# Testing concurrent algorithms with *lincheck*

Nikita Koval, Joker 2019

# Writing concurrent code is pain

Writing concurrent code is pain

… testing it is not much easier!

```
var i = 0
```

| | |
|---|---|
| `i.inc()` | `i.inc()` |

```
var i = 0
```

| `i.inc() // 0` | `i.inc() // 1` |
| --- | --- |
| `// 1` | `// 0` |

```
var i = 0
```

| `i.inc() // 0` | `i.inc() // 0` |
| --- | --- |

```
var i = 0
```

| i.inc() // 0 | i.inc() // 0 |
| --- | --- |

We do not expect this!

Execution ***is linearizable*** ⇔ ∃ equivalent *sequential* execution wrt *happens-before* order (a bit more complicated)

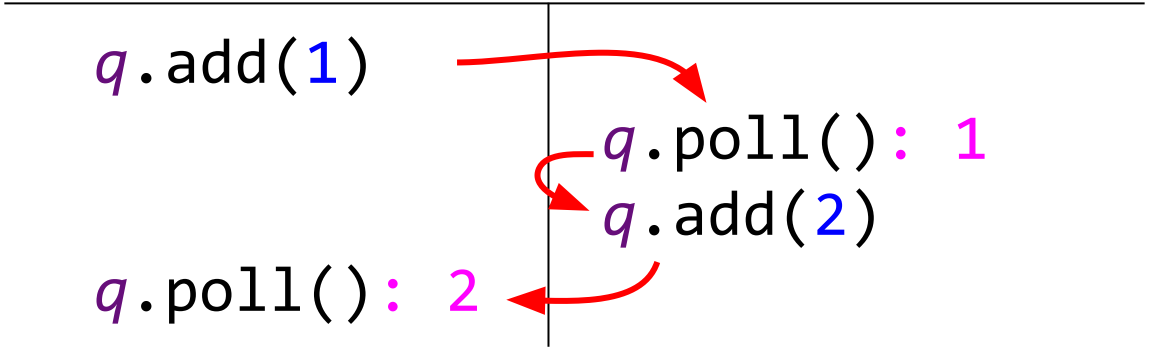
Execution ***is linearizable*** ⇔ ∃ equivalent *sequential* execution wrt *happens-before* order (a bit more complicated)

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

| | |
|---|---|
| `q.add(1)` | `q.poll(): 1` |
| `q.poll(): 2` | `q.add(2)` |

Execution ***is linearizable*** ⇔ ∃ equivalent *sequential* execution wrt *happens-before* order (a bit more complicated)

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



```
var i = 0
```

| | |
|---|---|
| `i.inc() // 0` | `i.inc() // 0` |

This counter is not linearizable

# How to check whether my data structure is linearizable?

# How to check whether my data structure is linearizable?

Formal proofs

# How to check whether my data structure is linearizable?

Formal proofs

Model checking

# How to check whether my data structure is linearizable?

Formal proofs

Testing

Model checking

# How to check whether my data structure is linearizable?

Formal proofs

**Testing**

Model checking

# How does the ideal test look?

# How does the ideal test look?

```
class ConcurrentQueueTest {
    val q = ConcurrentQueue<Int>()



}
```

Initial state

# How does the ideal test look?

```
class ConcurrentQueueTest {
    val q = ConcurrentQueue<Int>()

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

    @Operation fun poll() = q.poll()

}
```

Operations on the data structure

# How does the ideal test look?

```
class ConcurrentQueueTest {
    val q = ConcurrentQueue<Int>()

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

    @Operation fun poll() = q.poll()


}
```

Operation parameters can be non-fixed!

# How does the ideal test look?

```
class ConcurrentQueueTest {
    val q = ConcurrentQueue<Int>()

    @Operation fun          nt) = q.add(x)

    @Operation fun poll() = q.poll()

    @Test fun runTest() = LinChecker.check(this::class)
}
```

JUnit

# How does the ideal test look?

```
class ConcurrentQueueTest {
    val q = ConcurrentQueue<Int>()

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

    @Operation fun poll() = q.poll()

    @Test fun runTest() = LinChecker.check(this::class)
}
```

**The Magic Button**

# Lincheck Overview

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

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

# Lincheck Overview

*Lincheck* = **Lin**earizability **Check**er (supports not only linearizability)
https://github.com/Kotlin/kotlinx-lincheck

1. Generates a random scenario
2. Executes it a lot of times
3. Verifies the results

# Invalid Execution Example

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

# Invalid Execution Example

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

How to understand the error cause?

# Failed Scenario Minimization

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

→

```
Init part:
[add(9)]
Parallel part:
| poll(): null | add(4) |
```

*Lincheck* tries to remove actors iteratively
see `Options.minimizeFailedScenario(..)`

# How to generate scenarios?

# Scenario Configuration

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

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

    @Operation fun poll() =
        q.poll()
}
```

# Scenario Configuration

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

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

    @Operation fun poll() =
        q.poll()
}
```

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

# Scenario Configuration

```
@StressCTest(actorsBefore = 2,
             threads = 2, actorsPerThread = 3,
             actorsAfter = 1)
class MySuperFastQueueTest {
  val q = MySuperFastQueue<Int>()

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

  @Operation fun poll() =
      q.poll()
}
```

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

# Scenario Configuration

```
@StressCTest(actorsBefore = 2,
             threads = 2, actorsPerThread = 3,
             actorsAfter = 1)
class MySuperFastQueueTest {
   val q = MySuperFastQueue<Int>()

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

   @Test fun test() =
       LinChecker.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 MySuperFastQueueTest {
    val q = MySuperFastQueue<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)]
```

# Parameters Generation

We use parameter generators!

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

    @Operation fun add(@Param(gen = IntGen::class,
                              conf = "-10:10") x: Int) = q.add(x)

    @Operation fun poll() = q.poll()

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

# Parameters Generation

Let's add one more add-like method

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

    @Operation fun        add(@Param(gen = IntGen::class,
                              conf = "-10:10") x: Int) = q.add(x)

    @Operation fun addIfEmpty(@Param(gen = IntGen::class,
                              conf = "-10:10") x: Int) = q.addIfEmpty(x)

    @Operation fun poll() = q.poll()

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

# Parameters Generation

```kotlin
@Param(name = "elem", gen = IntGen::class, conf = "-10:10")
class MySuperFastQueueTest {
   val q = MySuperFastQueue<Int>()

   @Operation fun add(@Param(name="elem") x: Int) = q.add(x)
   @Operation fun addIfEmpty(@Param(name="elem") x: Int) =
                  q.addIfEmpty(x)
   @Operation fun poll() = q.poll()

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

We can share the configuration!

# Custom Parameter Generators

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

It is very simple to write your own ones!

# Custom Parameter Generators

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

Be careful, the running code can be loaded by another `ClassLoader`!

It is very simple to write your own ones!

# Constraints

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

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

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

# Constraints

What if we can invoke "close" only once by the queue contract?

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

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

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

# Constraints

What if we can invoke "close" only once by the queue contract?

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

    @Operation fun add(x: Int) = q.addIfNotClosed(x)
    @Operation fun poll() = q.poll()
    @Operation(runOnce = true) fun close() = q.close()

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

# Constraints

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

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

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

# Constraints

SC queue with two concurrent consumers is incorrect, what a surprise!

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

    @Operation fun add(x: Int) = q.add(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 MySuperFastQueueTest {
    val q = SingleConsumerTaskQueue<Int>()

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

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

# Constraints

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

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

    @Operation(group = "consumers") fun poll() = q.poll()
    @Operation(group = "consumers") fun poll(timeout: Long) = ...

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

# Number of Scenarios to Generate

```
@StressCTest(iterations = 100500)
class MySuperFastQueueTest {
    ...
    @Test fun test() =
        LinChecker.check(this::class)
}
```

```
class MySuperFastQueueTest {
    ...
    @Test fun test() = StressOptions()
        .iterations(100500)
        .check(this::class)
}
```

# Custom Scenarios

```
val s = scenario {
  initial {
    actor(MyQueueTest::add, 1)
  }
  parallel {
    thread {
      actor(MyQueueTest::add, 2)
      actor(MyQueueTest::add, 3)
    }
    thread {
      actor(MyQueueTest::poll)
      actor(MyQueueTest::poll)
    }
  }
}
```

github.com/Kotlin/kotlinx-lincheck/pull/8

# Custom Scenarios

```kotlin
val s = scenario {
  initial {
    actor(MyQueueTest::add, 1)
  }
  parallel {
    thread {
      actor(MyQueueTest::add, 2)
      actor(MyQueueTest::add, 3)
    }
    thread {
      actor(MyQueueTest::poll)
      actor(MyQueueTest::poll)
    }
  }
}
```

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

github.com/Kotlin/kotlinx-lincheck/pull/8

# Custom Scenarios

```
val s = scenario {
  initial {
    actor(MyQueueTest::add, 1)
  }
  parallel {
    thread {
      actor(MyQueueTest::add, 2)
      actor(MyQueueTest::add, 3)
    }
    thread {
      actor(MyQueueTest::poll)
      actor(MyQueueTest::poll)
    }
  }
}
```

Be careful, the running code can be loaded by another `ClassLoader`!

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

github.com/Kotlin/kotlinx-lincheck/pull/8

# How to run scenarios?

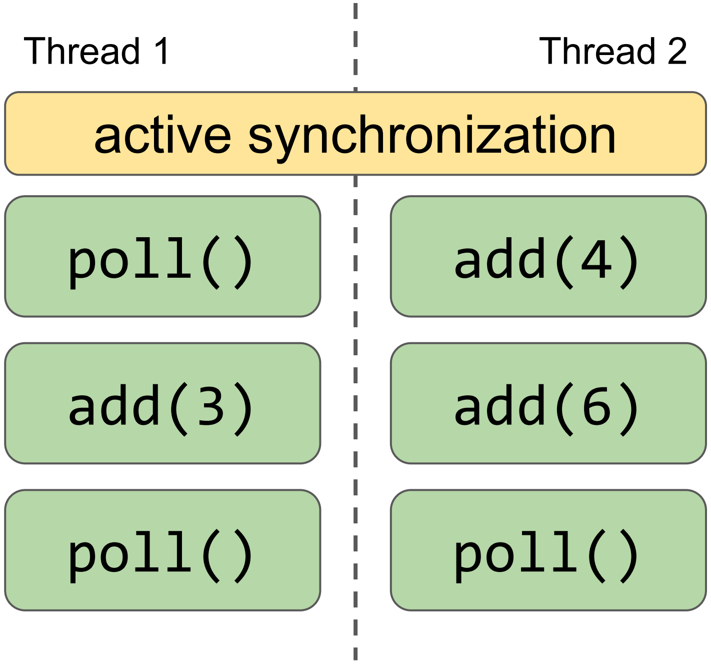
```
Init part:
[poll(), add(9)]
Parallel part:
| poll() | add(4) |
| add(3) | add(6) |
| poll() | poll() |
Post part:
[add(1)]
```

Sequential parts

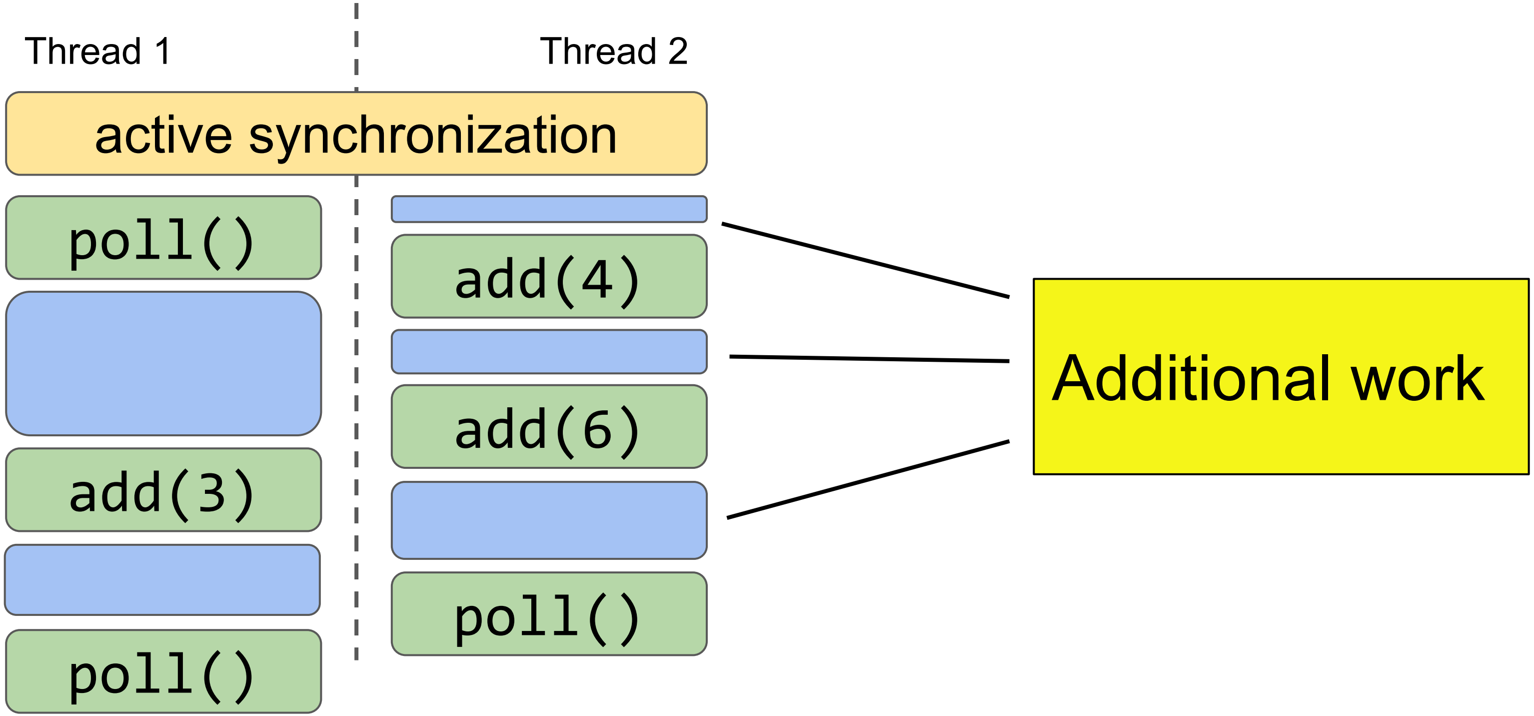
```
Init part:
[poll(), add(9)]
Parallel part:
| poll()  | add(4) |
| add(3)  | add(6) |
| poll()  | poll() |
Post part:
[add(1)]
```

How to run the parallel part?

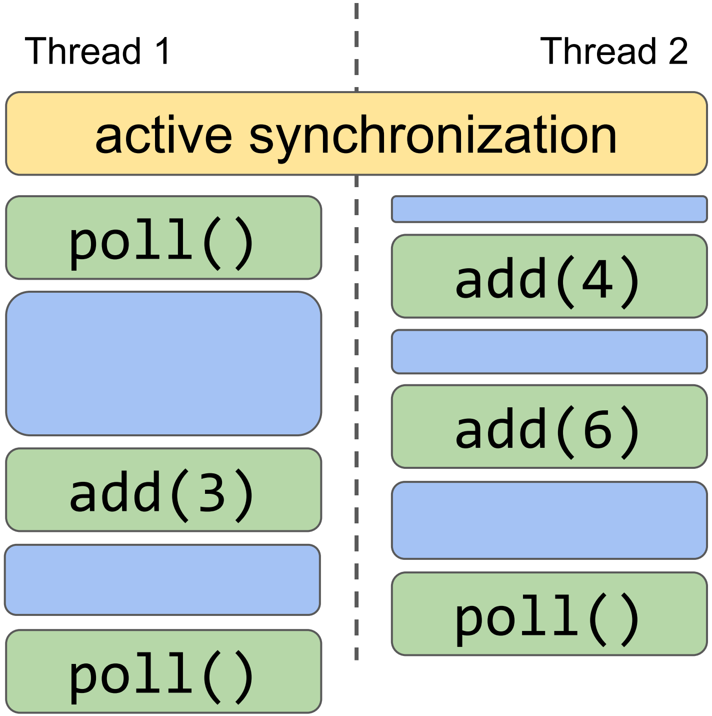
# Stress Testing



# Stress Testing



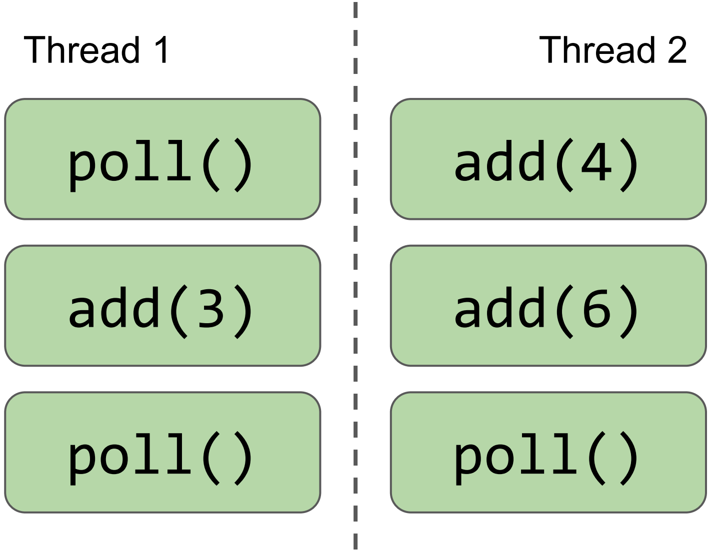
# Stress Testing




```kotlin
class MySuperFastQueueTest {
    ...

    @Test fun test() = StressOptions()
        .invocationsPerIteration(100500)
        .check(this::class)
}
```

# Model Checking




- Sequential Consistency (no races)
- Bounded by number of interleavings
- Increases the number of context switches
- Brute forces interleavings evenly

github.com/Kotlin/kotlinx-lincheck/pull/5

# Model Checking

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

   fun getAndInc(): Int {
       val cur = value // line 28
       value = cur + 1 // line 29
       return cur
   }

   fun get() = value
}
```

github.com/Kotlin/kotlinx-lincheck/pull/5

# Model Checking

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

   fun getAndInc(): Int {
       val cur = value // line 28
       value = cur + 1 // line 29
       return cur
   }

   fun get() = value
}
```

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

   @Operation fun getAndInc() = c.getAndInc()
   @Operation fun get() = c.get()

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

github.com/Kotlin/kotlinx-lincheck/pull/5

# Model Checking

```
java.lang.AssertionError: Invalid interleaving found:
= Invalid execution results: =
Parallel part:
| getAndInc(): 0 | getAndInc(): 0 |
Parallel part execution trace:
|                  | getAndInc(): 0                          |
|                  |     Counter.getAndInc(CounterTest.kt:28) |
|                  |     SWITCH                              |
| getAndInc(): 0   |                                         |
| SWITCH           |                                         |
|                  |     Counter.getAndInc(CounterTest.kt:29) |
|                  |     RESULT: 0                           |
|                  |     FINISH                              |
```

github.com/Kotlin/kotlinx-lincheck/pull/5

# How to check results?

# Results Verification

Simplest solution:

1. Generate all possible sequential histories and produce all possible results ***in advance***

2. On each invocation: check whether the current results are among the generated ones

# Results Verification

Simplest solution:

1. Generate all possible sequential histories
   and produce all possible results ***in advance***

2. On each invocation: check whether the current results
   are among the generated ones

2 threads x 15 operations ⇒ OutOfMemoryError

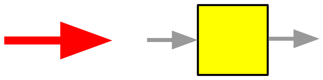
# Results Verification

Simplest solution:

1. Generate all possible sequential histories
   and produce all possible results ***in advance***
2. On each invocation: check whether the current results
   are among the generated ones

## Smarter solution: State Machine (LTS)

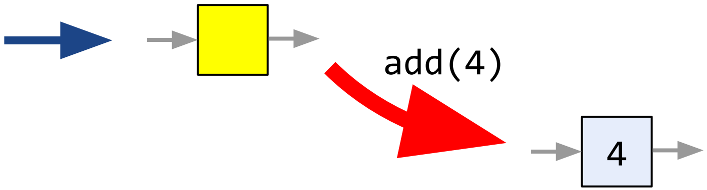
# LTS-Based Verification



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

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

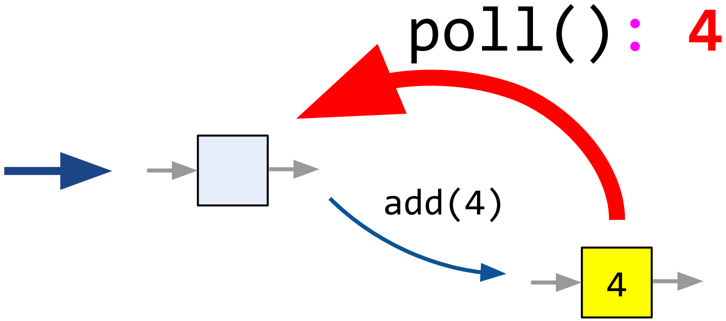
# LTS-Based Verification




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

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

# LTS-Based Verification

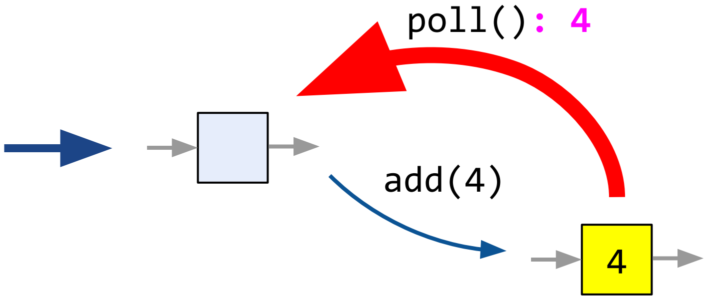



```
val q = MSQueue<Int>()
```

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

Result is different

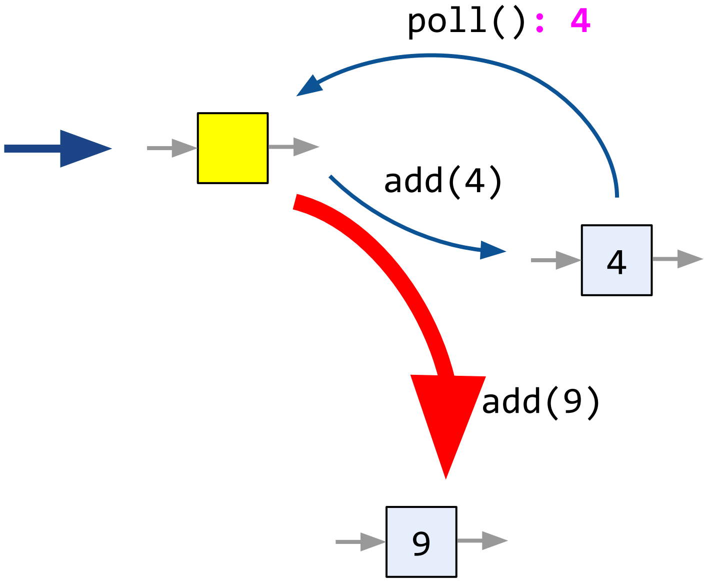
# LTS-Based Verification




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

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

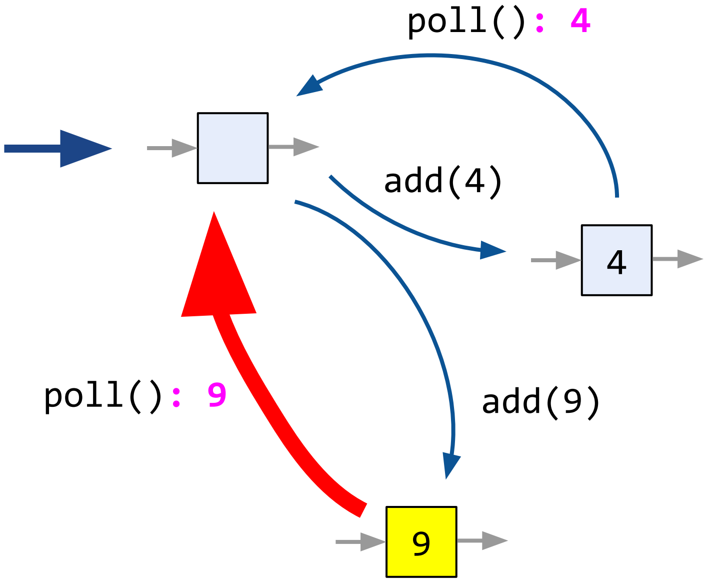
# LTS-Based Verification




`val q = MSQueue<Int>()`

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

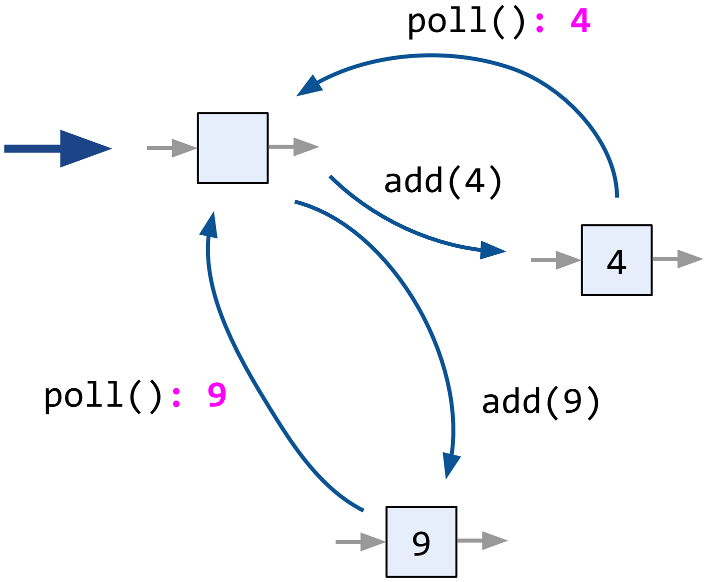
# LTS-Based Verification




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

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

# LTS-Based Verification




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

| | |
|---|---|
| q.add(4)<br>q.poll(): 9 | q.poll(): 4<br>q.add(9) |

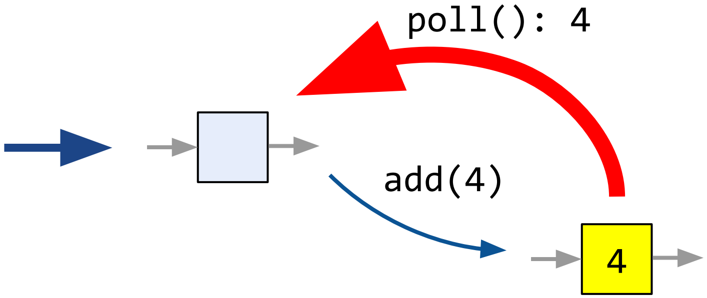
**A path is found ⇒ correct**

# Lazy LTS Creation

- We build LTS lazilly, like on the previous slides
- We use sequential implementation

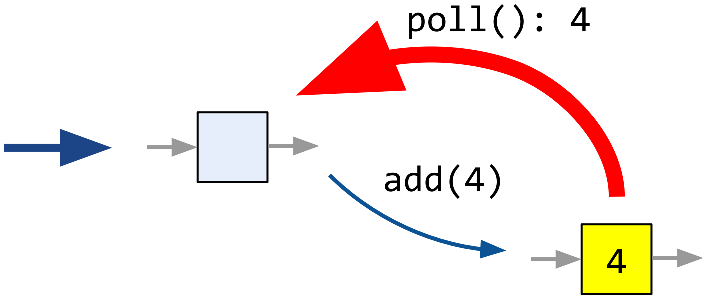
# Lazy LTS Creation

- We build LTS lazilly, like on the previous slides
- We use sequential implementation



# Lazy LTS Creation

- We build LTS lazilly, like on the previous slides
- We use sequential implementation
- Equivalence via `equals`/`hashcode` implementations




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

    // Operations here

    override fun generateState()
                        = elements(q)
}
```

# Sequential Specification

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

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

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

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

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

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

# What if my data structure is blocking by design?

# Rendezvous Channels

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

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

`send` waits for `receive` and vice versa

# Rendezvous 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)
```

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

# Rendezvous Channels

Have to wait for send

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 Channels

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 Channels

## Client 1

```
val task = Task(...)
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 Channels

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 Channels

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 Channels

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 Channels

### 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)
```

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

# Rendezvous Channels

Client 1

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

Worker

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

Client 2

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

Rendezvous!

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

```
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*

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

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

*register as a waiter* } request

*suspend*

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

*return element* } follow-up

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

# Rendezvous Channel Test Example

```
class RendezvousChannelTest: LinCheckState() {
   val c = Channel()

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

   override fun generateState() = Unit
}
```

# Rendezvous Channel Test Example

```
class RendezvousChannelTest: LinCheckState() {
   val c = Channel()

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

   override fun generateState() = Unit
}
```

Why “Unit”?

# State Equivalence

1. List of suspended operations
2. Set of resumed operations
3. *Externally observable state*

# State Equivalence

Maintained by *lincheck*

1. List of suspended operations
2. Set of resumed operations
3. *Externally observable state*

Specified via `equals/hashcode`

# Rendezvous Channel Test Example

```
class RendezvousChannelTest: LinCheckState() {
    val c = Channel()

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

    override fun generateState() = Unit
}
```

Why "Unit"?

Suspended and resumed operations define the channel state

# Buffered Channels

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)
}
```

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!

Worker

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

Client 2

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

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

# Buffered Channels

## Client 1

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

## Worker

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

## Client 2

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

The buffer is full, suspends

```kotlin
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

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

(1)

Client 2

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

(2)

Worker

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

(4) (3)

Retrieves the 2nd task,
no waiters to resume

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

# Buffered Channel Test Example

```kotlin
class BufferedChannelTest: LinCheckState() {
    val c = Channel()

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

    override fun generateState() = bufferedElements(c)
}
```

# Buffered Channel Test Example

```
class BufferedChannelTest: LinCheckState() {
    val c = Channel()

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

    override fun generateState() = bufferedElements(c)
}
```

*Externally observable state* = buffered elements + waiting senders elements **(optionally)**


https://www.youtube.com/watch?v=hwbpUEGHvvY

Are all “correct” data structures linearizable?

Sequential consistency

Quiescent consistency

Quasi-linearizability

Quantitative relaxation

Local linearizability

**Sequential consistency**

Quiescent consistency

Quasi-lineariz

We actually test for it

Quantitative relaxation

Local linearizability

Sequential consistency

**Quiescent consistency**

Quasi-li

We support this formalism,
and use it in Kotlin Coroutines*

Quantitative relaxation

Local linearizability

* https://github.com/Kotlin/kotlinx.coroutines/blob/1.3.2/kotlinx-coroutines-core/common/src/internal/LockFreeTaskQueue.kt

Sequential co… istency

Quiescent co…

**Decided to remove these contracts from *lincheck****

**Quasi-linearizability**

**Quantitative relaxation**

Local linearizability

* Got best decision award :)

Sequential consistency

Quiescent consistency

Quasi-linearizabil

Quantitative r

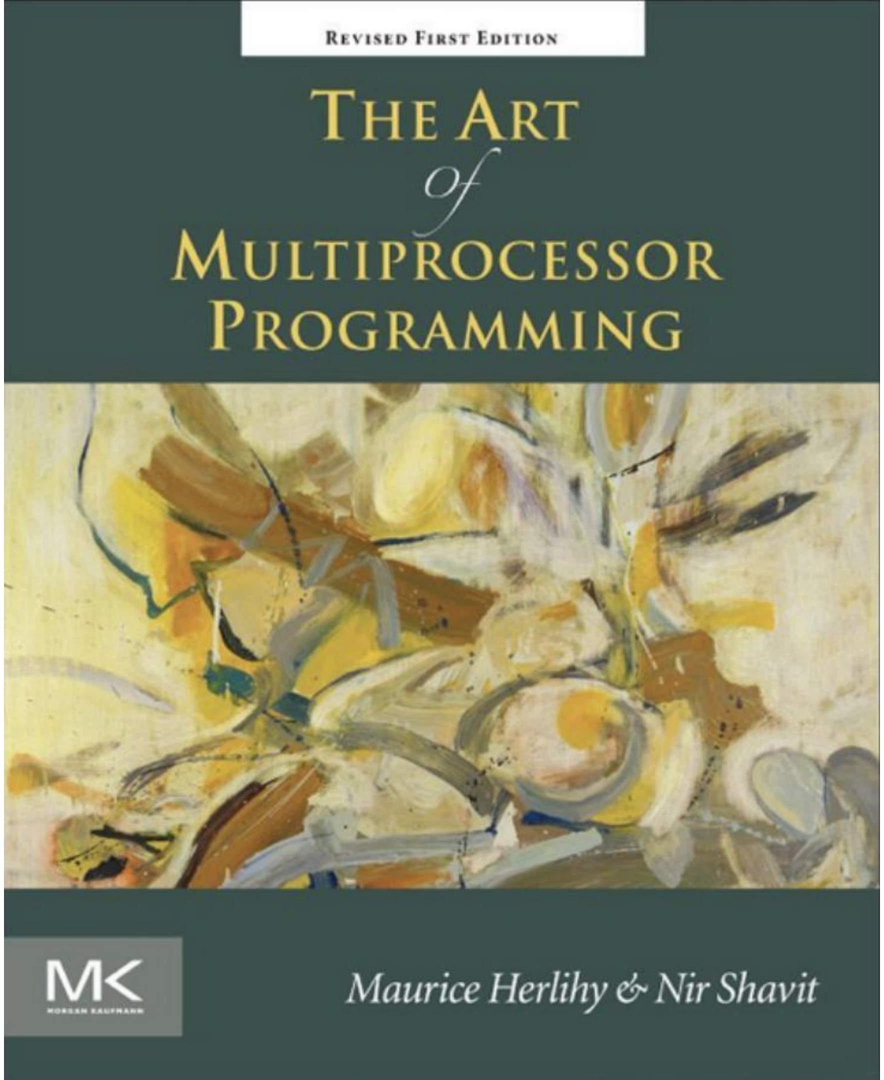
Not supported
Never have
Never will

**Local linearizability**

# Summary

- It is easy to check concurrent data structures with *lincheck*
- We support various popular contracts
  - single reader/writer, dual data structures
  - serializability, quiescent consistency

- We use *lincheck* in Kotlin Coroutines to test our algorithms and student assignments


REVISED FIRST EDITION
THE ART of MULTIPROCESSOR PROGRAMMING
Maurice Herlihy & Nir Shavit
MK

# Useful Materials

Hydra Conference

hydraconf.com

Summer Schools in SPb (2017 & 2019)

neerc.ifmo.ru/sptcc

sptdc.ru

Homework Assignments @ITMO (Koval & Elizarov)

github.com/ITMO-MPP

# Main Research Conferences

**PPoPP**. Principles and ***Practice*** of Parallel Programming

**PODC**. Symposium on ***Principles*** of Distributed Computing

**SPAA**. Symposium on ***Parallelism*** in Algorithms and Architectures

Others: DISC, OPODIS, Euro-Par, IPDPS, PACT, ...

# Questions?

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

Sequential model

↓

sequential specification
on operations

Concurrent model

↓

Linearizability
(usually)

# Custom Scenario Generators

```
class MyScenarioGenerator(testCfg: CTestConfiguration, testStr: CTestStructure)
    : ExecutionGenerator(testCfg, testStructure)
{
   override fun nextExecution() = ExecutionScenario(
       emptyList(), // init part
       listOf(
           listOf( Actor(method = MyQueueTest::add.javaMethod!!,
                             arguments = listOf(1),   handledExceptions = emptyList()) ),
           listOf( Actor(method = MyQueueTest::poll.javaMethod!!,
                             arguments = emptyList(), handledExceptions = emptyList()) )
       ),
       emptyList() // post part
   )
}
```

# Custom Scenario Generators

```
class MyScenarioGenerator(testCfg: CTestConfiguration, testStr: CTestStructure)
    : ExecutionGenerator(testCfg, testStructure)
{
   override fun nextExecution() = …
}


class MyQueueTest {
   ...
   @Test fun test() = StressOptions()
      .executionGenerator(MyScenarioGenerator::class.java)
      .check(this::class)
}
```

# Custom Scenario Generators

```
class MyScenarioGenerator(testCfg: CTestConfiguration, testStr: CTestStructure)
    : ExecutionGenerator(testCfg, testStructure)
{
   override fun nextExecution() = …
}


class MyQueueTest {
   ...
   @Test fun test() = StressOptions()
      .executionGenerator(MyScenarioGenerator::class.java)
      .check(this::class)
}
```

Be careful, the running code can be loaded by another `ClassLoader`!

# Can suspending operations be cancellable?

TODO: Cancellation support