automatically: SAT solvers, SMT solvers, ***model-checking***, static verification, symbolic execution, etc.

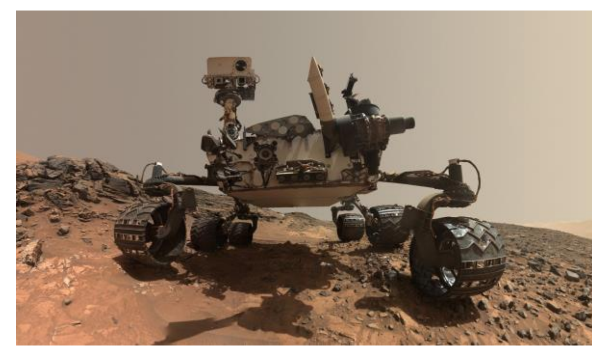
Model-checking

- Model-checking transforms programs into a *finite-state* models that encapsulate all possible interleavings in the system
  - Automata, Kripke structures, binary decision diagrams, etc.
- Properties are specified in some type of logic
  - Linear Temporal Logic (LTL), Computational Tree Logic (CTL), First-Order Logic (FOL), propositional logic, etc.
- The model of the program and the property are typically expressed in the same language so it is possible to automatically check whether they are satisfied
- Model-checking has been very successful in hardware verification at Intel

JavaPathFinder (switch to rocket science)

- JavaPathFinder is (among other things) a model-checker for Java programs

- It is developed at the Jet Propulsion Lab (JPL) at NASA
  - It was used to verify part of the system in the Curiosity rover that landed in Mars in 2012



- Let’s look at a few examples of using JavaPathFinder

Threads in Java – Example II

- Altogether (not executa
the executable program

HARDer: What is the minimum value of `counter` that this program can print?

```
long counter = 0;
final long PEOPLE  = 10_000;


Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile2.start();
turnstile2.join();turnstile2.join();
System.out.println(counter+" people entered");



public class Turnstile extends Thread {
    public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            counter++;
        }
    }
}
```

Another déjà vu?

Let's use javapathfinder to automatically get the answer

Threads in Java – Example II

- Altogether (not executa
the executable program

HARDer: What is the minimum value of **`counter`** that this program can print?

Can testing be used to answer this question?

```
Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile2.start();
turnstile2.join();turnstile2.join();
System.out.println(counter+" people entered");



public class Turnstile extends Thread {
    public void run() {
        for (int i = 0; i < PEOPLE; i++) {
          counter++;
        }
    }
}
```

Another déjà vu?

Let's use javapathfinder to automatically get the answer

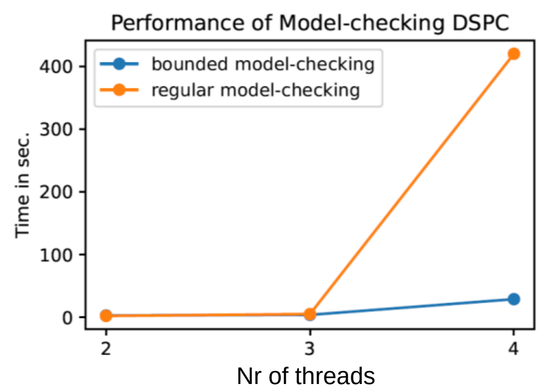
Bounded Buffer in JavaPathFinder

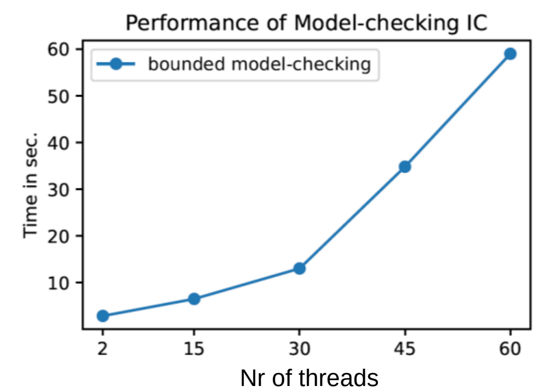
- Let’s now look at an example of finding deadlocks with JavaPathFinder in a buggy implementation of a bounded buffer

Too good to be true…

- Ok Raúl, if formal verification is so good, why isn't everyone using it all the time?
  - Welcome to the **state explosion problem**! (among other things)
  - Even for small programs the computational cost of proving that the system satisfies its specification can be astronomically expensive

- **The use of abstractions and/or narrow down the problem domain has helped formal verification to scale better**
  - Example: Proving that an IoT system satisfies a privacy requirement (my own work-in-progress paper)





Formal Verification – In Industry and at ITU

- Many companies have started to use formal verification in their software development process, so it might be a good asset to have in your toolbox

- At ITU, you can learn more about formal verification in the software analysis specialization, e.g., in the courses
  - Advanced Software Analysis
  - Program verification
  - …

- I believe modern software engineers should be aware of this technology and trained to use it (warning: personal opinion)

MSc thesis in Formal Verification

- Formal verification is an active topic of research
- ***<u>If you found this topic interesting, feel free to contact me regarding MSc thesis projects</u>***
  - Also keep an eye on people working at the Software Quality Group (SQUARE), the Centre of Security and Trust (CISAT) and the Programming, Logic and Semantics (PLS) group
- My interests focus on using formal verification to
  - Prove that systems satisfy legal privacy requirements (e.g., GDPR)
  - Quantify privacy risks in ML
  - Prove properties in probabilistic programs

Agenda

- Intro to concurrency properties
- Testing
  - Intro to JUnit 5
  - Counter
  - Bounded Buffer
  - Deadlocks
- Formal Verification
  - Java Path Finder