Lincheck
Testing concurrency on JVM

Sokolova Maria, Hydra 2021
@sokolova_m

Set-up check

For the workshop you will need:

1. IntelliJ IDEA

2. Cloned lincheck-workshop project:

```
$ git clone https://git.io/JZp6P
```

Writing concurrent code is pain

Writing concurrent code is pain

… testing it is not much easier!

Lincheck

Lincheck = **Lin**earizability **Check**er (not only linearizability)

https://github.com/Kotlin/kotlinx-lincheck

Lincheck

Lincheck = **Lin**earizability **Check**er (not only linearizability)

https://github.com/Kotlin/kotlinx-lincheck

1. Generates a random scenario
2. Executes a scenario using either the *stress* or the *model checking strategy*
3. Verifies the results

Counter

```kotlin
class Counter {
    @Volatile var value = 0

    fun addAndGet(delta: Int): Int {
        value += delta
        return value
    }
}
```

Let’s write a test

```
class Counter {
    @Volatile var value = 0

    fun addAndGet(delta: Int): Int {
        value += delta
        return value
    }
}
```

```
class CounterTest {
    private val c = Counter()




}
```

Initial state

Let’s write a test

```kotlin
class Counter {
    @Volatile var value = 0

    fun addAndGet(delta: Int): Int {
        value += delta
        return value
    }
}
```

```kotlin
class CounterTest {
    private val c = Counter()

    @Operation
    fun addAndGet(delta: Int) = c.addAndGet(delta)


}
```

Initial state

Counter operations

Let’s write a test

```kotlin
class Counter {
    @Volatile var value = 0

    fun addAndGet(delta: Int): Int {
        value += delta
        return value
    }
}
```

```kotlin
class CounterTest {
    private val c = Counter()

    @Operation
    fun addAndGet(delta: Int) = c.addAndGet(delta)

    @Test
    fun test() = StressOptions()
                    .check(this::class)
}
```

Initial state

Counter operations

Magic check

Let’s write a test

```
class Counter {
   @Volatile var value = 0

   fun addAndGet(delta: Int): Int {
       value += delta
       return value
   }
}
```

```
class CounterTest {
   private val c = Counter()

   @Operation
   fun addAndGet(delta: Int) = c.addAndGet(delta)

   @Test
   fun test() = StressOptions()
                   .check(this::class)
}
```

Initial state

Counter operations

Run the test!

Magic check

Run Counter test

1. Checkout **1.1-counter** branch:

   ```
   $ git checkout 1.1-counter
   ```

2. CounterTest.kt

3. Run `runStressTest()`, `runModelCheckingTest()`

Failed?

```
class Counter {
   @Volatile var value = 0

   fun addAndGet(delta: Int): Int {
       value += delta
       return value
   }
}
```

```
class CounterTest {
   private val c = Counter()

   @Operation
   fun addAndGet(delta: Int) = c.addAndGet(delta)

   @Test
   fun test() = StressOptions()
                  .check(this::class)
}
```

```
java.lang.AssertionError: Invalid
interleaving found:
= Invalid execution results: =
Parallel part:
| addAndGet(1): 1 | addAndGet(1): 1 |
```

Trace the error

```kotlin
class Counter {
    @Volatile var value = 0

    fun addAndGet(delta: Int): Int {
        value += delta
        return value
    }
}
```

```kotlin
class CounterTest {
    private val c = Counter()

    @Operation
    fun addAndGet(delta: Int) = c.addAndGet(delta)

    @Test
    fun test() = StressOptions()
                    .check(this::class)
}
```

Stress test

Trace the error

```kotlin
class Counter {
   @Volatile var value = 0

   fun addAndGet(delta: Int): Int {
       value += delta
       return value
   }
}
```

```kotlin
class CounterTest {
   private val c = Counter()

   @Operation
   fun addAndGet(delta: Int) = c.addAndGet(delta)

   @Test
   fun test() = ModelCheckingOptions()
                  .check(this::class)
}
```

Run the test!

Managed strategy

Trace the error

```
org.jetbrains.kotlinx.lincheck.LincheckAssertionError:
= Invalid execution results =
Parallel part:
| addAndGet(1): 1 | addAndGet(2): 2 |
= The following interleaving leads to the error =
Parallel part trace:
|                      | addAndGet(2)                                                      |
|                      |   addAndGet(2): 2 at CounterTest.addAndGet(CounterTest.kt:11) |
|                      |     value.READ: 0 at Counter.addAndGet(Counter.kt:7)          |
|                      |     switch                                                        |
| addAndGet(1): 1      |                                                                   |
|   thread is finished |                                                                   |
|                      |     value.WRITE(2) at Counter.addAndGet(Counter.kt:7)         |
|                      |     value.READ: 2 at Counter.addAndGet(Counter.kt:8)          |
|                      |   result: 2                                                       |
|                      |   thread is finished                                              |
```

Trace the error

```
org.jetbrains.kotlinx.lincheck.LincheckAssertionError:
= Invalid execution results =
Parallel part:
| addAndGet(1): 1 | addAndGet(2): 2 |
= The following interleaving leads to the error =
Parallel part trace:
|                      | addAndGet(2)                                           |
|                      |   addAndGet(2): 2 at CounterTest.addAndGet(CounterTest.kt:11) |
|                      |     value.READ: 0 at Counter.addAndGet(Counter.kt:7)   |
|                      |     switch                                             |
| addAndGet(1): 1      |                                                        |
|   thread is finished |                                                        |
|                      |     value.WRITE(2) at Counter.addAndGet(Counter.kt:7)  |
|                      |     value.READ: 2 at Counter.addAndGet(Counter.kt:8)   |
|                      |   result: 2                                            |
|                      |   thread is finished                                   |
```

Trace the error

```
org.jetbrains.kotlinx.lincheck.LincheckAssertionError:
= Invalid execution results =
Parallel part:
| addAndGet(1): 1 | addAndGet(2): 2 |
= The following interleaving leads to the error =
Parallel part trace:
|                     | addAndGet(2)                                         |
|                     |   addAndGet(2): 2 at CounterTest.addAndGet(CounterTest.kt:11) |
|                     |     value.READ: 0 at Counter.addAndGet(Counter.kt:7)    |
|                     |     switch                                             |
| addAndGet(1): 1     |                                                        |
|   thread is finished |                                                       |
|                     |     value.WRITE(2) at Counter.addAndGet(Counter.kt:7)  |
|                     |     value.READ: 2 at Counter.addAndGet(Counter.kt:8)   |
|                     |   result: 2                                            |
|                     |   thread is finished                                   |
```

Trace the error

```
org.jetbrains.kotlinx.lincheck.LincheckAssertionError:
= Invalid execution results =
Parallel part:
| addAndGet(1): 1 | addAndGet(2): 2 |
= The following interleaving leads to the error =
Parallel part trace:
|                      | addAndGet(2)                                               |
|                      |   addAndGet(2): 2 at CounterTest.addAndGet(CounterTest.kt:11) |
|                      |     value.READ: 0 at Counter.addAndGet(Counter.kt:7)       |
|                      |     switch                                                 |
| addAndGet(1): 1      |                                                            |
|   thread is finished |                                                            |
|                      |     value.WRITE(2) at Counter.addAndGet(Counter.kt:7)      |
|                      |     value.READ: 2 at Counter.addAndGet(Counter.kt:8)       |
|                      |   result: 2                                                |
|                      |   thread is finished                                       |
```

Trace the error

```
org.jetbrains.kotlinx.lincheck.LincheckAssertionError:
= Invalid execution results =
Parallel part:
| addAndGet(1): 1 | addAndGet(2): 2 |
= The following interleaving leads to the error =
Parallel part trace:
|                      | addAndGet(2)                                          |
|                      |   addAndGet(2): 2 at CounterTest.addAndGet(CounterTest.kt:11) |
|                      |     value.READ: 0 at Counter.addAndGet(Counter.kt:7)  |
|                      |     switch                                            |
| addAndGet(1): 1      |                                                       |
|   thread is finished |                                                       |
|                      |     value.WRITE(2) at Counter.addAndGet(Counter.kt:7) |
|                      |     value.READ: 2 at Counter.addAndGet(Counter.kt:8)  |
|                      |   result: 2                                           |
|                      |   thread is finished                                  |
```

Trace the error

```
org.jetbrains.kotlinx.lincheck.LincheckAssertionError:
= Invalid execution results =
Parallel part:
| addAndGet(1): 1 | addAndGet(2): 2 |
= The following interleaving leads to the error =
Parallel part trace:
|                      | addAndGet(2)                                               |
|                      |   addAndGet(2): 2 at CounterTest.addAndGet(CounterTest.kt:11) |
|                      |     value.READ: 0 at Counter.addAndGet(Counter.kt:7)        |
|                      |     switch                                                 |
| addAndGet(1): 1      |                                                            |
|   thread is finished |                                                            |
|                      |     value.WRITE(2) at Counter.addAndGet(Counter.kt:7)       |
|                      |     value.READ: 2 at Counter.addAndGet(Counter.kt:8)        |
|                      |   result: 2                                                |
|                      |   thread is finished                                       |
```

Trace the error

```
org.jetbrains.kotlinx.lincheck.LincheckAssertionError:
= Invalid execution results =
Parallel part:
| addAndGet(1): 1 | addAndGet(2): 2 |
= The following interleaving leads to the error =
Parallel part trace:
|                      | addAndGet(2)                                                     |
|                      |   addAndGet(2): 2 at CounterTest.addAndGet(CounterTest.kt:11)   |
|                      |     value.READ: 0 at Counter.addAndGet(Counter.kt:7)             |
|                      |     switch                                                       |
| addAndGet(1): 1      |                                                                  |
|   thread is finished |                                                                  |
|                      |     value.WRITE(2) at Counter.addAndGet(Counter.kt:7)            |
|                      |     value.READ: 2 at Counter.addAndGet(Counter.kt:8)             |
|                      |   result: 2                                                      |
|                      |   thread is finished                                             |
```

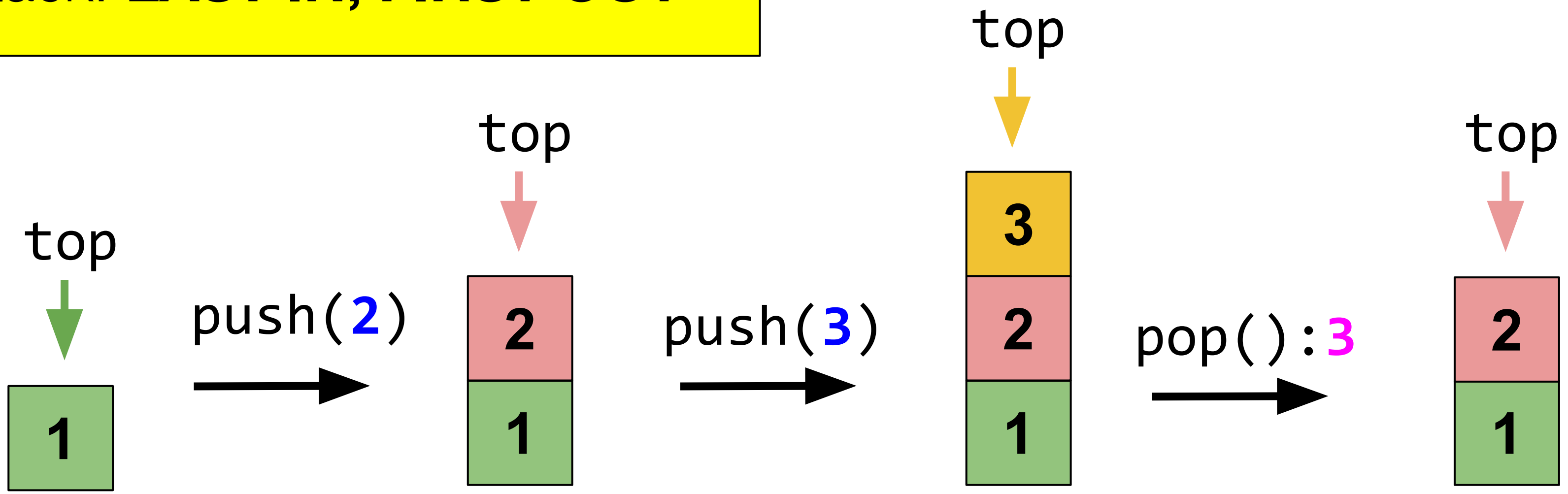
Correctness

Sequential algorithm → Sequential specification on operations

Correctness

Sequential algorithm → Sequential specification on operations

Stack: **LAST IN, FIRST OUT**



Correctness

Sequential algorithm → Sequential specification on operations

Concurrent algorithm → Linearizability (usually)

Correctness

Sequential algorithm → Sequential specification on operations

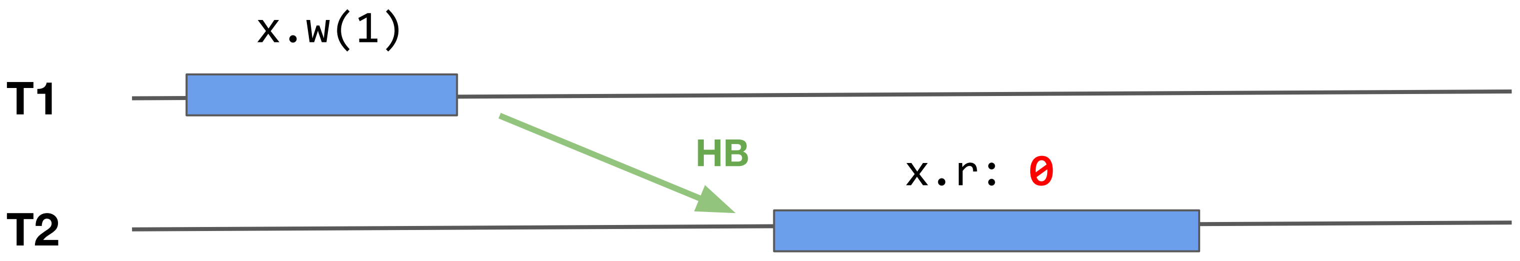
Concurrent algorithm → Linearizability (usually)

```
val c = Counter()
```

`c.addAndGet(1): 1`	
	`c.addAndGet(1): 2`

Execution is **linearizable** iff $\exists$ equivalent sequential execution wrt *happens-before order*

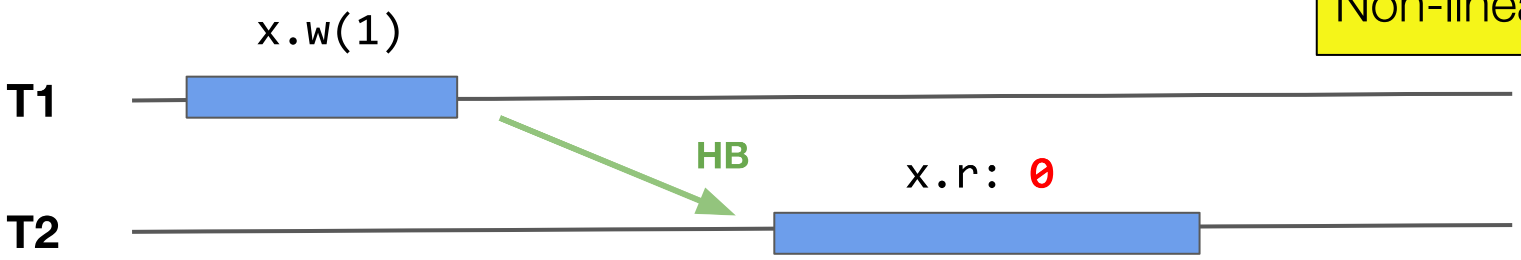
Execution is **linearizable** iff ∃ equivalent sequential execution wrt *happens-before order*

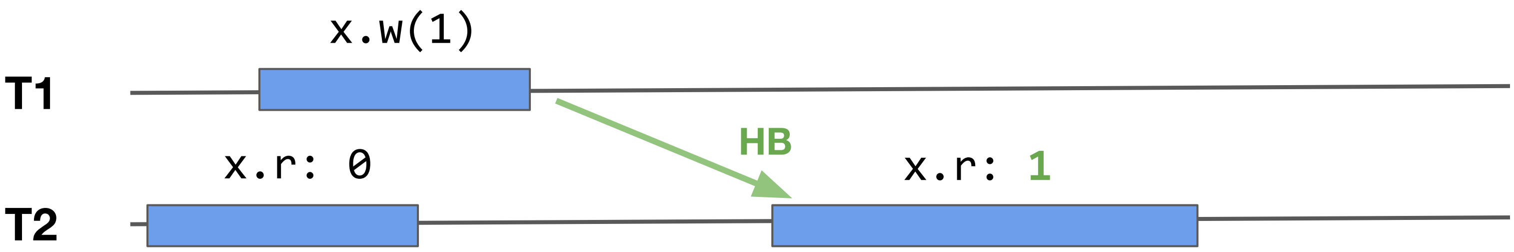


Execution is **linearizable** iff ∃ equivalent sequential execution wrt *happens-before order*

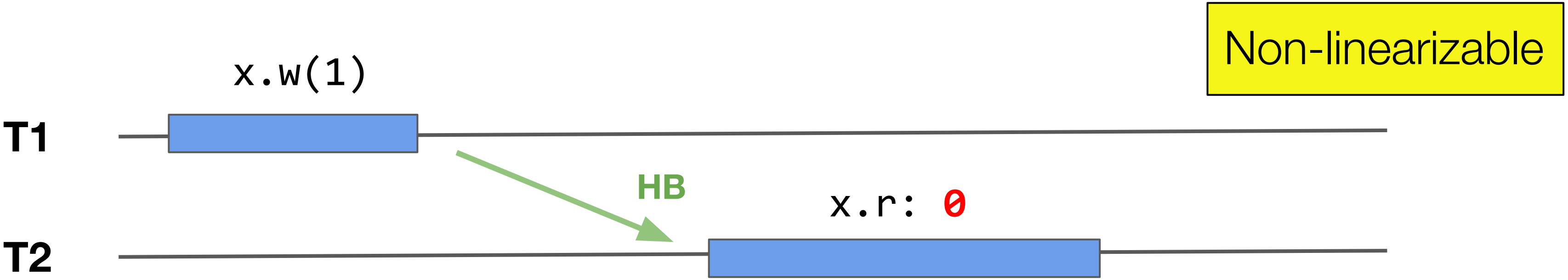
Execution is **linearizable** iff ∃ equivalent sequential execution wrt *happens-before order*

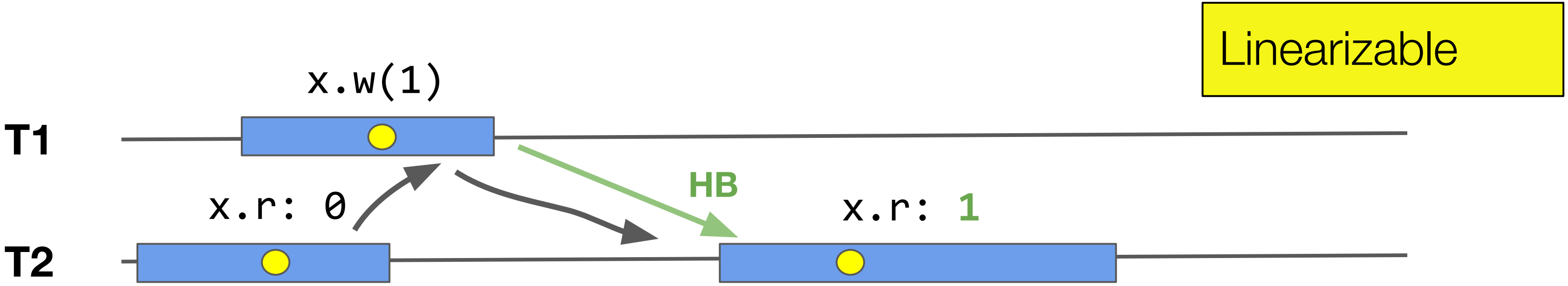
Non-linearizable





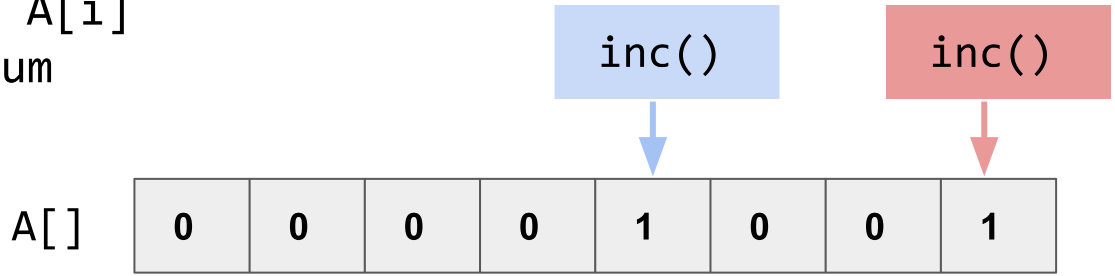
Execution is **linearizable** iff ∃ equivalent sequential execution wrt *happens-before order*





Scalable counter

```
fun inc() {
  i := rand(K)
  FAA(&A[i], +1)
}

fun sum() {
  sum := 0
  for (i := 0; i < K; i++)
    sum += A[i]
  return sum
}
```

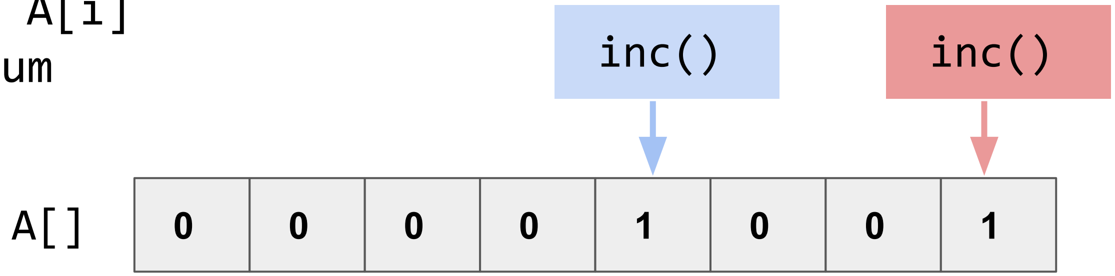


Scalable counter

```
fun inc() {
  i := rand(K)
  FAA(&A[i], +1)
}

fun sum() {
  sum := 0
  for (i := 0; i < K; i++)
    sum += A[i]
  return sum
}
```

`java.util.concurrent.atomic.LongAdder`

- Preferable to AtomicLong when multiple threads update a common sum
- Increases array size under high contention



Write a test for LongAdder

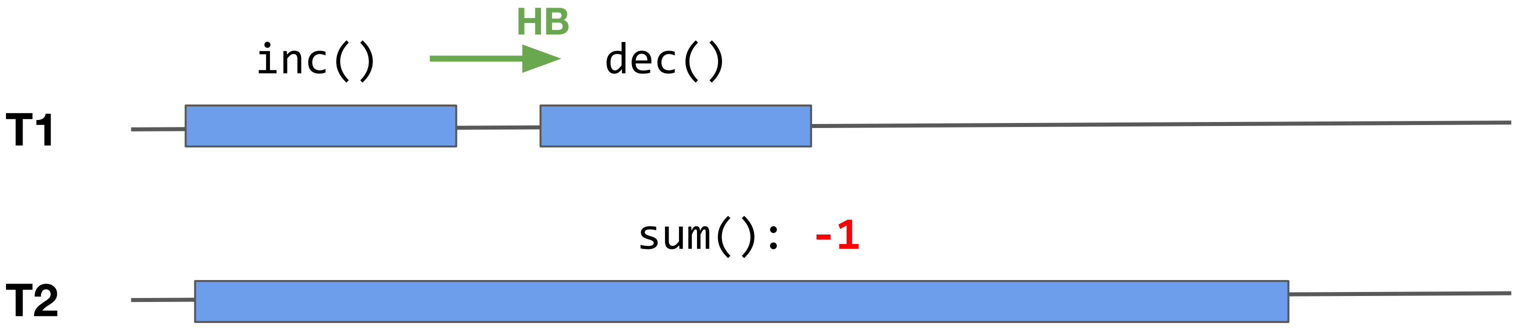
1. Checkout **1.2-longAdder** branch:

   ```
   $ git checkout 1.2-longAdder
   ```

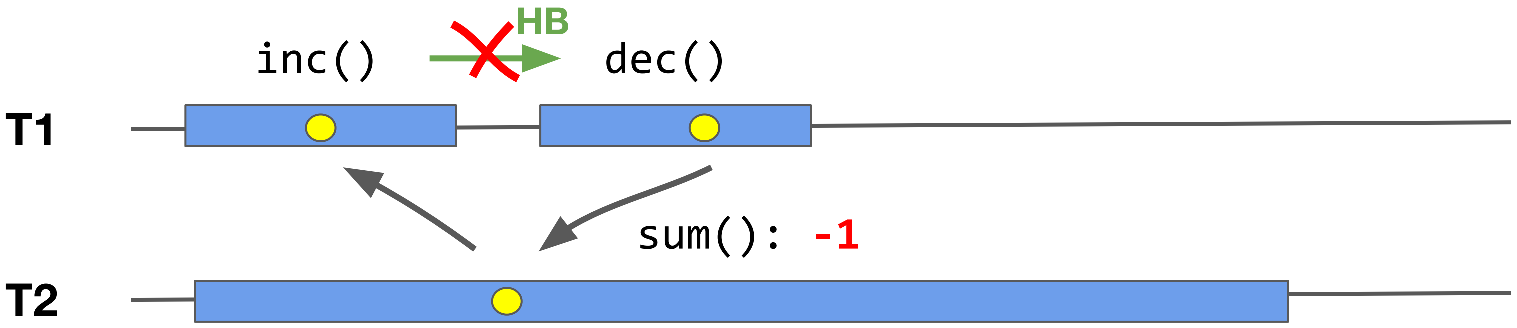
2. Complete `TODO()` blocks in `LongAdderTest` class

3. Run `runModelCheckingTest()`

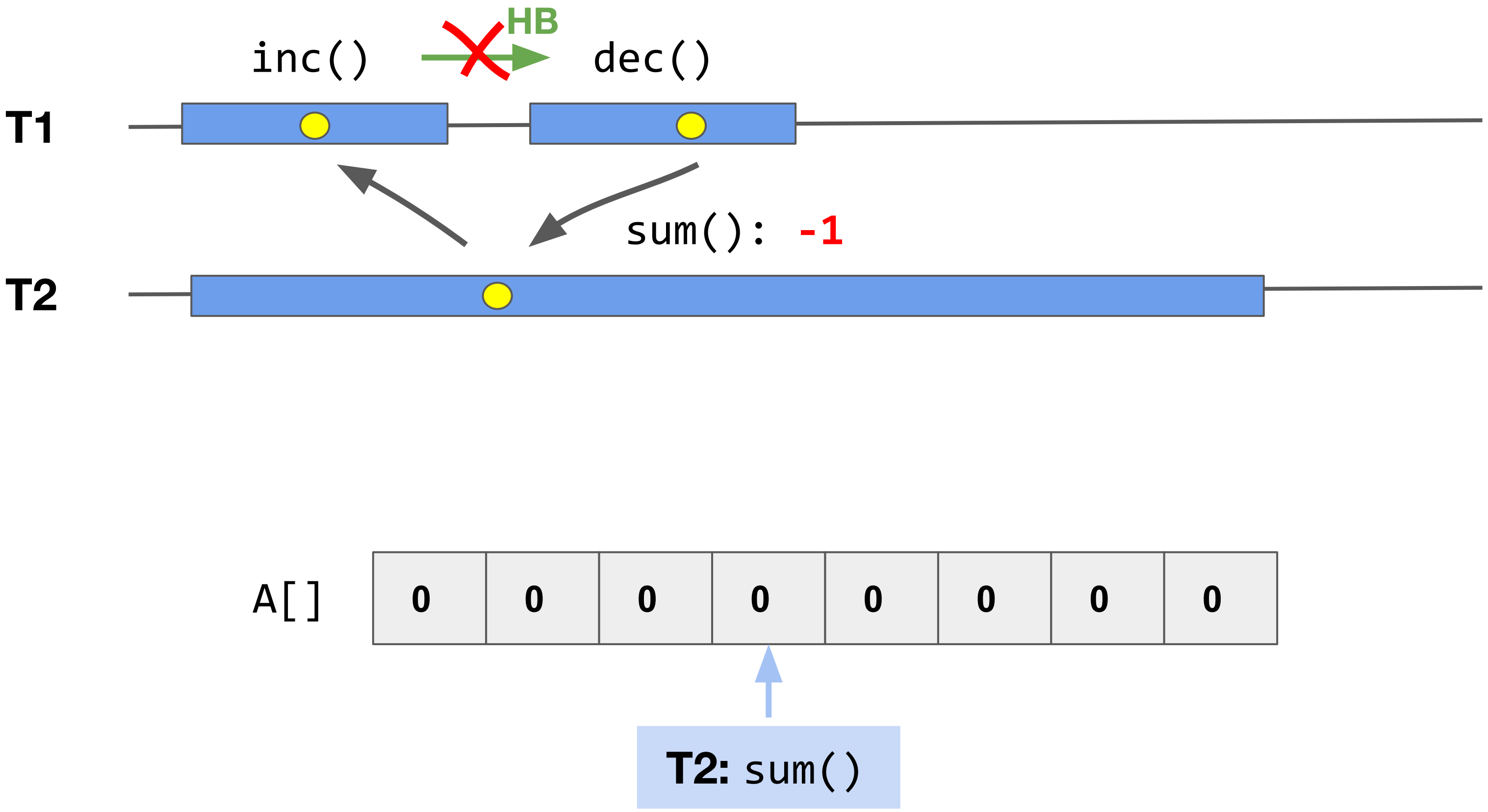
Why LongAdder is not linearizable?



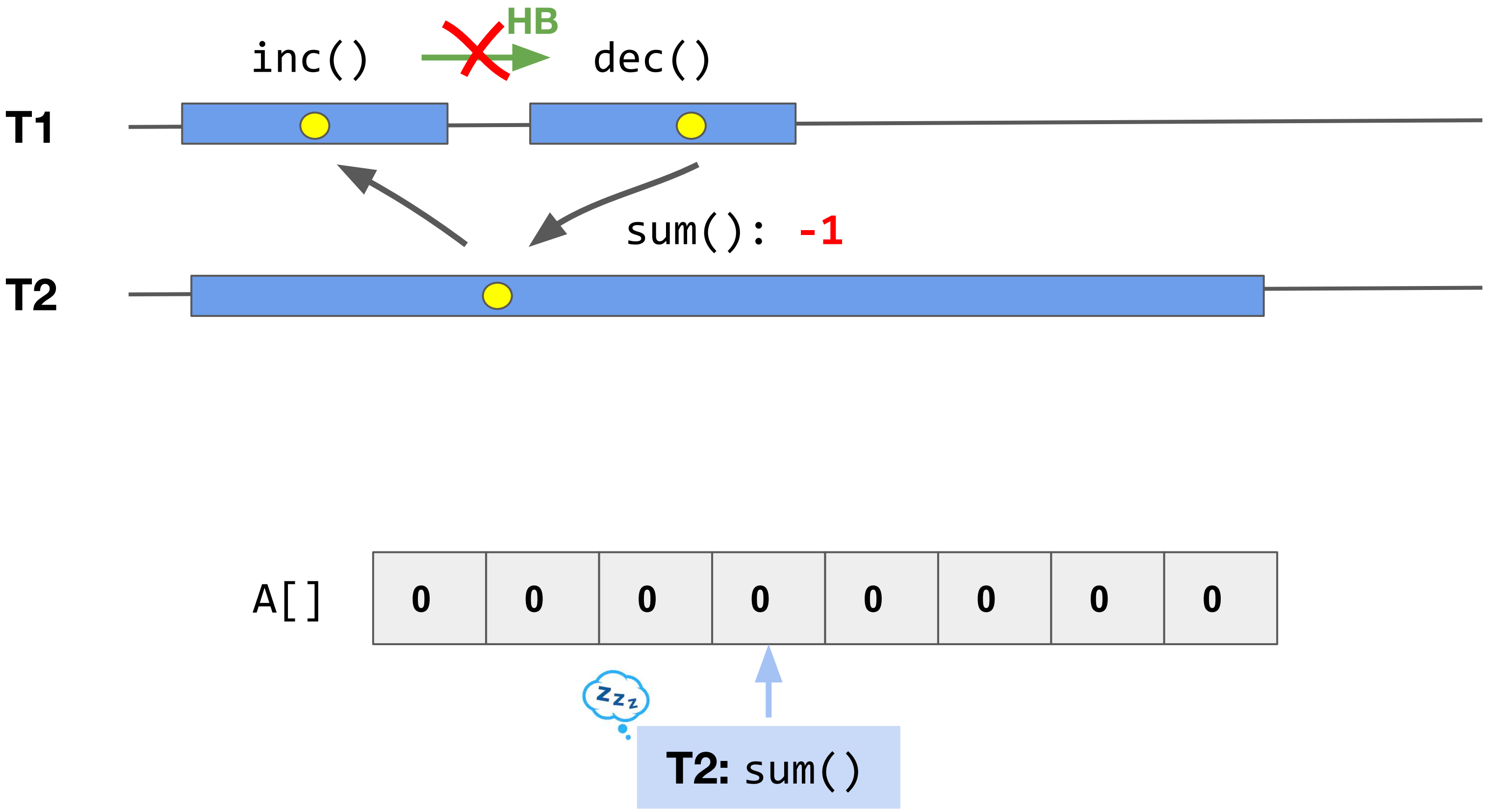
Why LongAdder is not linearizable?



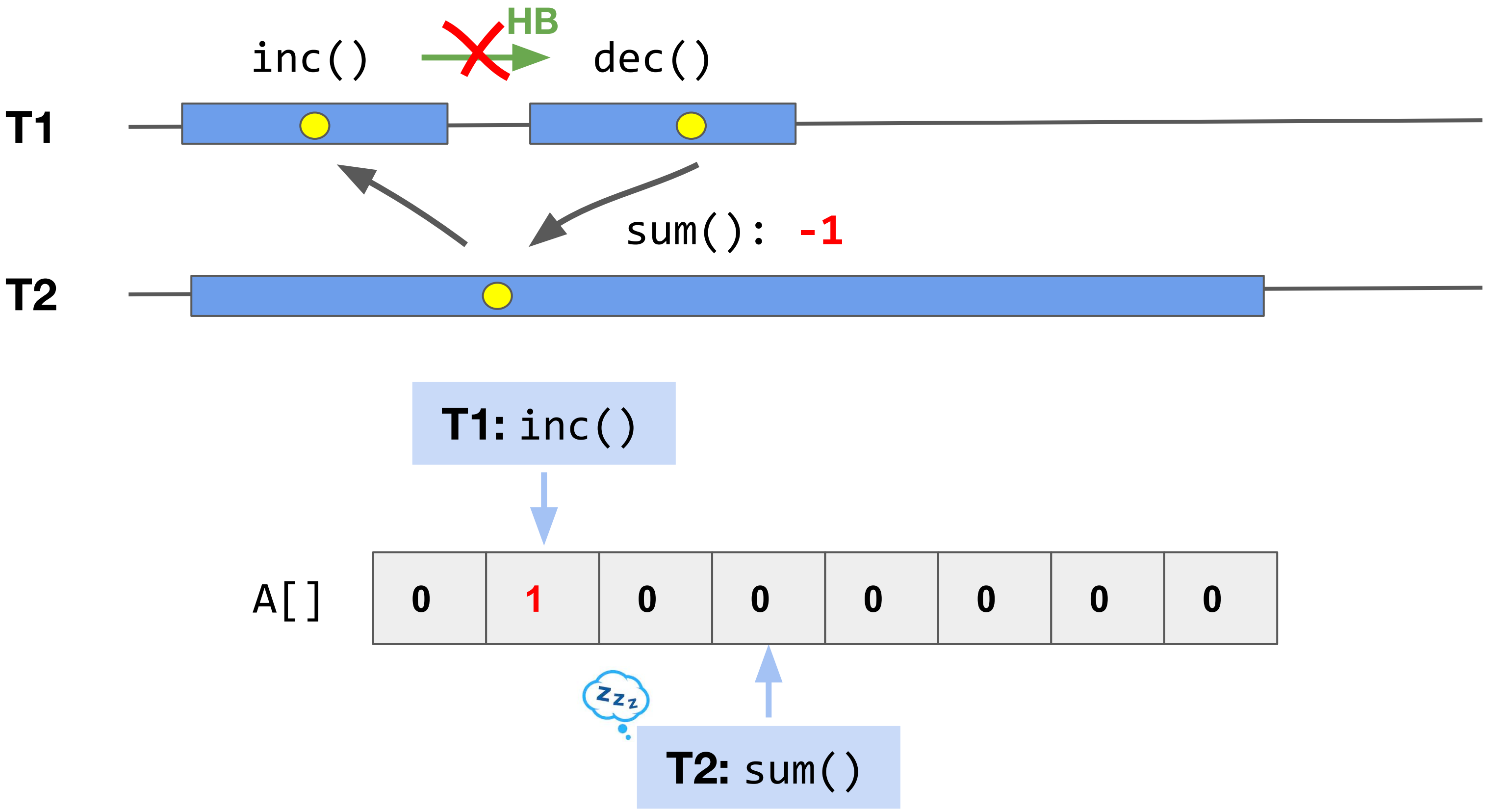
Why LongAdder is not linearizable?



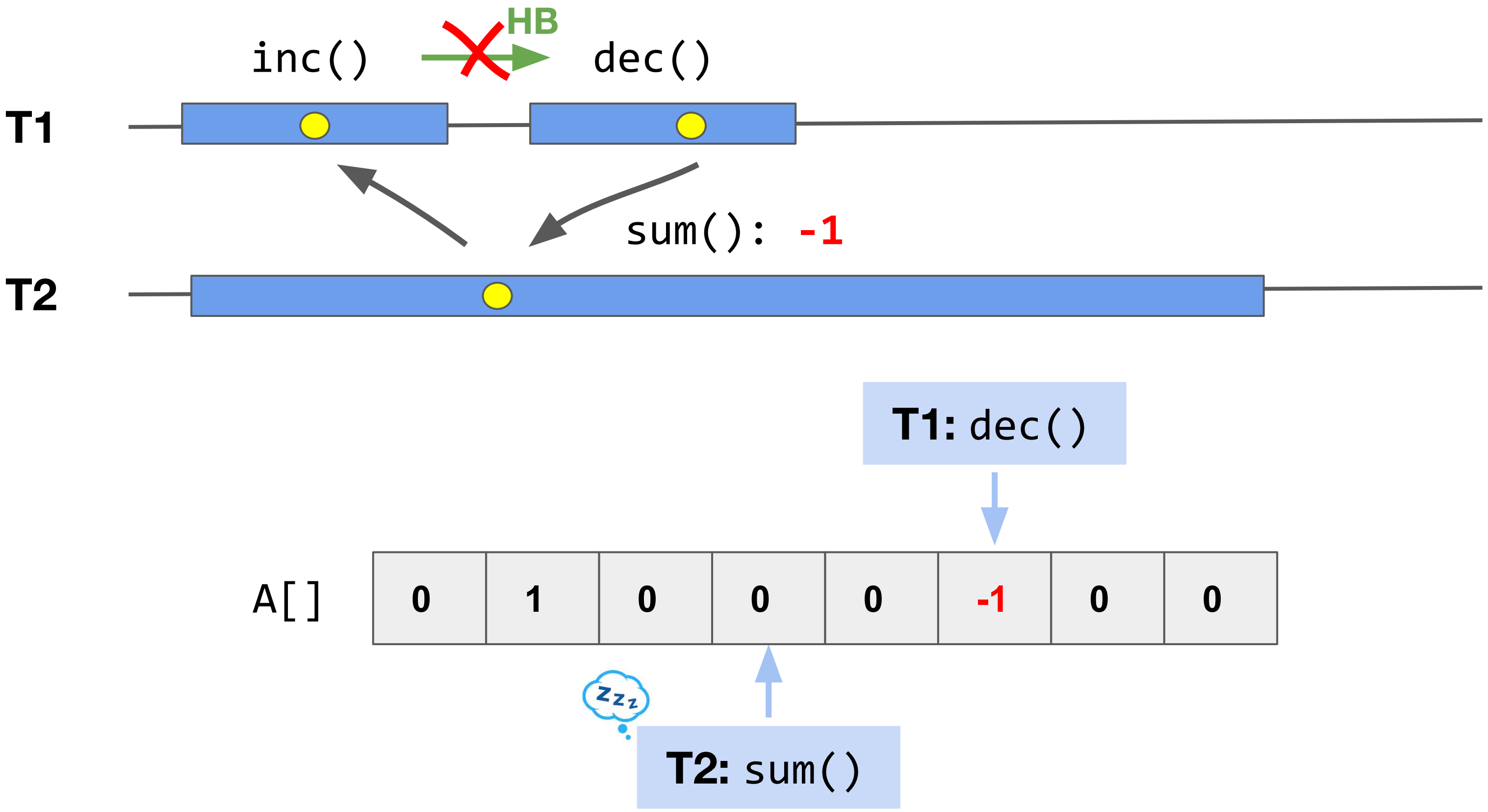
Why LongAdder is not linearizable?



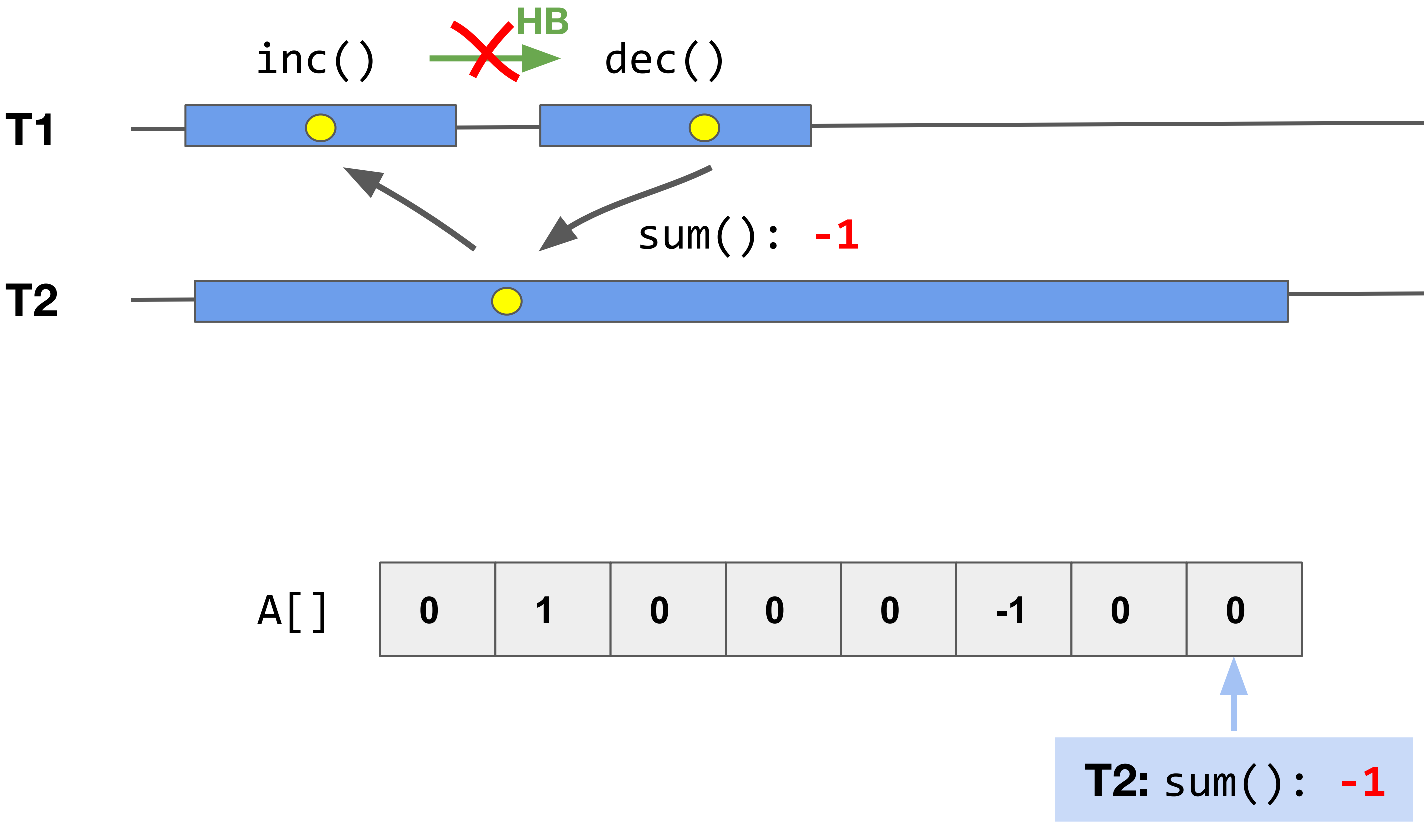
Why LongAdder is not linearizable?



Why LongAdder is not linearizable?



Why LongAdder is not linearizable?



Testing strategies

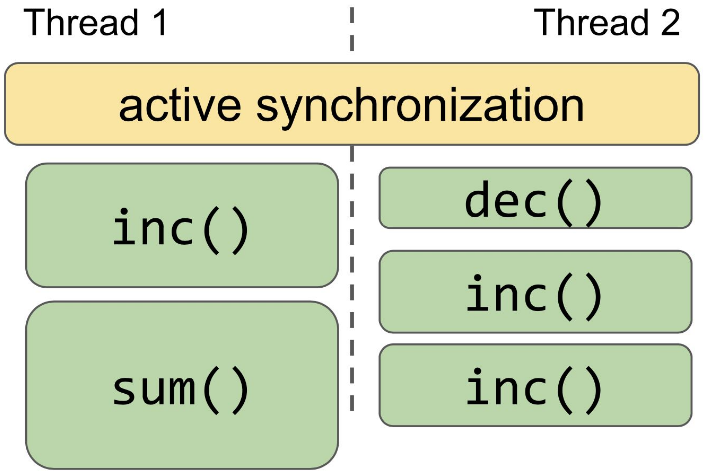
Stress testing

```
class LongAdderTest {
    ....
    @Test
    fun test() = StressOptions()
                     .check(this::class)
}
```

- Starts real threads
- Actively synchronizes them
- Execute the operation many times

Stress testing

```kotlin
class LongAdderTest {
    ....
    @Test
    fun test() = StressOptions()
                     .check(this::class)
}
```



- Starts real threads
- Actively synchronizes them
- Execute the operation many times

Stress testing

Day 3. June 17

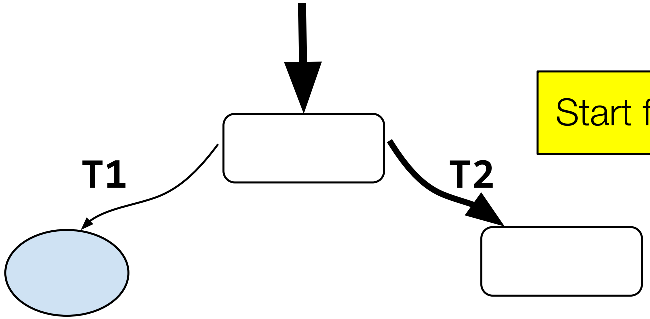
Time UTC+03:00	Lecture
18:35 **Track 1**	Workshop: Java Concurrency Stress (JCStress) **Aleksey Shipilev** *Red Hat* *#concurrency #java-memory-model #testing #JVM* EN
20:20 **Track 1**	Workshop: Java Concurrency Stress (JCStress) (part 2) **Aleksey Shipilev** *Red Hat* *#concurrency #java-memory-model #testing #JVM* EN

Model checking

```
class CounterTest {
    ....
    @Test
    fun test() = ModelCheckingOptions()
                    .check(this::class)
}
```

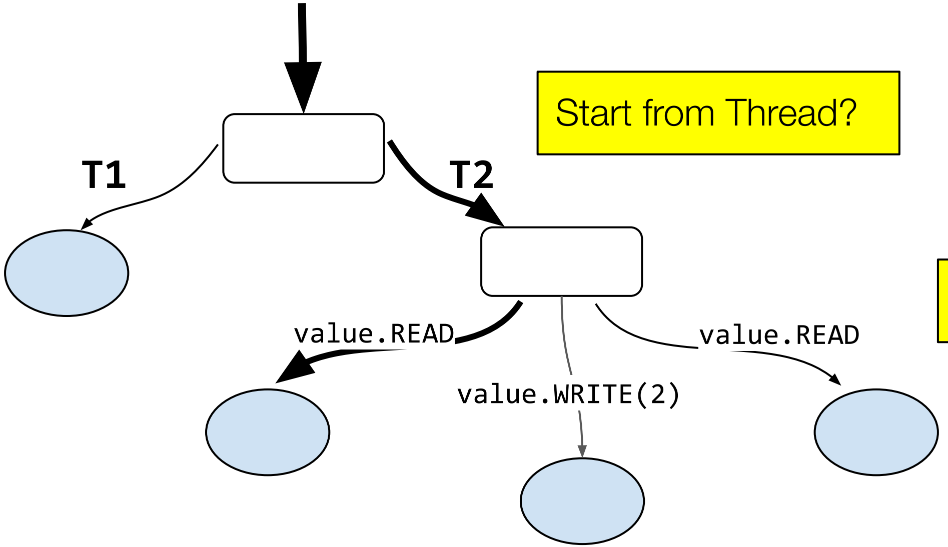
- Sequential consistency memory model
- Examines many different interleavings within a bounded number of context switches
- Provides a trace

Interleaving tree



Start from Thread?

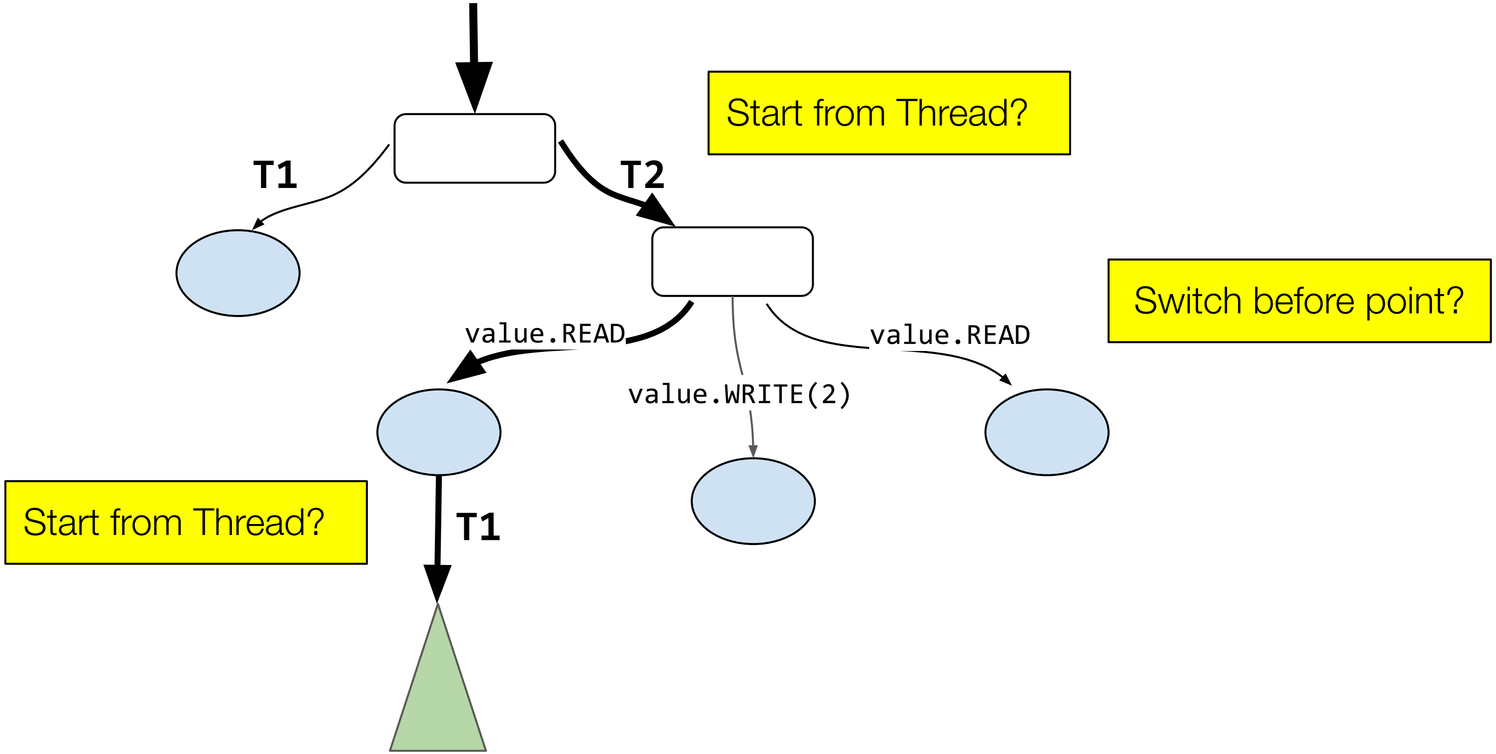
Interleaving tree



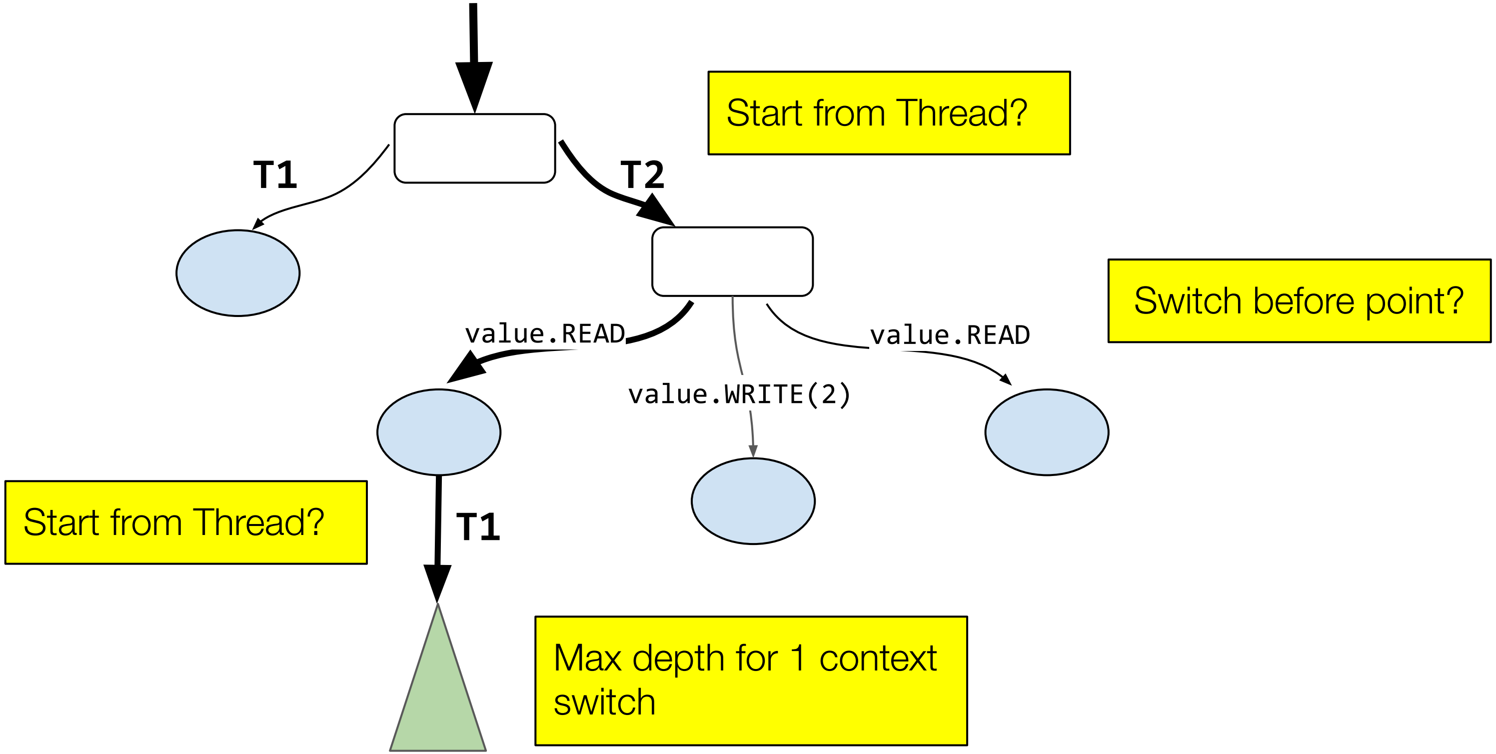
Start from Thread?

Switch before point?

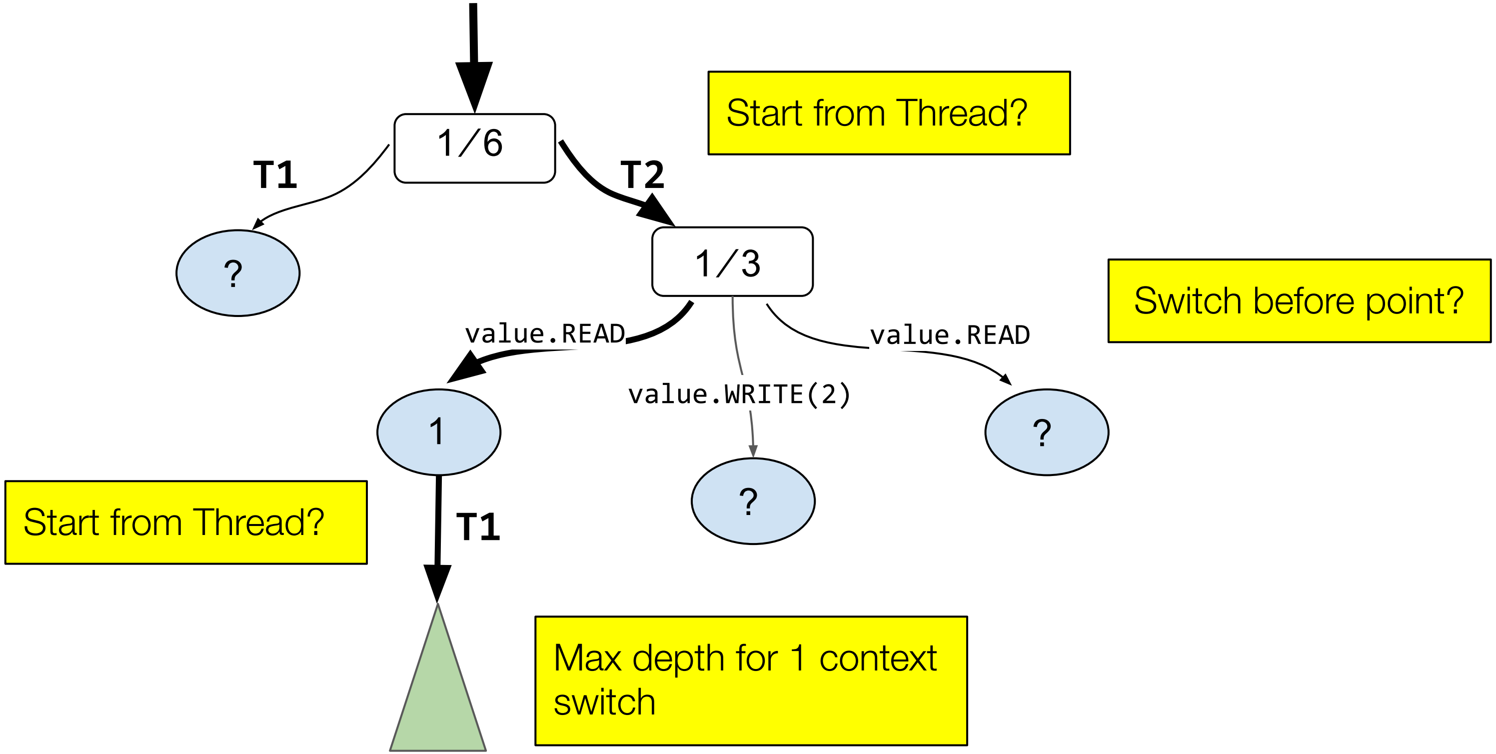
Interleaving tree



Interleaving tree



Interleaving tree



Stress vs Model checking?

Model checking:

- Bugs under sequentially consistent memory model
- Better coverage
- Trace to reproduce the error

Stress testing:

- Low-level effects bugs
- Rare bugs that require many context switches to reproduce (bounds on model checking do not allow that now)

Configurations

Scenario configuration

```
class MyQueueTest {
   val q = MyQueue<Int>()

   @Operation fun add(x: Int) = q.add(x)
   @Operation fun poll() = q.poll()

   @Test fun test() = StressOptions()
        .actorsBefore(2)
        .threads(2).actorsPerThread(3)
        .actorsAfter(1)
        .check(this::class)
}
```

Scenario configuration

```
class MyQueueTest {
   val q = MyQueue<Int>()

   @Operation fun add(x: Int) = q.add(x)
   @Operation fun poll() = q.poll()

   @Test fun test() = StressOptions()
        .actorsBefore(2)
        .threads(2).actorsPerThread(3)
        .actorsAfter(1)
        .check(this::class)
}
```

```
Init part:
[poll(), add(9)]
Parallel part:
| poll() | add(4) |
| add(3) | add(6) |
| poll() | poll() |
Post part:
[add(1)]
```

Scenario configuration

```
class MyQueueTest {
   val q = MyQueue<Int>()

   @Operation fun add(x: Int) = q.add(x)
   @Operation fun poll() = q.poll()

   @Test fun test() = StressOptions()
        .actorsBefore(2)
        .threads(2).actorsPerThread(3)
        .actorsAfter(1)
        .invocationsPerIteration(10_000) // Run each scenario 10_000 times
        .iterations(100) // Try 100 different scenarios
        .check(this::class)
}
```

Custom scenarios

```
@Test fun test() = StressOptions()
    .addCustomScenario {
        parallel {
          thread {
            actor(::add, 5)
            actor(::add, 6)
          }
          thread {
            actor(::poll)
          }
        }
    }
    .check(this::class)
```

```
Parallel part:
| add(5) | poll() |
| add(6) |        |
```

Stress test example

1. Checkout **2.1-low-level** branch:

   ```
   $ git checkout 2.1-low-level
   ```

2. WeakMemoryModel.kt

3. Run `test()`

Parameters Generation

```
class MyHashMap {
   val map = MyHashMap<Int, Int>()

   @Operation
   fun put(@Param(gen = IntGen::class,
                  conf = "0:10") key: Int, v: Int) = map.put(key, v)



   @Test fun test() = ...
}
```

We use parameter generators!

Parameters Generation

```
class MyHashMap {
   val map = MyHashMap<Int, Int>()

   @Operation
   fun put(@Param(gen = IntGen::class,
                  conf = "0:10") key: Int, v: Int) = map.put(key, v)

   @Operation fun get(@Param(gen = IntGen::class,
                             conf = "0:10") key: Int) = map.get(key)

   @Test fun test() = ...
}
```

We can share configurations

Parameters Generation

```
@Param(name = "key", gen = IntGen::class, conf = "0:10")
class MyHashMap {
   val map = MyHashMap<Int, Int>()

   @Operation
   fun put(@Param(name = "key") key: Int, v: Int) = map.put(key, v)

   @Operation fun get(@Param(name = "key") key: Int) = map.get(key)

   @Test fun test() = ...
}
```

We can share configurations

Back to LongAdder

1. Checkout **3.1-longAdder** branch:

```
$ git checkout 3.1-longAdder
```

2. Checkout **3.2-jctools** branch

Custom Parameter Generators

```
class RandomIntParameterGenerator(ignoredConf: String)
    : ParameterGenerator<Int> {
    override fun generate() = Random.nextInt()
}
```

Generated values should be Serializable

Constraints

```
class MpscQueueTest {
   val q = MpscLinkedAtomicQueue<Int>()

   @Operation fun offer(x: Int) = q.offer(x)
   @Operation fun poll() = q.poll()

   @Test fun test() = ...
}
```

```
Parallel part:
| add(2)      | add(4)           |
| poll(): 2   | poll(): null     |
```

Constraints

SC queue is incorrect for 2 consumers by the contract

```
class MpscQueueTest {
   val q = MpscLinkedAtomicQueue<Int>()

   @Operation fun offer(x: Int) = q.offer(x)
   @Operation fun poll() = q.poll()

   @Test fun test() = ...
}
```

```
Parallel part:
| add(2)     | add(4)        |
| poll(): 2  | poll(): null  |
```

Constraints

```
@OpGroupConfig(name = "consumers", nonParallel = true)
class MpscQueueTest {
   val q = MpscLinkedAtomicQueue<Int>()

   @Operation fun offer(x: Int) = q.offer(x)
   @Operation(group = "consumers") fun poll() = q.poll()

   @Test fun test() = ...
}
```

Single-consumer queue

1. Checkout **4.1-constraints** branch

Progress guarantees

An algorithm without explicit synchronization may still be blocking and checking for liveness bugs is a non-trivial task.

Progress guarantees

An algorithm without explicit synchronization may still be blocking and checking for liveness bugs is a non-trivial task.

Lincheck may test the algorithm for **obstruction-freedom** violation.

```
@Test
fun test() = ModelCheckingOptions()
                .checkObstructionFreedom()
                .check(this::class)
```

Progress guarantees

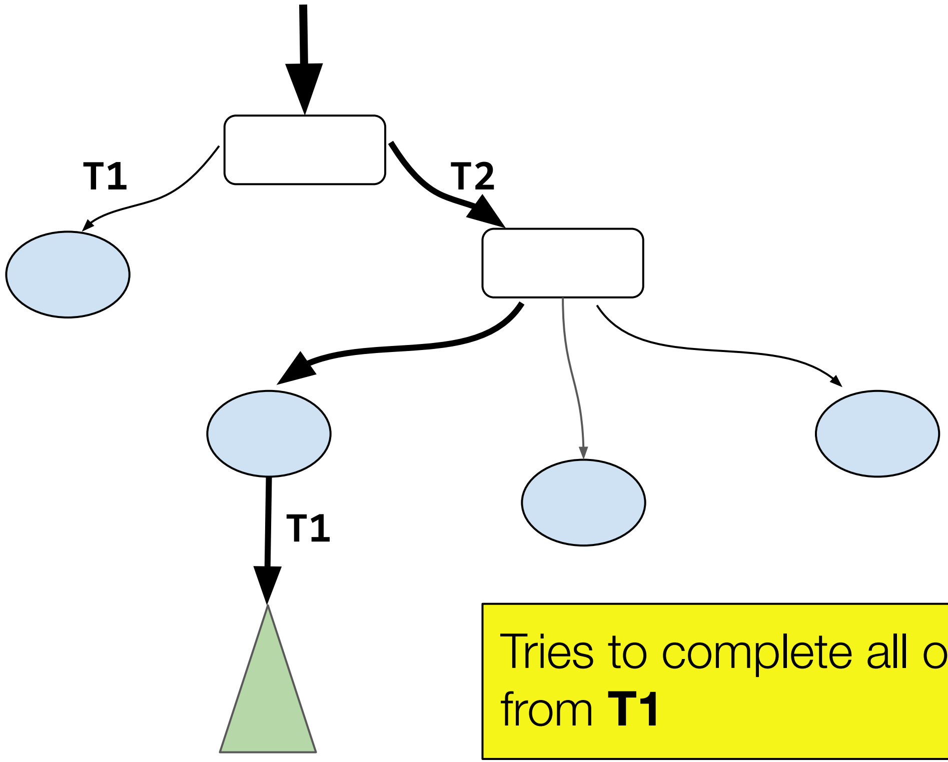
An algorithm without explicit synchronization may still be blocking and checking for liveness bugs is a non-trivial task.

Lincheck may test the algorithm for **obstruction-freedom** violation.

```
@Test
fun test() = ModelCheckingOptions()
             .checkObstructionFreedom()
             .check(this::class)
```

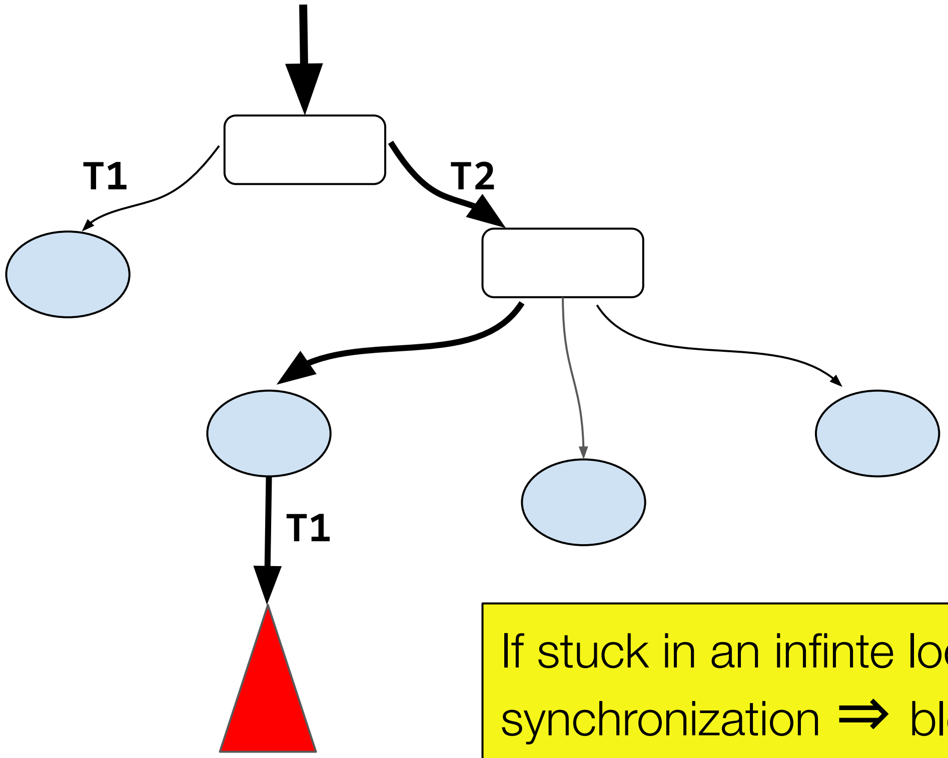
Progress guarantees are checked in model checking mode

Progress guarantees



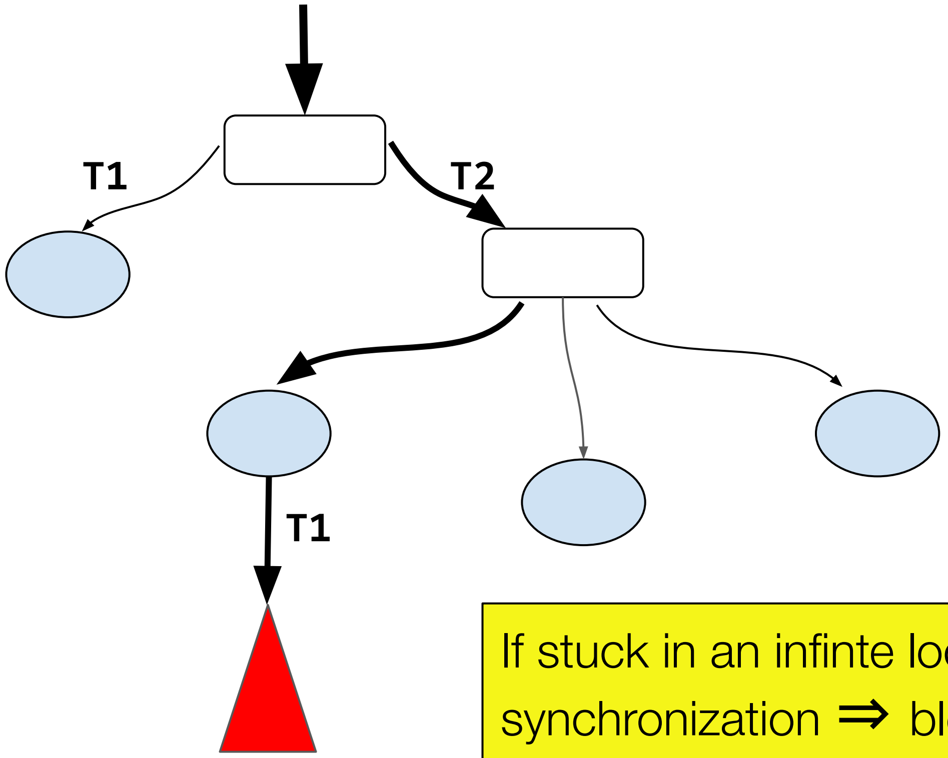
Tries to complete all operatios from **T1**

Progress guarantees



If stuck in an infinte loop or synchronization ⇒ blocking

Progress guarantees



Obstruction-freedom: any operation may be completed in a bounded number of steps if all the other processes stop.

If stuck in an infinte loop or synchronization ⇒ blocking

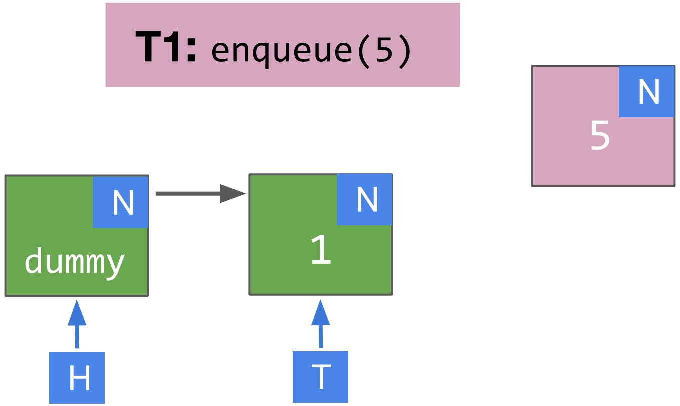
Check whether the algorithm is blocking?

1. Checkout **5.1-dataHolder** branch

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue

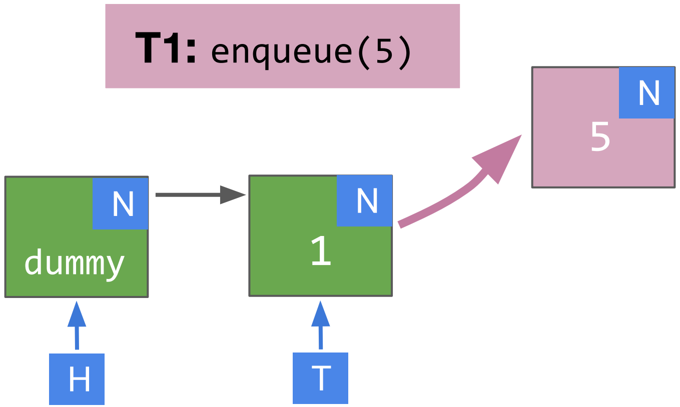



```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    }
  }
}
```

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue

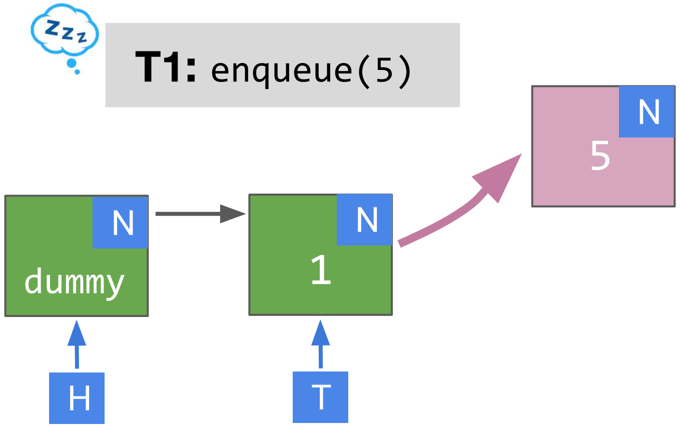



```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    }
  }
}
```

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue

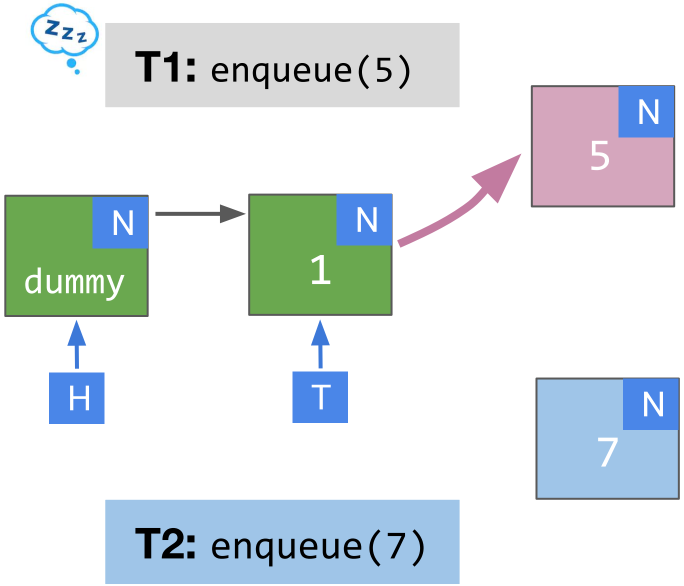



```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    }
  }
}
```

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue

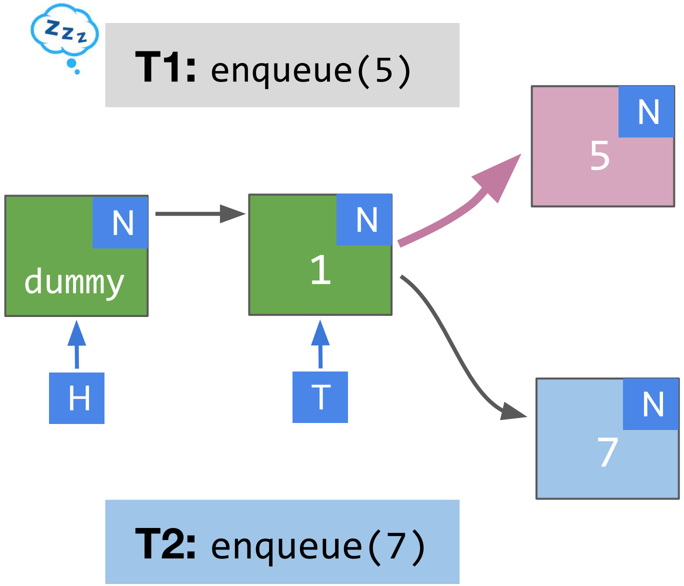



```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    }
  }
}
```

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue

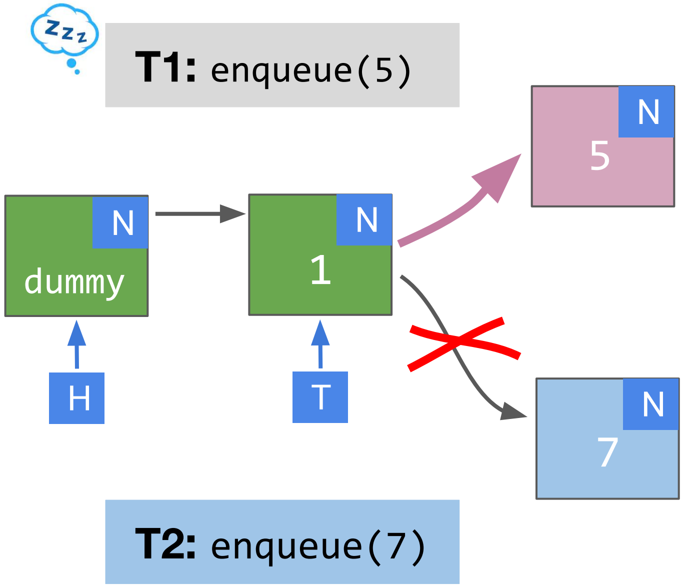



```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    }
  }
}
```

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue

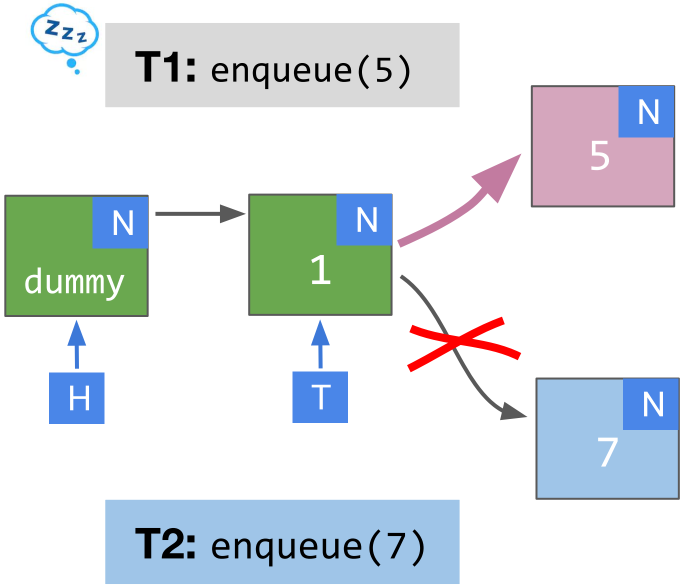



```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    }
  }
}
```

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue



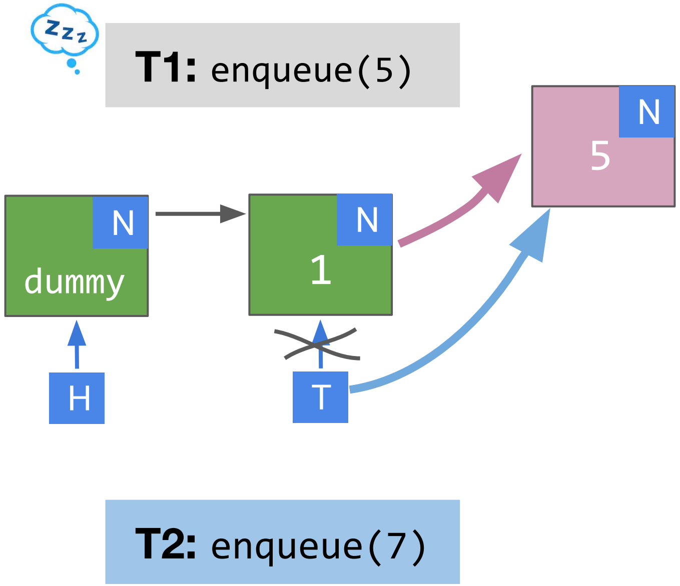

```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    }
  }
}
```

T2 can not proceed with **enqueue(7)** ⇒ blocking algorithm

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue



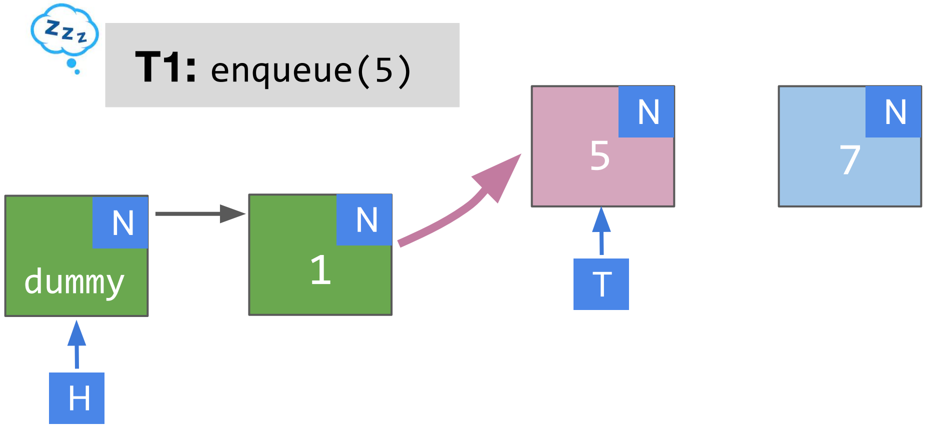

```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    } else {
      T.CAS(tail, tail.N)
    }
```

T2 should help **T1** to complete enqueue(5)

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue




```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    } else {
      T.CAS(tail, tail.N)
    }
  }
```

T2: `enqueue(7)`

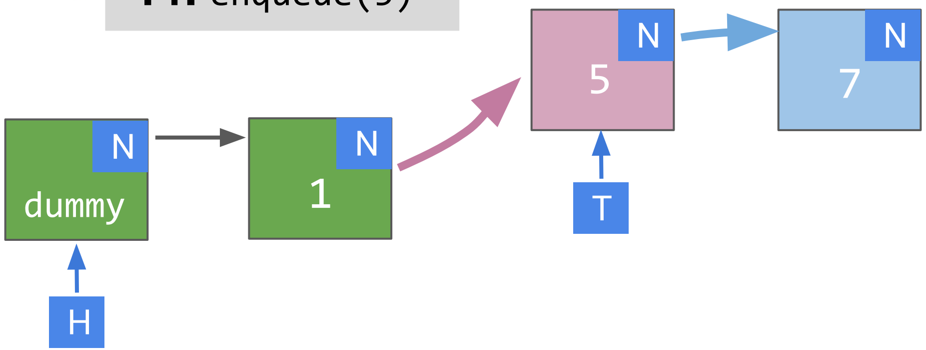
Now **T2** can proceed with `enqueue(7)`

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue

T1: `enqueue(5)`



T2: `enqueue(7)`

```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    } else {
      T.CAS(tail, tail.N)
    }
  }
}
```

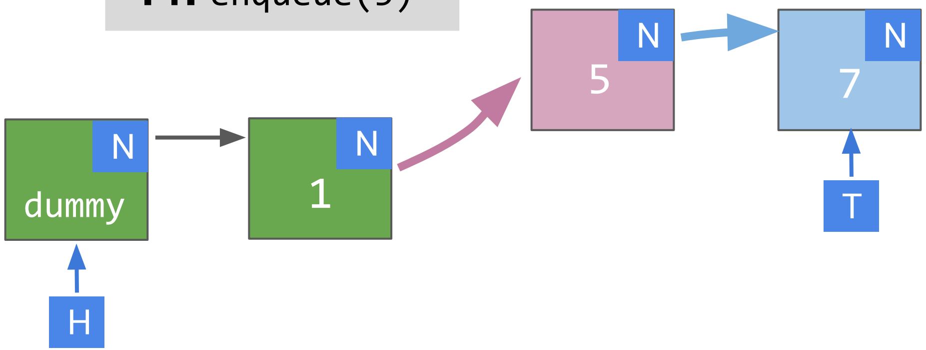
Now **T2** can proceed with `enqueue(7)`

Helping in lock-free algorithms

Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, j.u.c.ConcurrentLinkedQueue

T1: `enqueue(5)`



T2: `enqueue(7)`

```
fun enqueue(x) {
  while (true) {
    tail := T
    if (tail.N.CAS(null, x)) {
      T.CAS(tail, x)
    } else {
      T.CAS(tail, tail.N)
    }
  }
```

Now **T2** can proceed with `enqueue(7)`

Helping in lock-free algorithms

1. Checkout **5.2-msqueue** branch

Modular testing

If some linearizable data structures are used as building blocks for your algorithm, the number of all possible interleavings may be enormous.

You can treat operations of these data structures atomic ⇒ reduce the number of interleavings and speed up testing.

```
@Test
fun test() = ModelCheckingOptions()
.addGuarantee(
   forClasses(ConcurrentHashMap::class.qualifiedName!!).allMethods().treatAsAtomic()
)
.check(this::class)
```

Modular testing

1. Checkout **6.1-multimap** branch

Verification

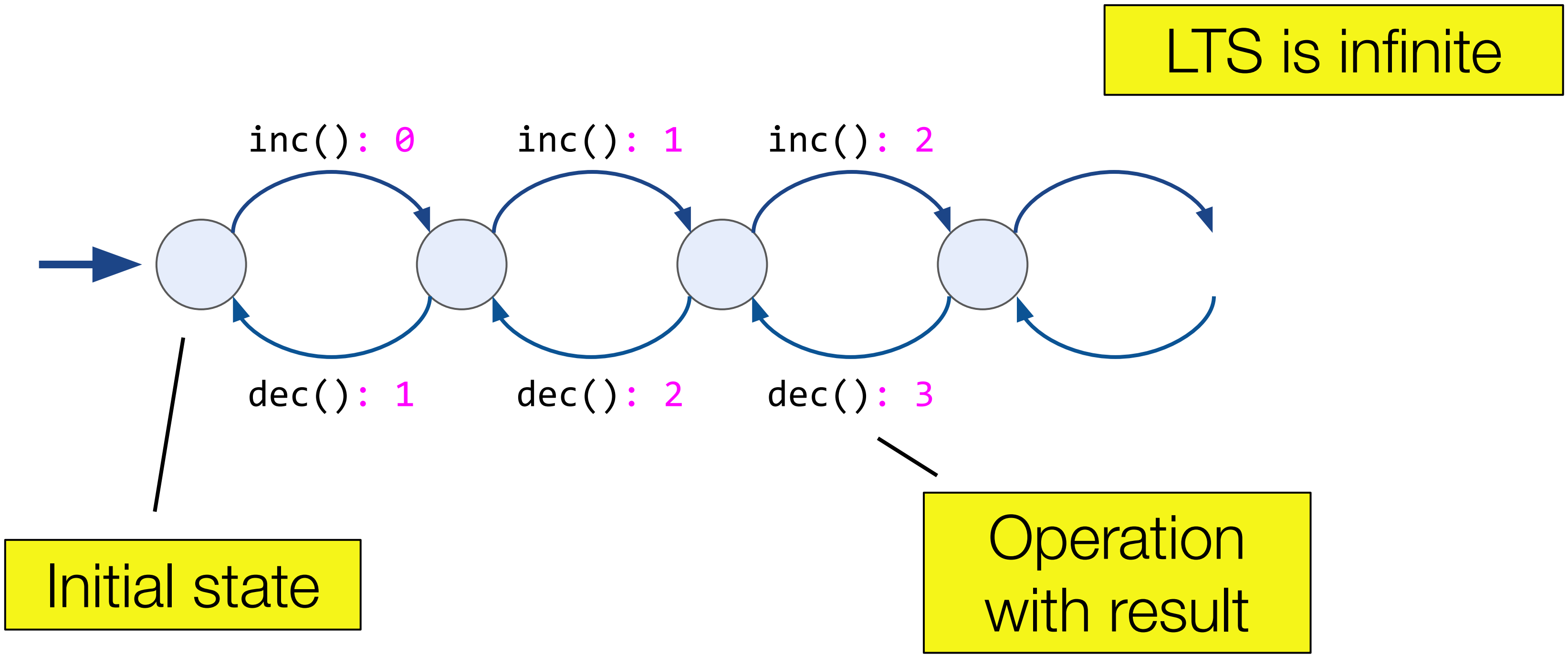
Results Verification

Simplest solution:

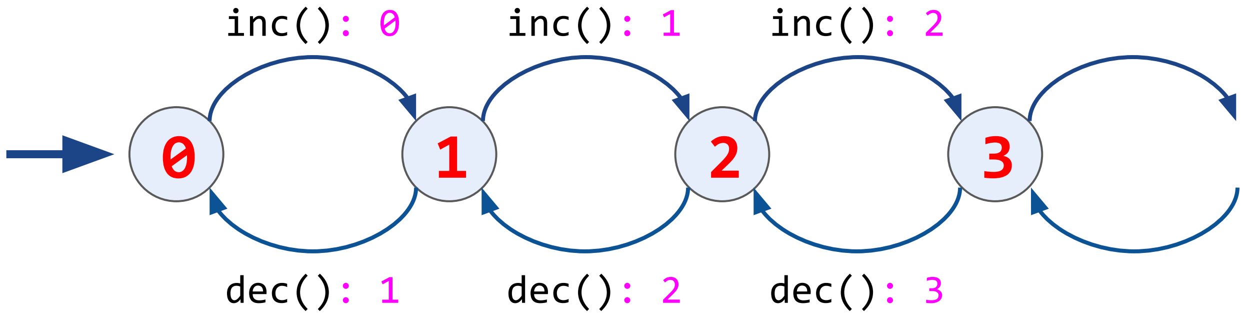
1. Generate all possible sequential histories
2. Check whether one of them produces the same results

Smarter solution: Labeled Transition System (LTS)

LTS (Labeled Transition System)



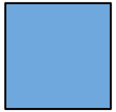
LTS (Labeled Transition System)



LTS-based verification

```
val q = MSQueue<Int>()
```

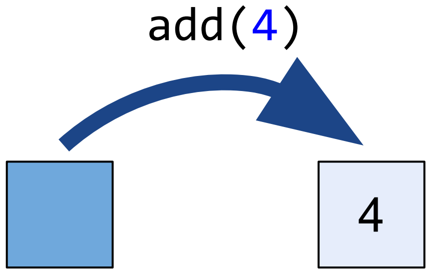
`q.add(4)`	`q.add(9)`
`q.poll(): 9`	`q.poll(): 4`



LTS-based verification

```
val q = MSQueue<Int>()
```

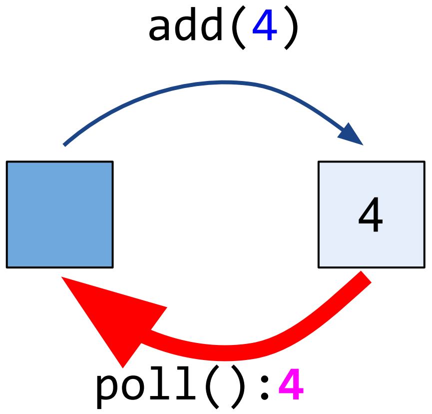
`q.add(4)`	
`q.poll(): 9`	`q.add(9)`
	`q.poll(): 4`



LTS-based verification

```
val q = MSQueue<Int>()
```

`q.add(4)`	
`q.poll(): 9`	
	`q.add(9)`
	`q.poll(): 4`

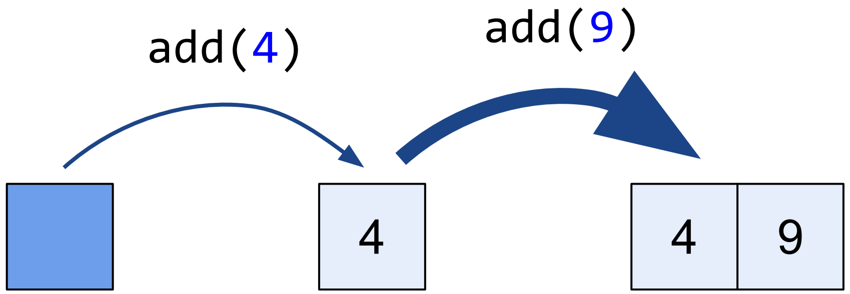


Result is different

LTS-based verification

```
val q = MSQueue<Int>()
```

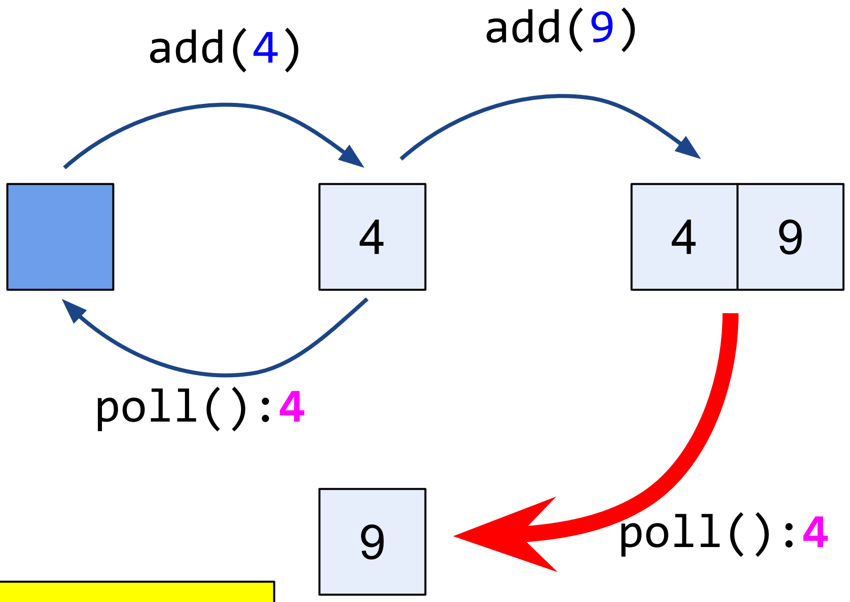
q.add(4)	
	q.add(9)
q.poll(): 9	q.poll(): 4



LTS-based verification

```
val q = MSQueue<Int>()
```

q.add(4)	
	q.add(9)
q.poll(): 9	
	q.poll(): 4

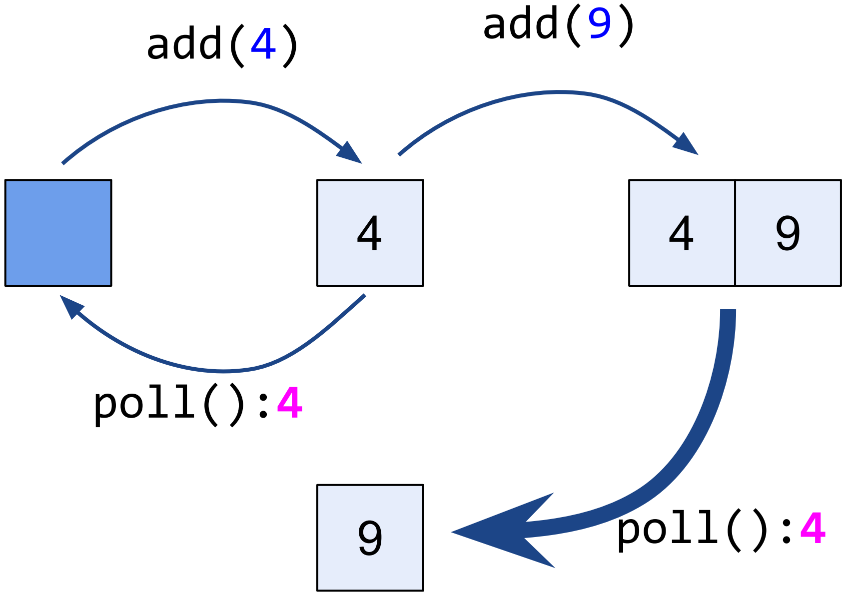


Result is different

LTS-based verification

```
val q = MSQueue<Int>()
```

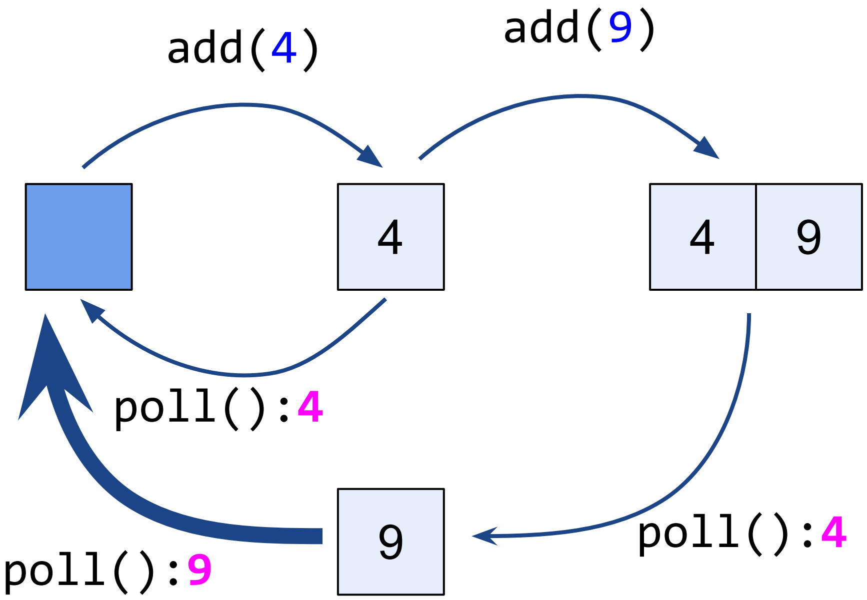
q.add(4)	
	q.add(9)
	q.poll(): 4
q.poll(): 9	



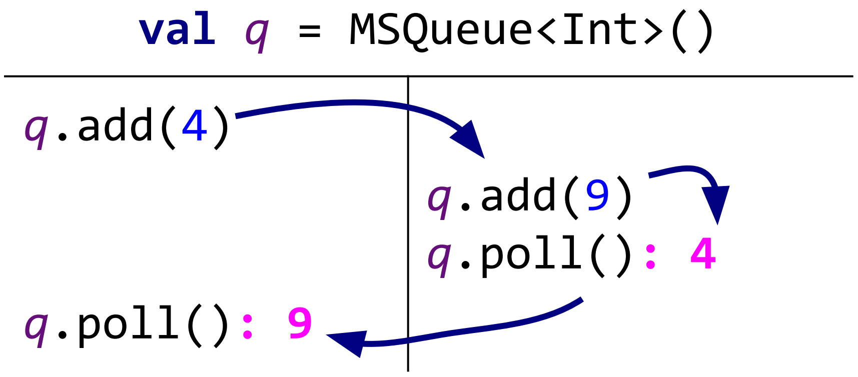
LTS-based verification

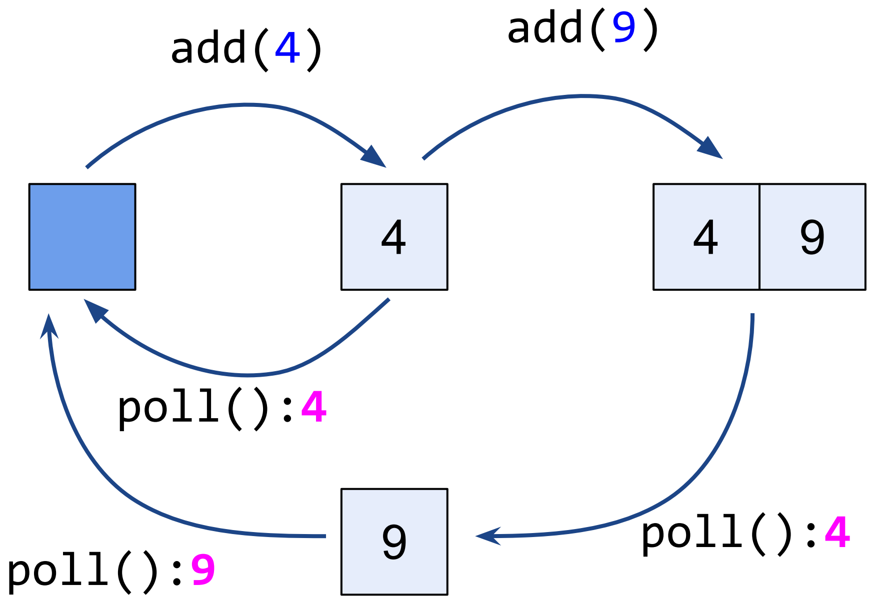
```
val q = MSQueue<Int>()
```

q.add(4)	
	q.add(9)
	q.poll(): 4
q.poll(): 9	



LTS-based verification





Path is found ⇒ correct

State equivalency

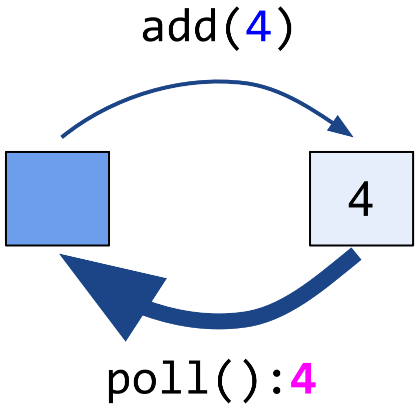
Warning: To make verification faster, you can specify the state equivalence relation on your sequential specification. ….

Till this moment you got this warning for all tests

State equivalency

Warning: To make verification faster, you can specify the state equivalence relation on your sequential specification. ….

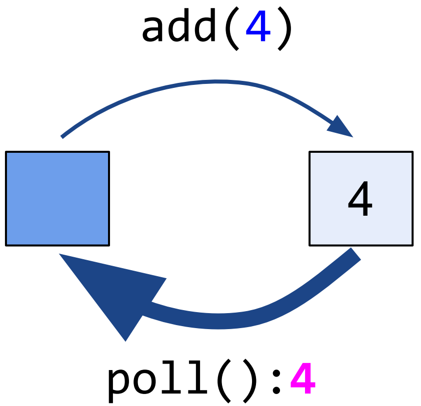
We can set state equivalency
relation via `equals/hashCode()`:



State equivalency

Warning: To make verification faster, you can specify the state equivalence relation on your sequential specification. ….

We can set state equivalency relation via `equals/hashCode()`:

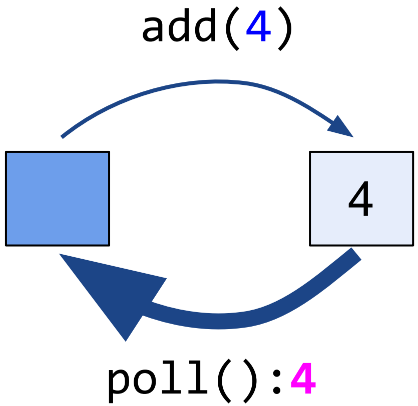



```
class MSQueueTest {
    val q = MSQueue<Int>()

    // Operations here

    override fun equals(other: Any?) = ...
    override fun hashCode() = ...
}
```

State equivalency

Warning: To make verification faster, you can specify the state equivalence relation on your sequential specification. ….

We can set state equivalency relation via `equals/hashCode()`:




```
class MSQueueTest : VerifierState() {
    val q = MSQueue<Int>()

    // Operations here

    override fun generateState() = q
}
```

Sequential specification

```
class MySuperFastQueueTest {
   val q = MySuperFastQueue<Int>()

   @Operation fun offer(x: Int) = q.offer(x)
   @Operation fun poll() = q.poll()

   @Test fun test() = StressOptions()
      .sequentialSpecification(SequentialQueue::class.java)
      .check(this::class)
}
```

```
class SequentialQueue : VerifierState() {
   val q = LinkedList<Int>()

   fun offer(x: Int) { q.poll(x) }
   fun poll() = q.poll()

   @Override fun generateState() = q
}
```

Verification

1. Checkout **7.1-verification** branch

Blocking data structures

Lincheck supports testing blocking data structures implemented via suspending functions from Kotlin language.

The examples of such data structures from the Kotlin Coroutines library: Channel, Mutex, Flow..

See the corotuines guide for details.

Rendezvous channel

```
class Channel<T> {
    suspend fun send(x: T)
    suspend fun receive()
}
```

```
val c = Channel<Int>()
```

`c.send(4)`	`c.receive() // 4`

Rendezvous channel

```
class Channel<T> {
    suspend fun send(x: T)
    suspend fun receive()
}
```

```
val c = Channel<Int>()
```

c.send(4)	c.receive() // 4

send waits for receive and vice versa

Rendezvous channel

Client 1

```
val task = Task(...)
tasks.send(task)
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

Worker

```
while(true) {
    val task = tasks.receive()
    processTask(task)
}
```

1

Have to wait for send

```
val tasks = Channel<Task>()
```

Rendezvous channel

Client 1

```
val task = Task(...)
tasks.send(task)
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

Worker


```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

1

```
val tasks = Channel<Task>()
```

Rendezvous channel

Client 1

```
val task = Task(...)
tasks.send(task)
```

Worker

```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

1

Client 2

```
val task = Task(...)
tasks.send(task)
```

```
val tasks = Channel<Task>()
```

Rendezvous channel

Rendezvous!

Client 1

```
    val task = Task(...)
(2) tasks.send(task)
```

Worker

```
    while(true) {
(1)     val task = tasks.receive()
        processTask(task)
    }
```

Client 2

```
    val task = Task(...)
    tasks.send(task)
```

```
val tasks = Channel<Task>()
```

Rendezvous channel

Client 1

```
val task = Task(...)
② tasks.send(task)
```

Worker

```
while(true) {
①   val task = tasks.receive()
③   processTask(task)
}
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

```
val tasks = Channel<Task>()
```

Rendezvous channel

Client 1

```
  val task = Task(...)
2 tasks.send(task)
```

Worker

```
  while(true) {
1   val task = tasks.receive()
3   processTask(task)
  }
```

Client 2

```
  val task = Task(...)
4 tasks.send(task)
```

Have to wait for receive

```
val tasks = Channel<Task>()
```

Rendezvous channel

Client 1

```
  val task = Task(...)
2 tasks.send(task)
```

Worker

```
  while(true) {
1   val task = tasks.receive()
3   processTask(task)
  }
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

```
val tasks = Channel<Task>()
```

Rendezvous channel

Client 1

```
    val task = Task(...)
(2) tasks.send(task)
```

Worker

```
        while(true) {
(5) (1)     val task = tasks.receive()
    (3)     processTask(task)
        }
```

Client 2

```
    val task = Task(...)
(4) tasks.send(task)
```

Rendezvous!

```
val tasks = Channel<Task>()
```

Rendezvous channel

```
val c = Channel<Int>()
```

`c.send(4)`	`c.receive() // S + 4`

Non-linearizable
because of suspension

```
val c = Channel<Int>()
```

```
c.receive(): // S + 4
```

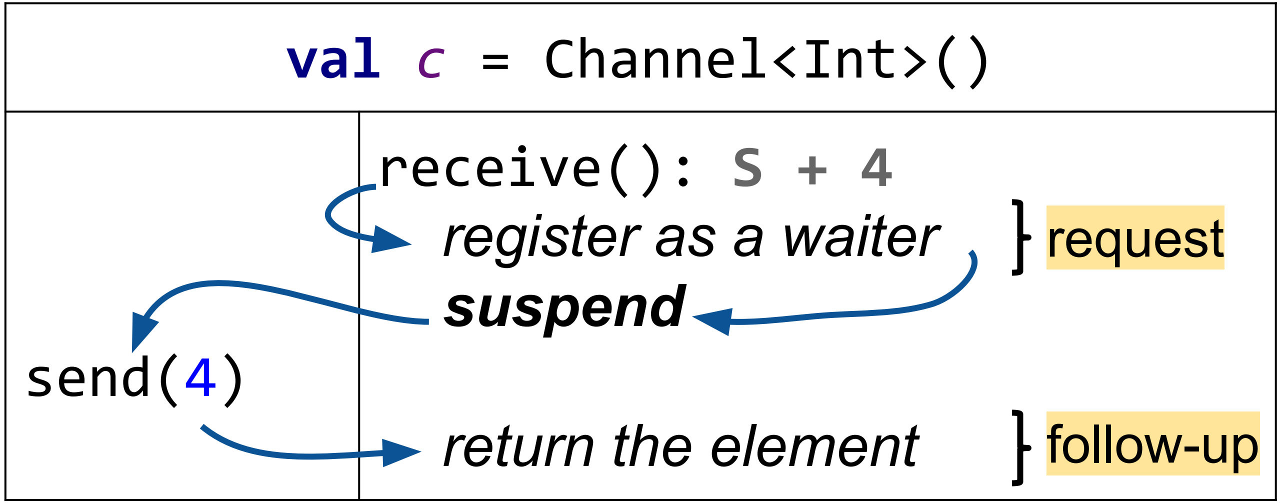
register as a waiter

suspend

```
c.send(4)
```

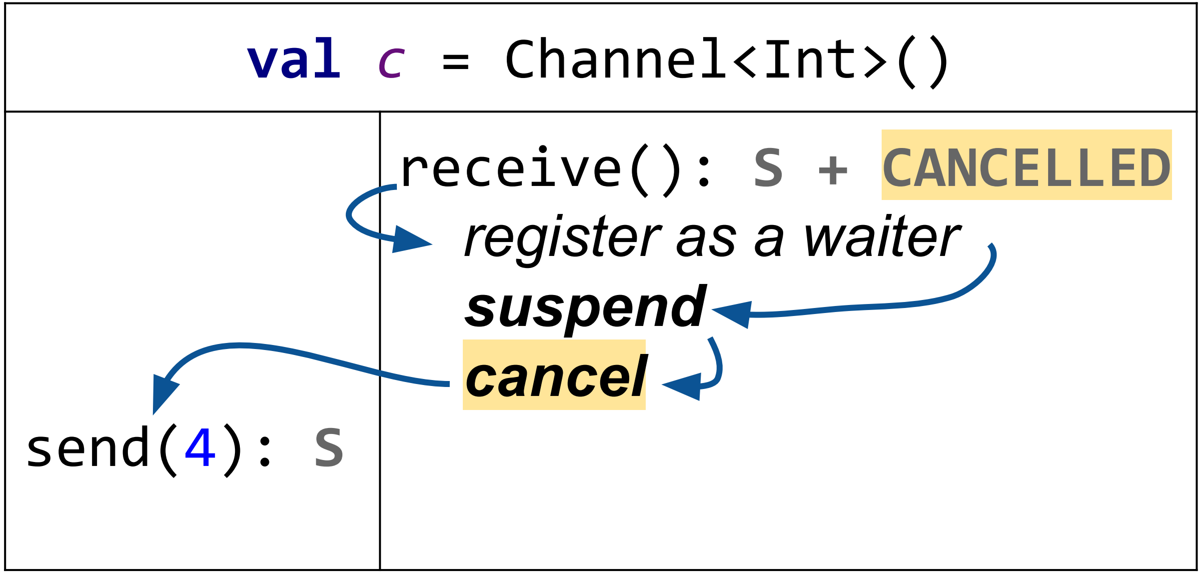
return element

Dual Data Structures*



* “Nonblocking Concurrent Data Structures with Condition Synchronization” by Scherer, W.N. and Scott, M.L.

Dual Data Structures*



* “Nonblocking Concurrent Data Structures with Condition Synchronization” by Scherer, W.N. and Scott, M.L.

Rendezvous channel test

```
class Channel<T> {
    suspend fun send(x: T)
    suspend fun receive()
}
```

```
class ChannelTest {
    val ch = Channel<Int>()

    @Operation
    suspend fun send(x: Int) = ch.send(x)
    @Operation
    suspend fun receive() = ch.receive()

    @Test fun test() = StressOptions()
                           .check(this::class)
}
```

Buffered Channels

Client 1

```
val task = Task(...)
tasks.send(task)
```

Worker

```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

One element can be sent without suspension

```
val tasks = Channel<Task>(capacity = 1)
```

Buffered Channels

Client 1

```
    val task = Task(...)
(1) tasks.send(task)
```

Does not suspend!

Client 2

```
    val task = Task(...)
    tasks.send(task)
```

Worker

```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

```
val tasks = Channel<Task>(capacity = 1)
```

Buffered Channels

Client 1

```
val task = Task(...)
tasks.send(task)
```

1

Worker

```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

2

The buffer is full, suspends

```
val tasks = Channel<Task>(capacity = 1)
```

Buffered Channels

Client 1

```
    val task = Task(...)
(1) tasks.send(task)
```

Worker

```
    while(true) {
(3)   val task = tasks.receive()
      processTask(task)
    }
```

Client 2

```
    val task = Task(...)
(2) tasks.send(task)
```

Receives the buffered element,
resumes the 2nd client,
and moves its task to the buffer

```
val tasks = Channel<Task>(capacity = 1)
```

Buffered Channels

Client 1

```
    val task = Task(...)
(1) tasks.send(task)
```

Client 2

```
    val task = Task(...)
(2) tasks.send(task)
```

Worker

```
        while(true) {
(4) (3)   val task = tasks.receive()
          processTask(task)
        }
```

Retrieves the 2nd task,
no waiters to resume

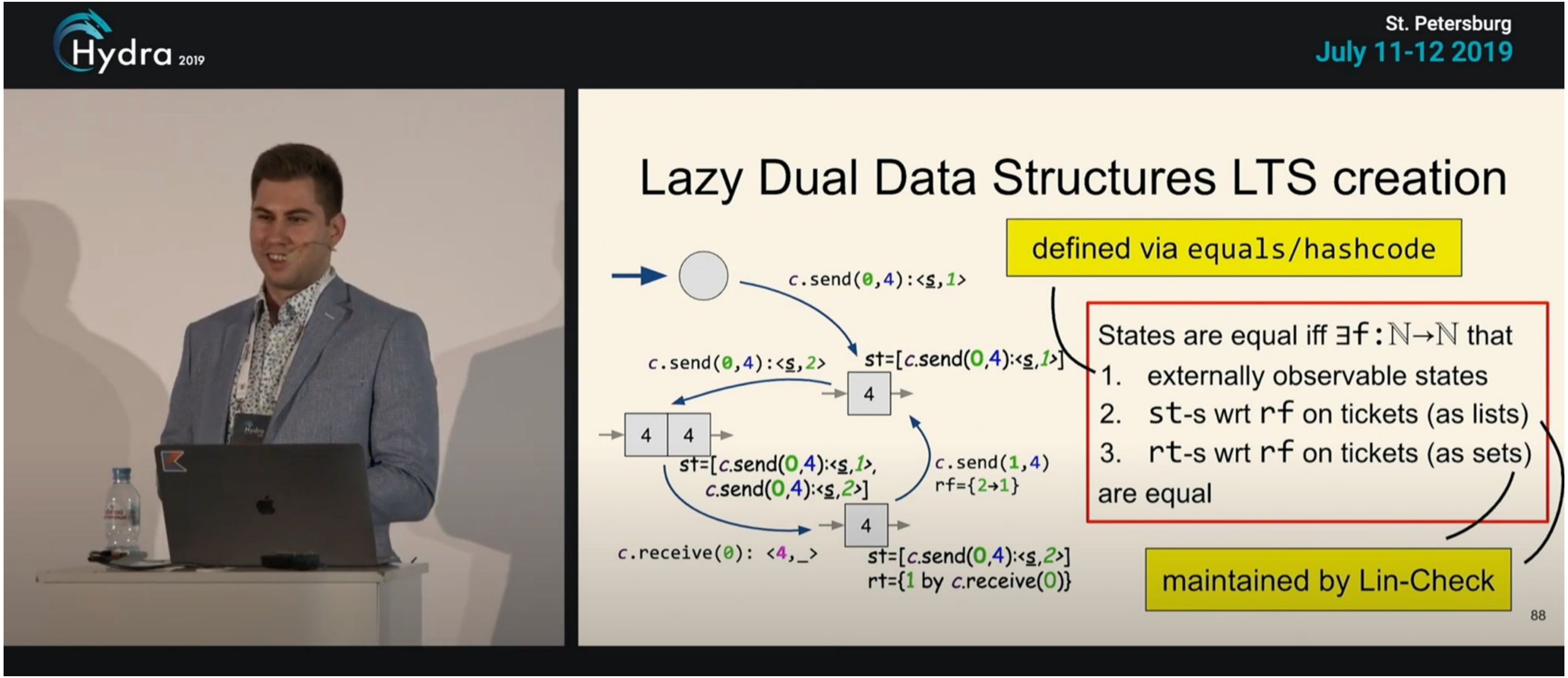
```
val tasks = Channel<Task>(capacity = 1)
```

Blocking data structures

1. Checkout **8.1-rendezvous-channel** branch

2. Checkout **8.2-buffeed-channel** branch

For deep dive into verification of blocking data structures: [“LinCheck: Testing concurrent data structures in Java”] by Nikita Koval @ Hydra 2019



Questions?