Lincheck. Тестирование многопоточных структур данных.

Соколова Мария, Heisenbug 2019

Писать многопоточный код сложно

Писать многопоточный код сложно

… тестировать еще сложнее

Писать многопоточный код сложно

… тестировать еще сложнее

Счётчик

```
class Counter {
    var value = 0
    fun inc() = ++value
}
```

Счётчик

```
class Counter {
    var value = 0
    fun inc() = ++value
}
```

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

`c.inc()`	`c.inc()`

Счётчик

```
class Counter {
    var value = 0
    fun inc() = ++value
}
```

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

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

Счётчик

```
class Counter {
    var value = 0
    fun inc()  = ++value
}
```

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

	c.inc(): 1 (2)
(1) c.inc(): 2	

Счётчик

```
class Counter {
    var value = 0
    fun inc()  = ++value
}
```

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

`c.inc(): 1`	`c.inc(): 1`

Счётчик

```
class Counter {
    var value = 0
    fun inc()  = ++value
}
```

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

`c.inc(): 1`	`c.inc(): 1`

Не ожидаем такого поведения счетчика

Счётчик

```
class Counter {
    var value = 0
    fun inc()  = ++value
}
```

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

1 `value.R(): 0`	
	`value.R(): 0` 2
	`value.W(1)` 3
4 `value.W(1)`	

Банк

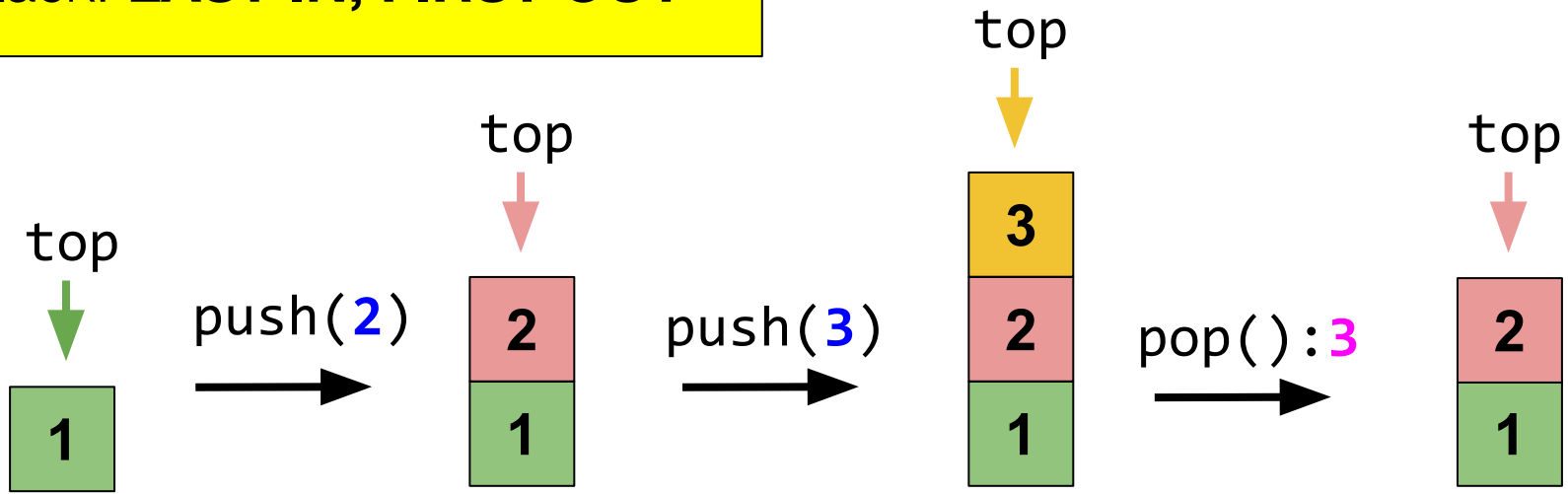
Корректность

Последовательный алгоритм  Последовательная спецификация на операциях

Корректность

Последовательный алгоритм → Последовательная спецификация на операциях

Stack: **LAST IN, FIRST OUT**



Корректность

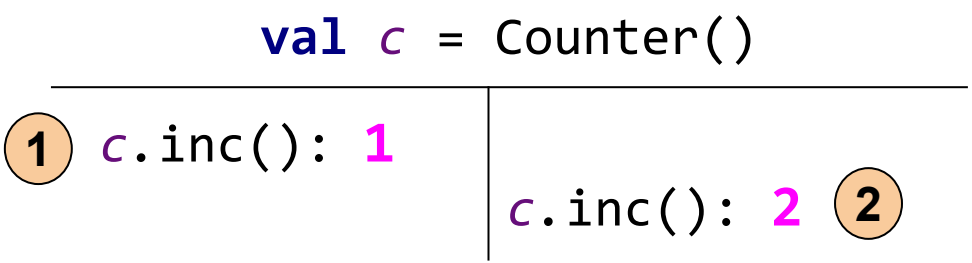
Последовательный алгоритм → Последовательная спецификация на операциях

Многопоточный алгоритм → **Атомарность** (линеаризуемость)

Корректность

Последовательный алгоритм → Последовательная спецификация на операциях

Многопоточный алгоритм → **Атомарность** (линеаризуемость)



Исполнение **линеаризуемо** ⇔ ∃ эквивалентное последовательное исполнение

Исполнение **линеаризуемо** ⇔ ∃ эквивалентное последовательное исполнение

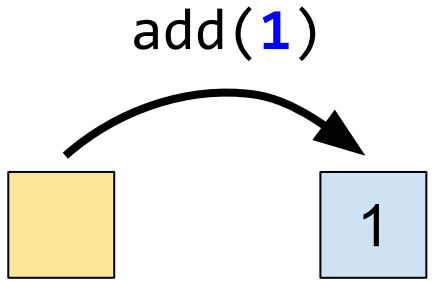
```
val q = MyQueue<Int>()
```

`add(1)`	`poll(): 1`
`poll(): 2`	`add(2)`

Исполнение **линеаризуемо** ⇔ ∃ эквивалентное последовательное исполнение

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

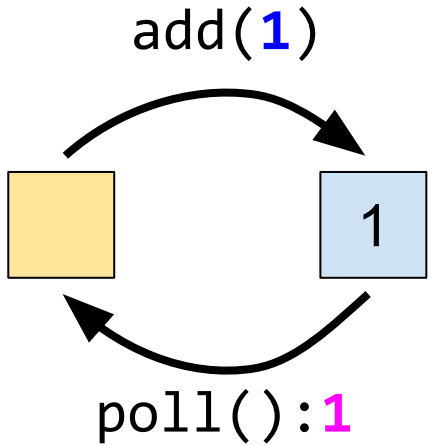
1 `add(1)`	`poll(): 1`
`poll(): 2`	`add(2)`



Исполнение **линеаризуемо** ⇔ ∃ эквивалентное последовательное исполнение

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

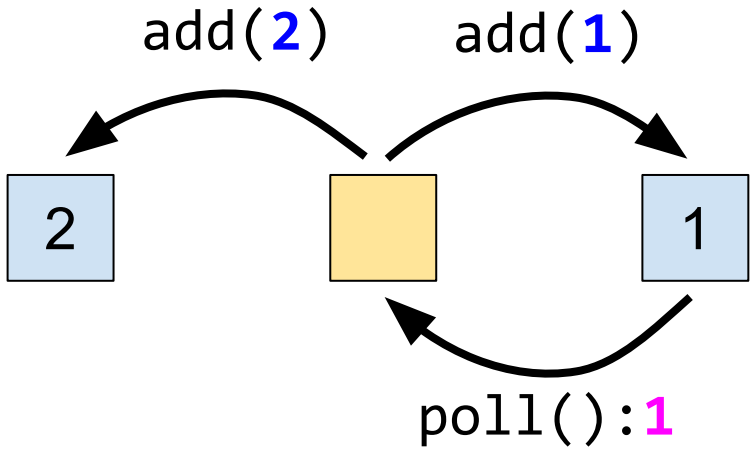
① `add(1)`	
	`poll(): 1` ②
`poll(): 2`	`add(2)`



Исполнение **линеаризуемо** ⇔ ∃ эквивалентное последовательное исполнение

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

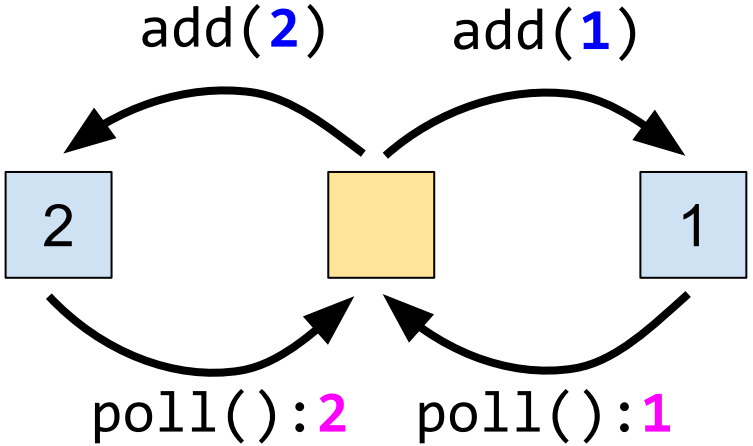
① `add(1)`	`poll(): 1` ②
	`add(2)` ③
`poll(): 2`	



Исполнение **линеаризуемо** ⇔ ∃ эквивалентное последовательное исполнение

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

1 `add(1)`	`poll(): 1` 2 `add(2)` 3
4 `poll(): 2`	



```
class Counter {
    private var value = 0
    fun inc()  = ++value
}
```

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

`c.inc(): 1`	`c.inc(): 1`

Такое исполнение
нелинеаризуемо

Как проверить **корректность** структуры данных?

Формальная верификация

Верификация руками

Доказательство инвариантов для всех возможных исполнений алгоритма на бумаге

Полуавтоматическая верификация

Интерактивные инструменты для доказательств:

- Coq, Isabelle, TLA+

Автоматическая верификация

Model checking:

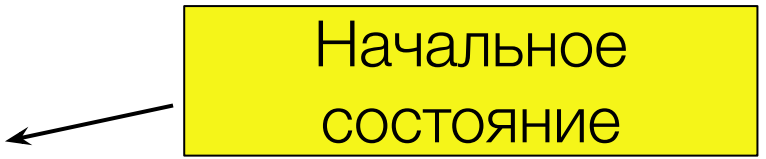
- Java PathFinder (JPF)
- Для всех возможных исполнений алгоритма проверяется выполнение инварианта
- В этом году появился инструмент, позволяющий верифицировать исполнения с учетом ослабленной модели памяти

* “**Model Checking of Software Systems under Weak Memory Models**” by Ngo, T.-P. 2019. Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1745`

Тестирование

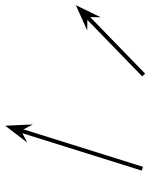
Как хочется тестировать?

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




}
```



Как хочется тестировать?

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

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

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

}
```



Операции структуры данных

Как хочется тестировать?

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

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

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

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

Проверка на линеаризуемость

Lincheck

Lincheck = Linearizability Checker (не только линеаризуемость)

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

Lincheck

Lincheck = Linearizability Checker (не только линеаризуемость)
https://github.com/Kotlin/kotlinx-lincheck

Как работает **lincheck**:

1. Генерирует рандомные сценарии
2. Исполняет каждый сценарий много раз
3. Проверяет корректность полученных результатов

Как сгенерировать сценарий?

Генерация сценария

```
class MyQueueTest {
    ...
    @Operation fun add(x: Int) = ...
    @Operation fun poll() = ...

}
```

Генерация сценария

```
class MyQueueTest {
    ...
    @Operation fun add(x: Int) = ...
    @Operation fun 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)]
```

Генерация сценария

```
class MyQueueTest {
    ...
    @Operation fun add(x: Int) = ...
    @Operation fun poll() = ...

    @Test fun test() = StressOptions()
            .actorsBefore(2)
            .threads(2).actorsPerThread(3)
            .actorsAfter(1)
            .iterations(100_000)
            .check(this::class)
}
```

Сколько разных сценариев сгенерировать?

Генерация сценария

```
class MyQueueTest {
    ...
    @Operation fun add(x: Int) = ...
    @Operation fun poll() = ...

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

Генерация сценария

```
class MyQueueTest {
    ...
    @Operation fun add(x: Int) = ...
    @Operation fun poll() = ...

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

Генерируем байт-код для каждого из потоков (ASM)

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

// start thread #1	// start thread #2
r1[0] = q.poll()	r2[0] = q.add(4)
r1[1] = q.add(3)	r2[0] = q.add(6)
// done	// done

Генерация сценария

```
class MyQueueTest {
    ...
    @Operation fun add(x: Int) = ...
    @Operation fun poll() = ...

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

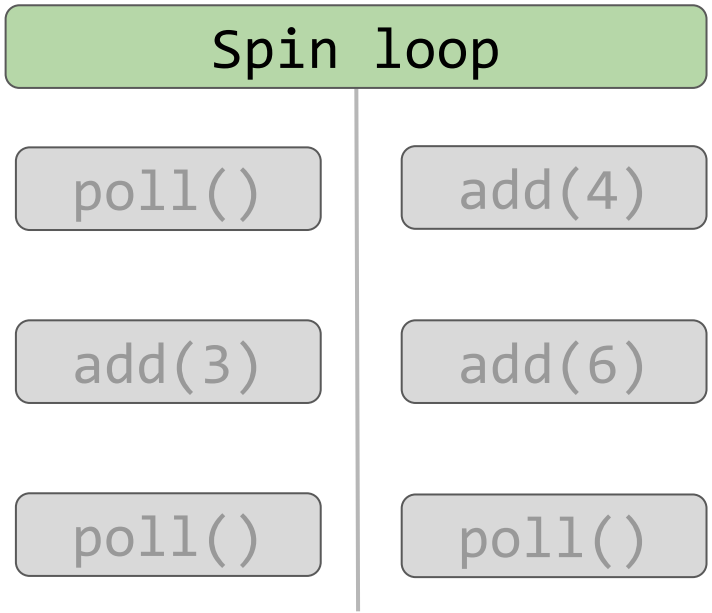
Сохраняем результаты операций

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

// start thread #1	// start thread #2
r1[0] = q.poll()	r2[0] = q.add(4)
r1[1] = q.add(3)	r2[1] = q.add(6)
// done	// done

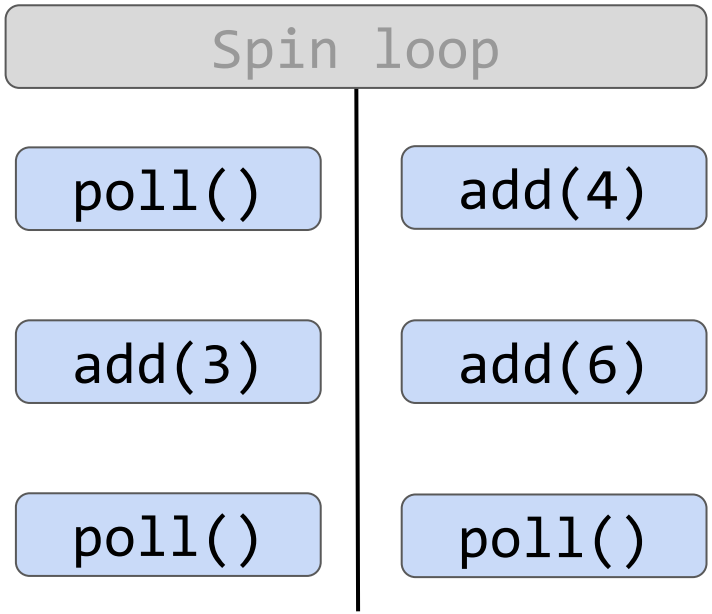
Как исполнять сценарий?

Стресс режим



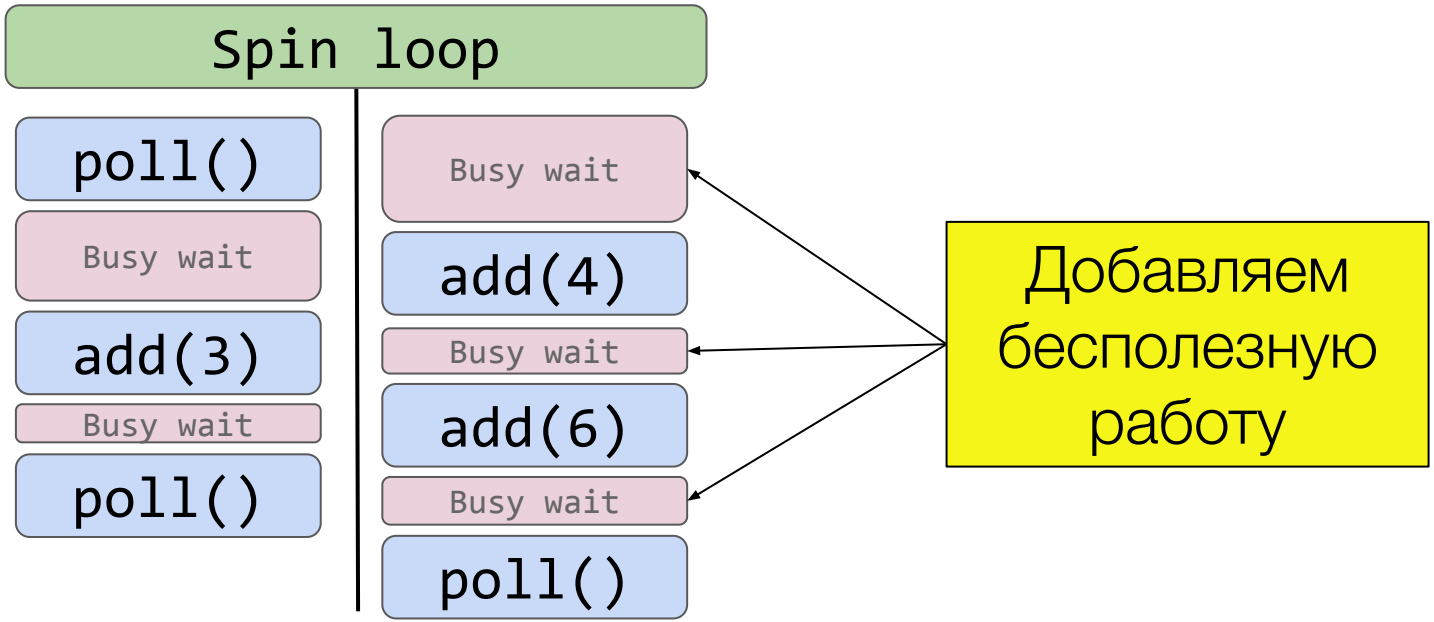
Одновременный старт

Стресс режим

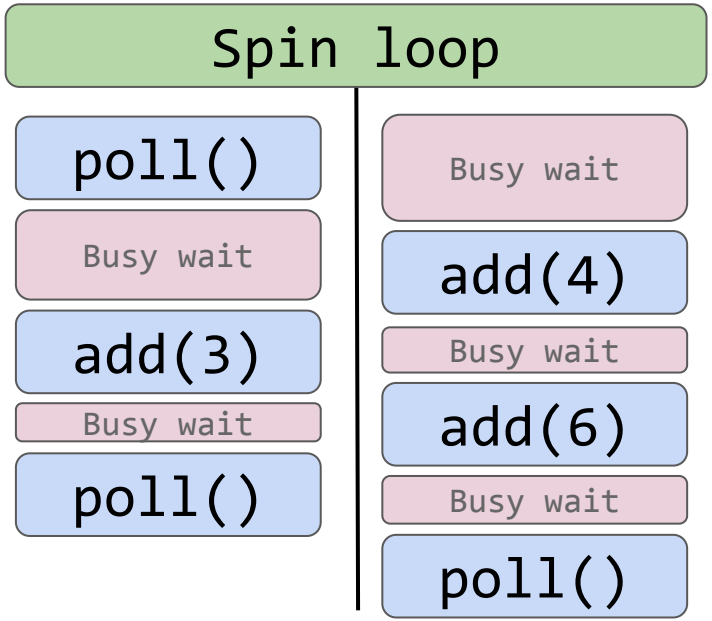


Параллельное исполнение операций

Стресс режим

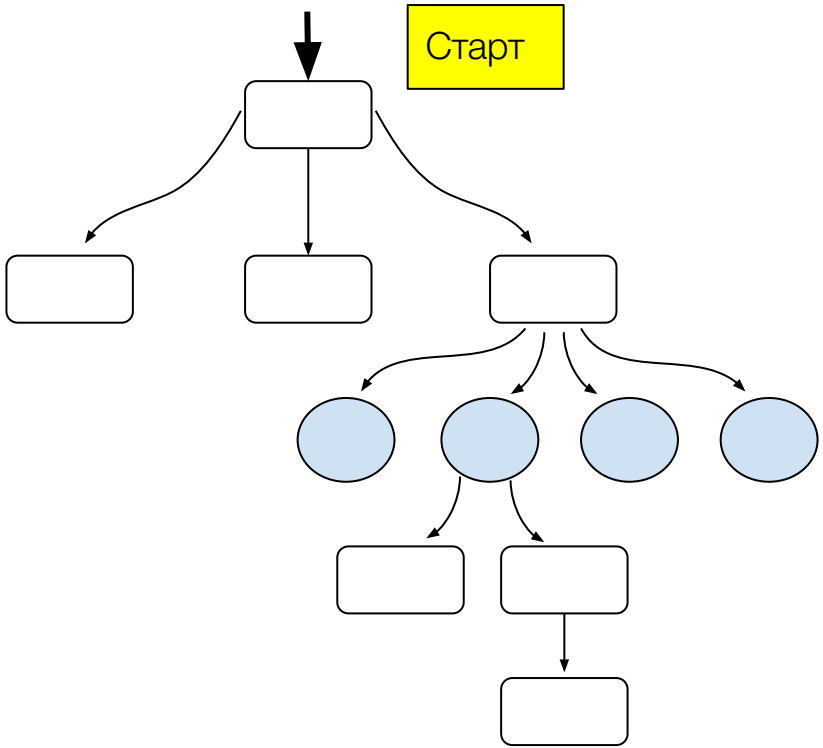


Стресс режим



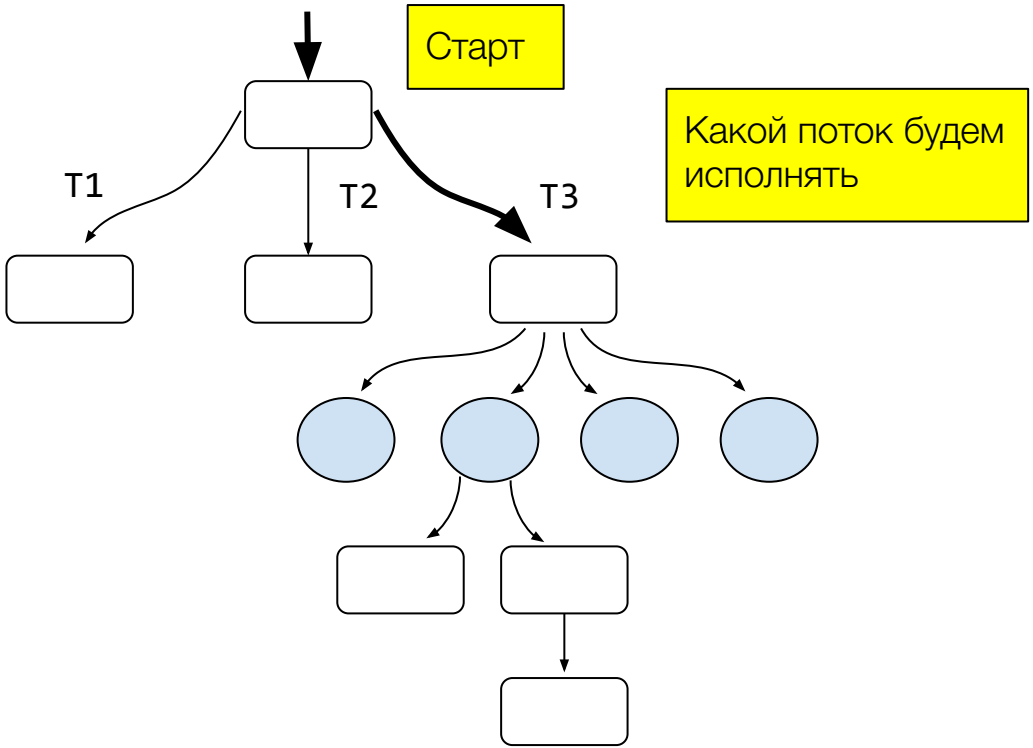

```
class MyQueueTest {
    ...
    @Operation fun add(x: Int) = ...
    @Operation fun poll() = ...

@Test fun test() = StressOptions()
      .threads(2)
      .actorsPerThread(3)
      .invocationsPerIteration(10_000)
      .check(this::class)
}
```

Управляемое исполнение



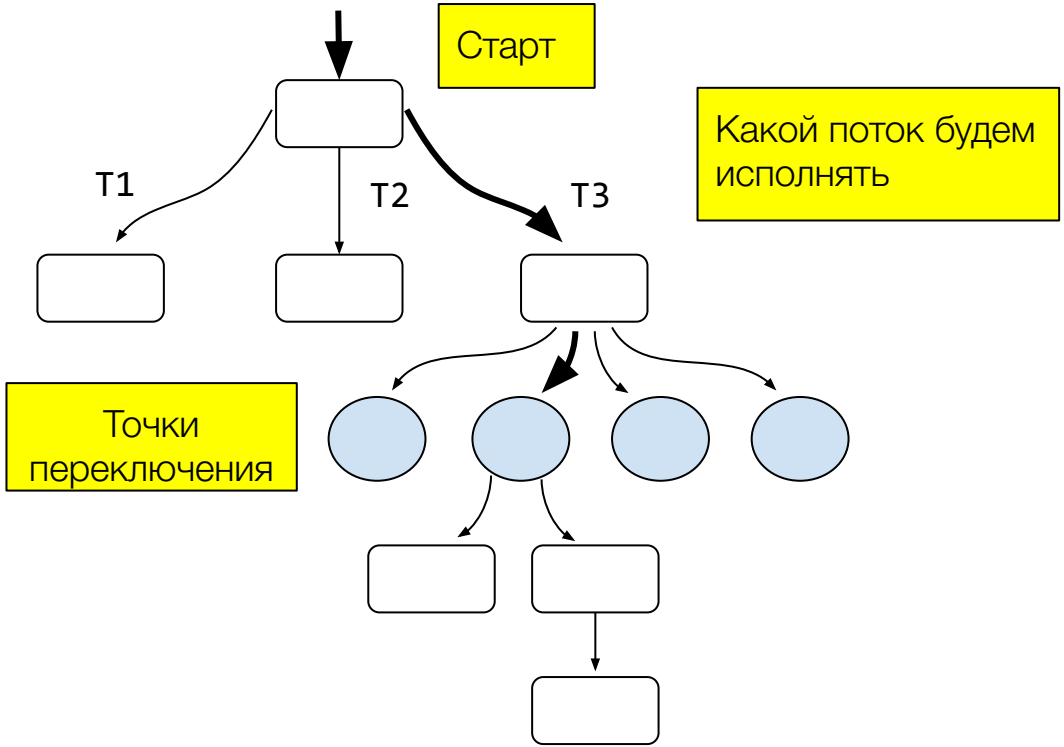
- Строим дерево покрытий всех возможных исполнений
- В каждой вершине храним, какую часть возможных исполнений уже покрыли для взвешенного рандомного выбора, куда переключить исполнение дальше
- Точки переключения для потока: **read / write** операции над разделяемой памятью

Управляемое исполнение



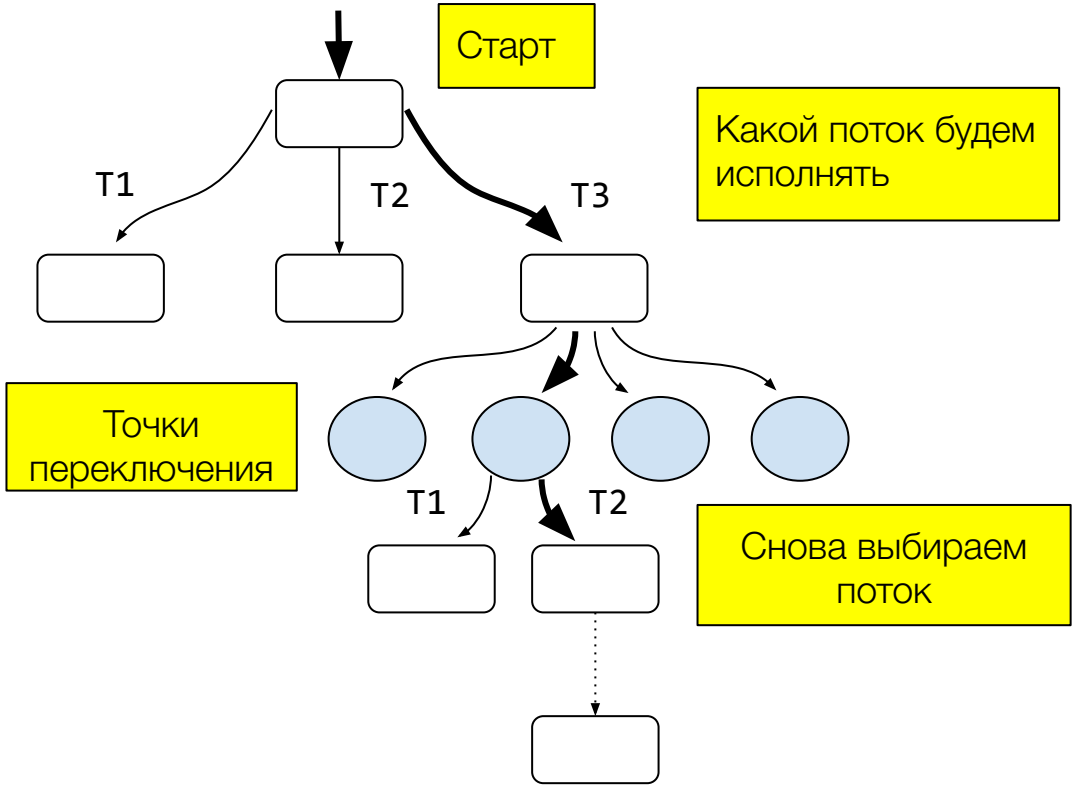
- Строим дерево покрытий всех возможных исполнений
- В каждой вершине храним, какую часть возможных исполнений уже покрыли для взвешенного рандомного выбора, куда переключить исполнение дальше
- Точки переключения для потока: **read / write** операции над разделяемой памятью

Управляемое исполнение



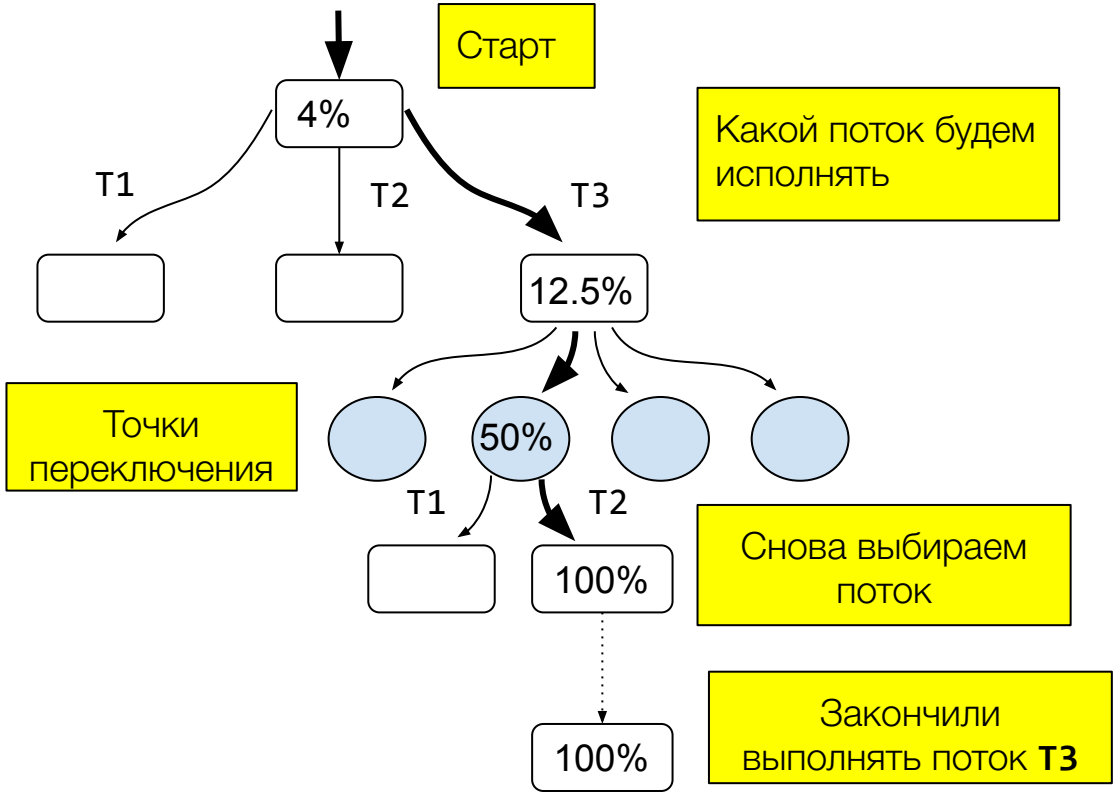
- Строим дерево покрытий всех возможных исполнений
- В каждой вершине храним, какую часть возможных исполнений уже покрыли для взвешенного рандомного выбора, куда переключить исполнение дальше
- Точки переключения для потока: **read / write** операции над разделяемой памятью

Управляемое исполнение



- Строим дерево покрытий всех возможных исполнений
- В каждой вершине храним, какую часть возможных исполнений уже покрыли для взвешенного рандомного выбора, куда переключить исполнение дальше
- Точки переключения для потока: **read / write** операции над разделяемой памятью

Управляемое исполнение



- Строим дерево покрытий всех возможных исполнений
- В каждой вершине храним, какую часть возможных исполнений уже покрыли для взвешенного рандомного выбора, куда переключить исполнение дальше
- Точки переключения для потока это **read / write** операции над разделяемой памятью

Управляемое исполнение: счётчик

```
class Counter {
    private var value = 0


}
```

Управляемое исполнение: счётчик

```
class Counter {
    private var value = 0

    fun inc(): Int {
        val cur = value
        value = cur + 1
        return cur
    }
}
```

Управляемое исполнение: счётчик

```
class Counter {
    private var value = 0

    fun inc(): Int {
        val cur = value
        value = cur + 1
        return cur
    }

    fun get() = value
}
```

Управляемое исполнение: счётчик

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

    fun inc(): Int {
        val cur = value
        value = cur + 1
        return cur
    }

    fun get() = value
}
```

Протестируем

Управляемое исполнение: счётчик

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

    fun inc(): Int {
        val cur = value
        value = cur + 1
        return cur
    }

    fun get() = value
}
```

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

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

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

Управляемое исполнение: счётчик

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

`c.inc(): 0`	`c.inc(): 0`

```
java.lang.AssertionError:
Invalid interleaving found:
= Invalid execution results: =
Parallel part:
| inc(): 0 | inc(): 0 |
Parallel part execution trace:
|           | inc(): 0                                  |
|           |  R(): 0 at Counter.inc(Counter.kt:1)|
|           |  SWITCH                                   |
| inc(): 0 |                                           |
| SWITCH    |                                           |
|           |  W(1) at Counter.inc(Counter.kt:2)   |
|           |  RESULT: 0                                |
|           |  FINISH                                   |
```

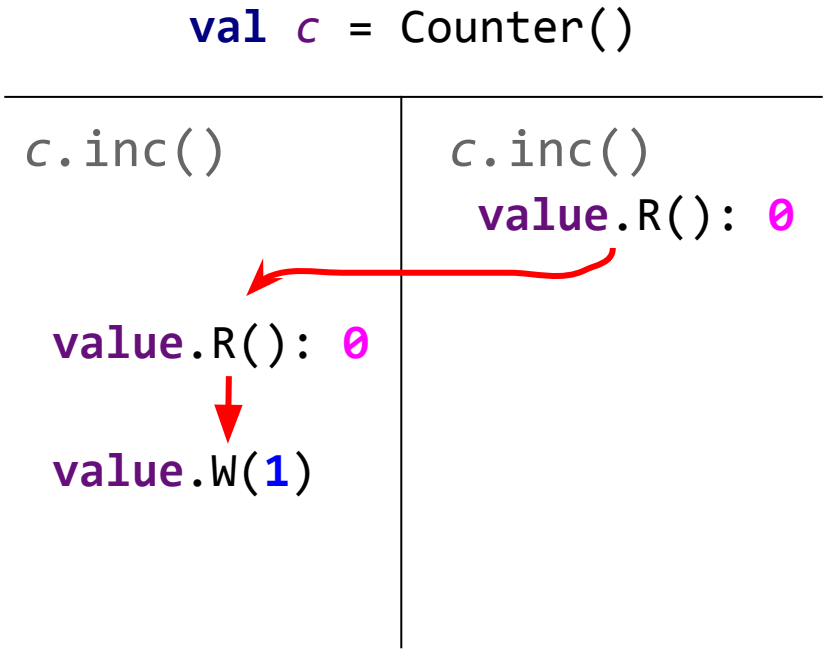
Управляемое исполнение: счётчик

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

`c.inc()`	`c.inc()` `value.R(): 0`

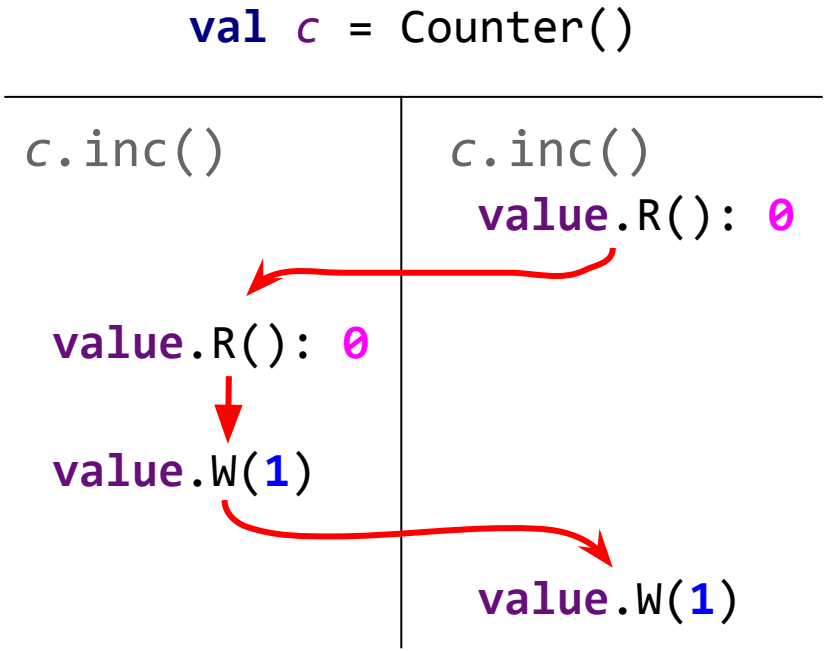
```
java.lang.AssertionError:
Invalid interleaving found:
= Invalid execution results: =
Parallel part:
| inc(): 0 | inc(): 0 |
Parallel part execution trace:
|           | inc(): 0                                  |
|           |  R(): 0 at Counter.inc(Counter.kt:1)|
|           |  SWITCH                                   |
| inc(): 0 |                                           |
| SWITCH    |                                           |
|           |  W(1) at Counter.inc(Counter.kt:2)   |
|           |  RESULT: 0                                |
|           |  FINISH                                   |
```

Управляемое исполнение: счётчик



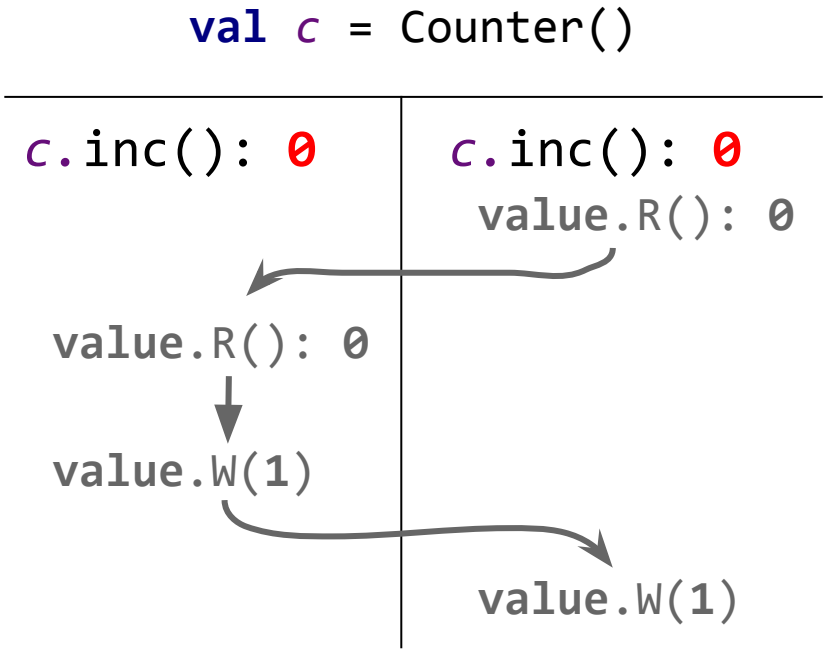

```
java.lang.AssertionError:
Invalid interleaving found:
= Invalid execution results: =
Parallel part:
| inc(): 0 | inc(): 0 |
Parallel part execution trace:
|           | inc(): 0                                  |
|           |  R(): 0 at Counter.inc(Counter.kt:1)|
|           |  SWITCH                                   |
| inc(): 0 |                                           |
| SWITCH   |                                           |
|           |  W(1) at Counter.inc(Counter.kt:2)  |
|           |  RESULT: 0                                |
|           |  FINISH                                   |
```

Управляемое исполнение: счётчик




```
java.lang.AssertionError:
Invalid interleaving found:
= Invalid execution results: =
Parallel part:
| inc(): 0 | inc(): 0 |
Parallel part execution trace:
|           | inc(): 0                                  |
|           |  R(): 0 at Counter.inc(Counter.kt:1)|
|           |  SWITCH                                   |
| inc(): 0 |                                           |
| SWITCH   |                                           |
|           |  W(1) at Counter.inc(Counter.kt:2)  |
|           |  RESULT: 0                                |
|           |  FINISH                                   |
```

Управляемое исполнение: счётчик




```
java.lang.AssertionError:
Invalid interleaving found:
= Invalid execution results: =
Parallel part:
| inc(): 0 | inc(): 0 |
Parallel part execution trace:
|           | inc(): 0                                  |
|           |  R(): 0 at Counter.inc(Counter.kt:1)|
|           |  SWITCH                                   |
| inc(): 0 |                                           |
| SWITCH   |                                           |
|           |  W(1) at Counter.inc(Counter.kt:2)  |
|           |  RESULT: 0                                |
|           |  FINISH                                   |
```

	Стресс режим	**Управляемое исполнение**
Системность перебора возможных исполнений	-	+
Trace воспроизведения некорректного исполнения	-	+
Воспроизведение багов под гонкой	+	-

	Стресс режим	**Управляемое исполнение**
Системность перебора возможных исполнений	-	+
Trace воспроизведения некорректного исполнения	-	+
Воспроизведение багов под гонкой	+	-

Ослабленные модели памяти

a = ? b = ?

```
int x, y
```

`x := 1`	`y := 1`
`a := y`	`b := x`

Ослабленные модели памяти

int x, y	
x := 1	y := 1
a := y	b := x

a = 0	**b** = 1
a = 1	**b** = 0
a = 1	**b** = 1

Последовательная согласованность (Sequential consistency)
(Лампорт, 1979): «Результат любого исполнения не отличим от случая, когда все операции на всех процессорах исполняются в некотором последовательном порядке, и операции на конкретном процессоре исполняются в порядке, обозначенном программой»

Ослабленные модели памяти

```
int x, y
```

`x := 1`	`y := 1`
`a := y`	`b := x`

a = 0	**b** = 1
a = 1	**b** = 0
a = 1	**b** = 1
a = 0	**b** = 0

При **последовательной согласованности** такие результаты не получить

Ослабленные модели памяти

```
int x, y
```

`x := 1`	`y := 1`
`a := y`	`b := x`

a = 0	**b** = 1
a = 1	**b** = 0
a = 1	**b** = 1
a = 0	**b** = 0

Но воспроспроизводится на **x86**

Ослабленные модели памяти

int x, y	
x := 1	y := 1
a := y	b := x

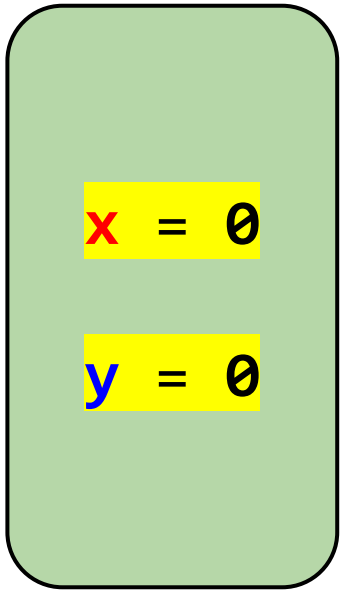
T1

```
x := 1
a := y
```

T2

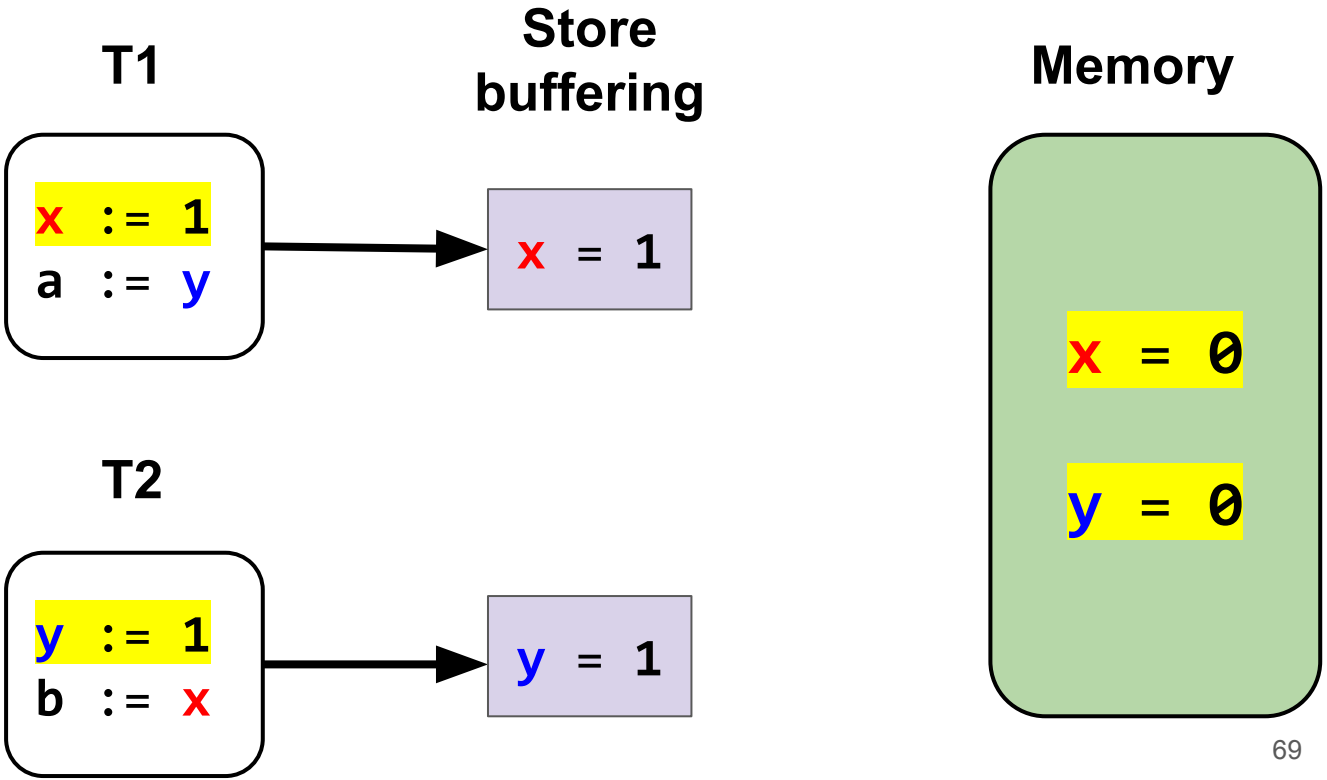
```
y := 1
b := x
```

Memory

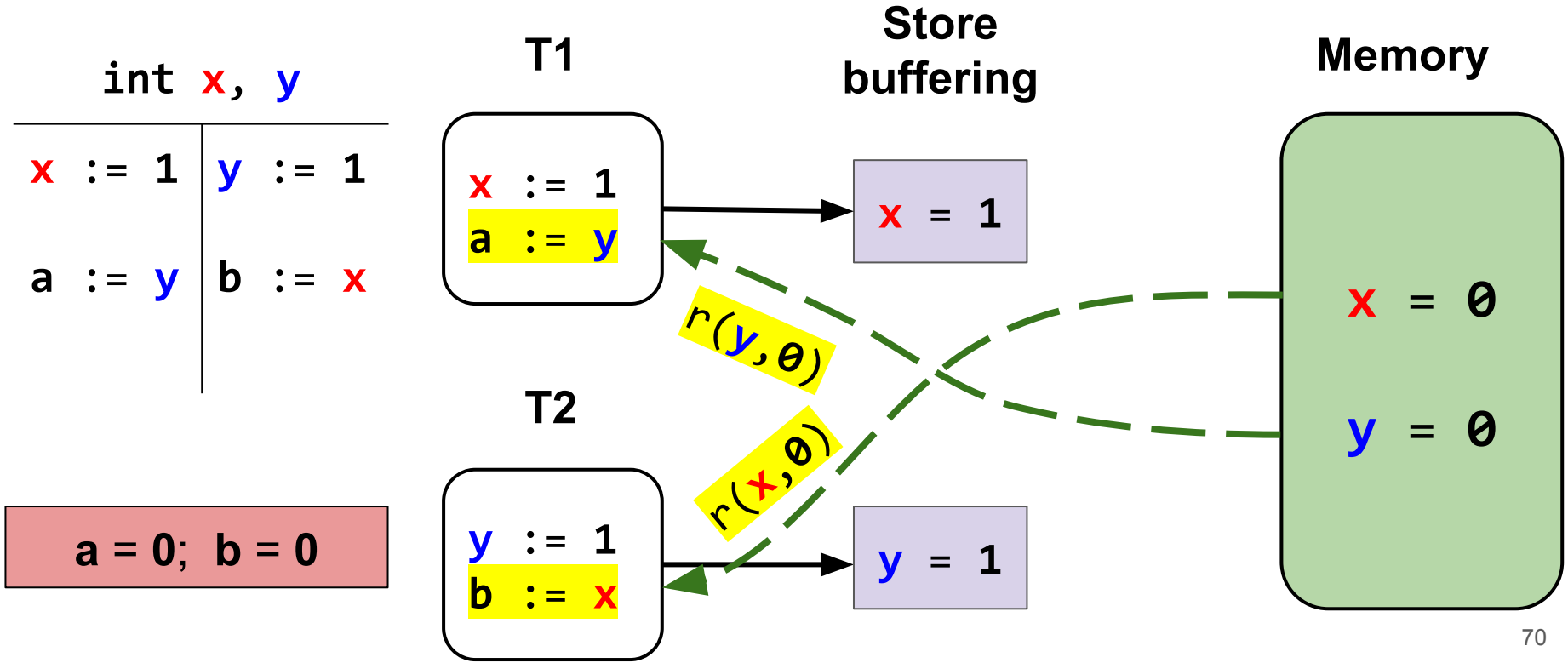


Ослабленные модели памяти

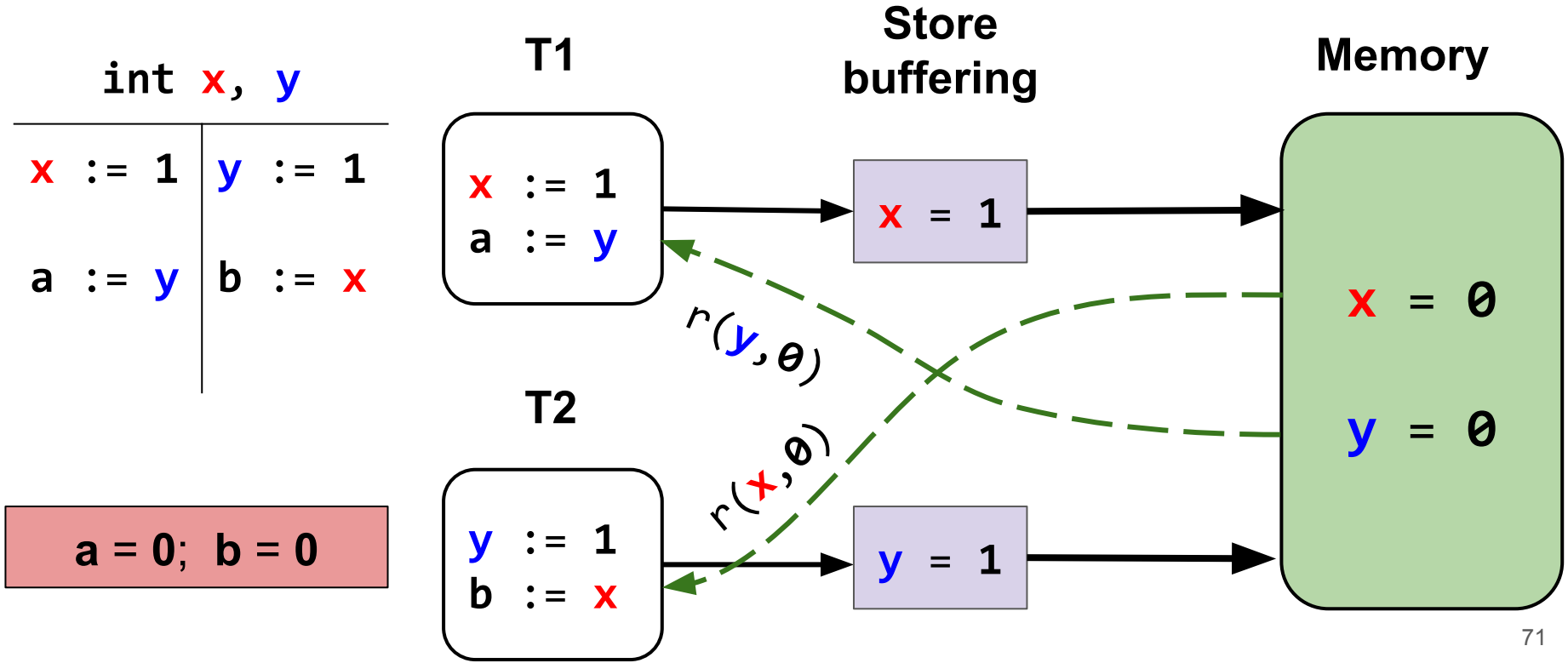
int x, y	
x := 1	y := 1
a := y	b := x



Ослабленные модели памяти



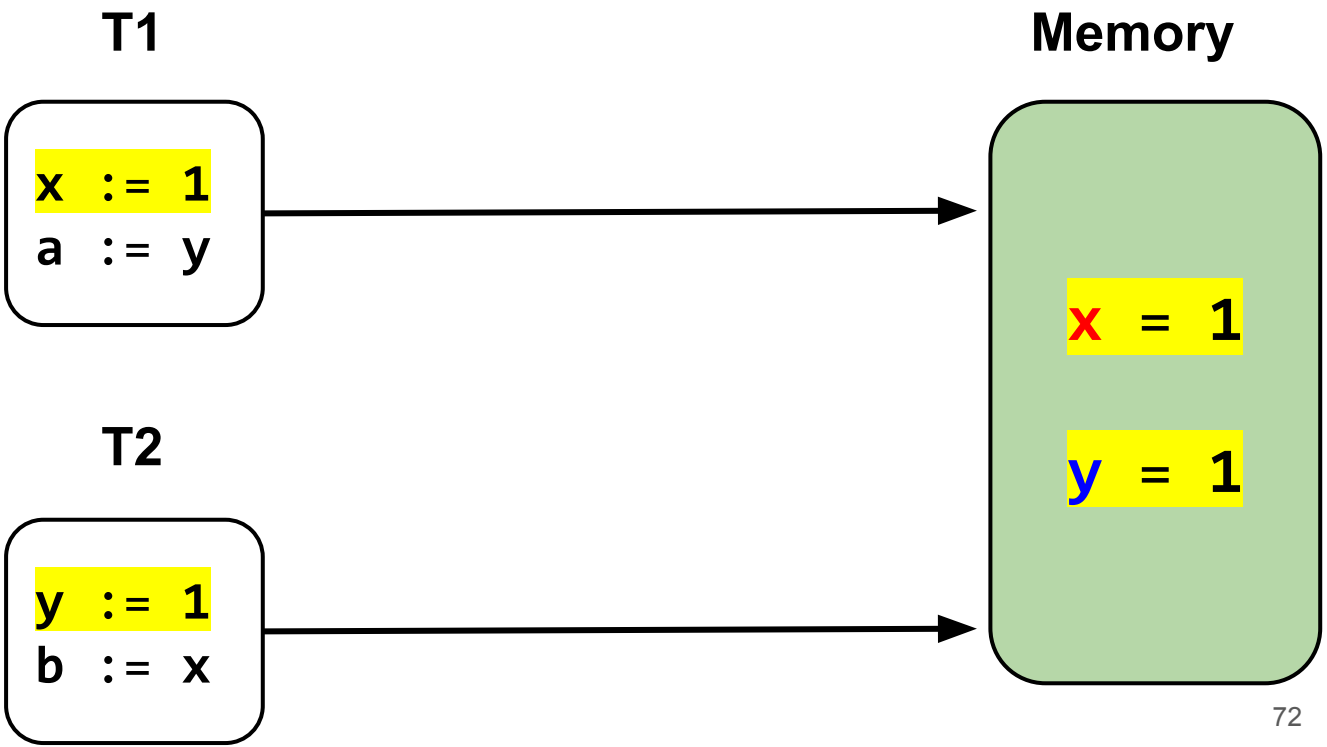
Ослабленные модели памяти



Ослабленные модели памяти

```
volatile int x,y
```

`x := 1`	`y := 1`
`a := y`	`b := x`



Как проверять результаты?

Верификация результатов

Наивное решение:

1. Сгенерировать **все последовательные истории исполнения** и получить все корректные результаты заранее
2. Для каждого запуска сценария проверять, есть ли полученные результаты среди корректных

Верификация результатов

Наивное решение:

1. Сгенерировать **все последовательные истории исполнения** и получить все корректные результаты заранее
2. Для каждого запуска сценария проверять, есть ли полученные результаты среди корректных

2 потока x 15 операций ⇒ OutOfMemoryError

Верификация результатов

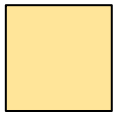
Более эффективное решение:

Граф состояний (Labeled Transition System)

Верификация с помощью LTS

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

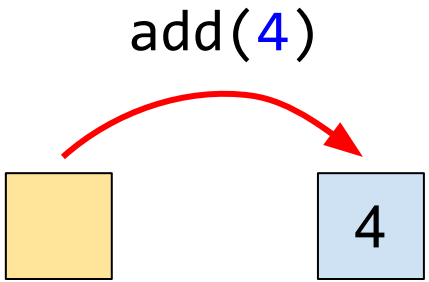
`q.add(4)`	`q.add(9)`
`q.poll(): 9`	`q.poll(): 4`



Верификация с помощью LTS

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

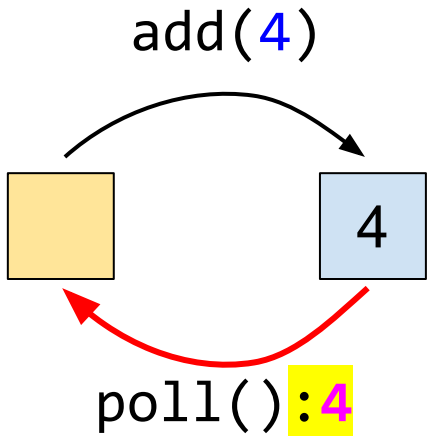
1 `q.add(4)`	
`q.poll(): 9`	`q.add(9)`
	`q.poll(): 4`



Верификация с помощью LTS

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

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

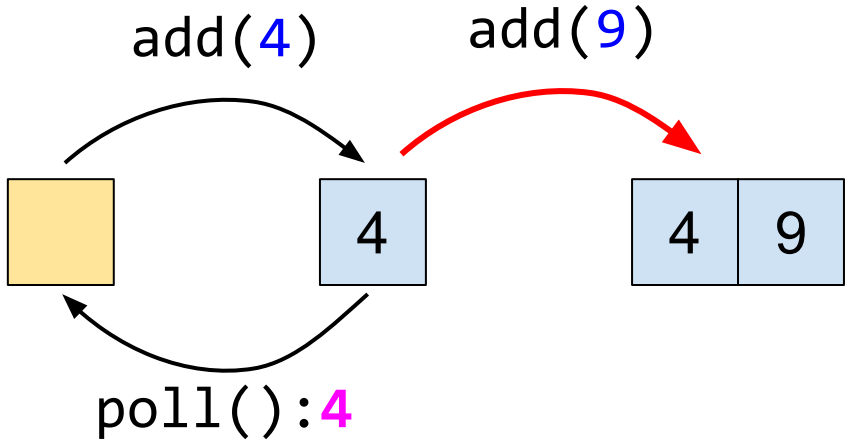


Неверный результат

Верификация с помощью LTS

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

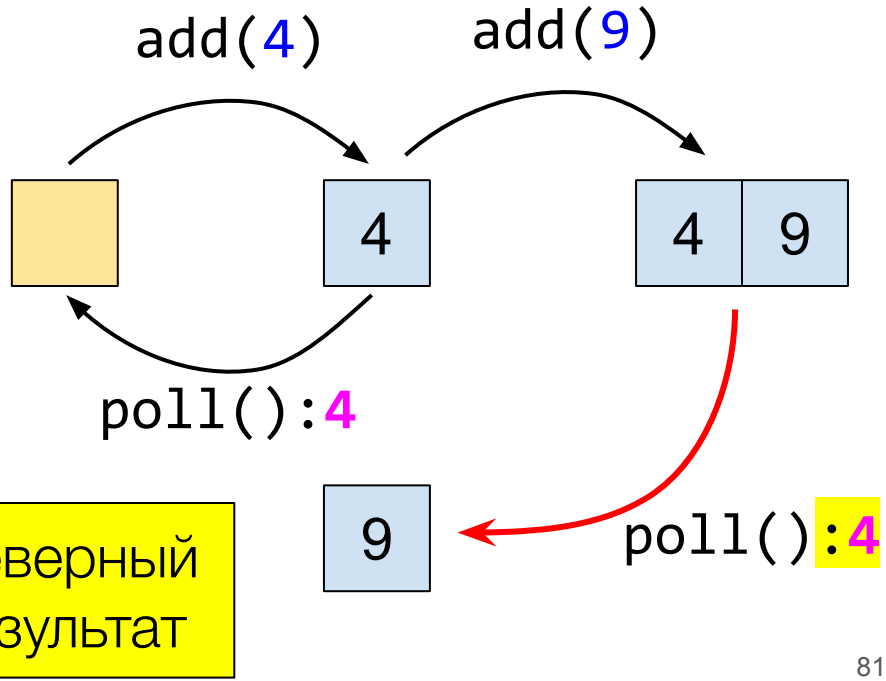
1 `q.add(4)`	
	`q.add(9)` 2
`q.poll(): 9`	`q.poll(): 4`



Верификация с помощью LTS

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

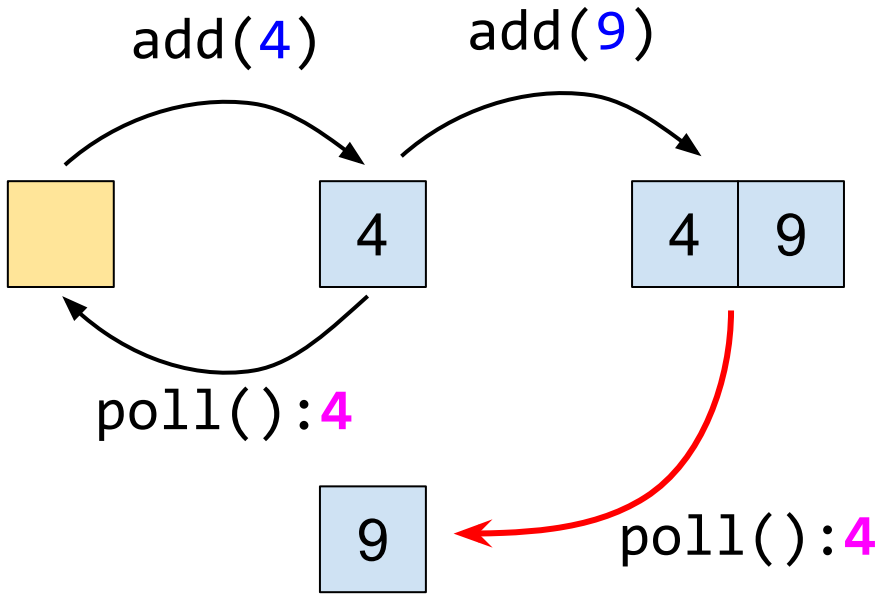
1 `q.add(4)`	
	`q.add(9)` 2
3 `q.poll(): 9`	
	`q.poll(): 4`



Верификация с помощью LTS

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

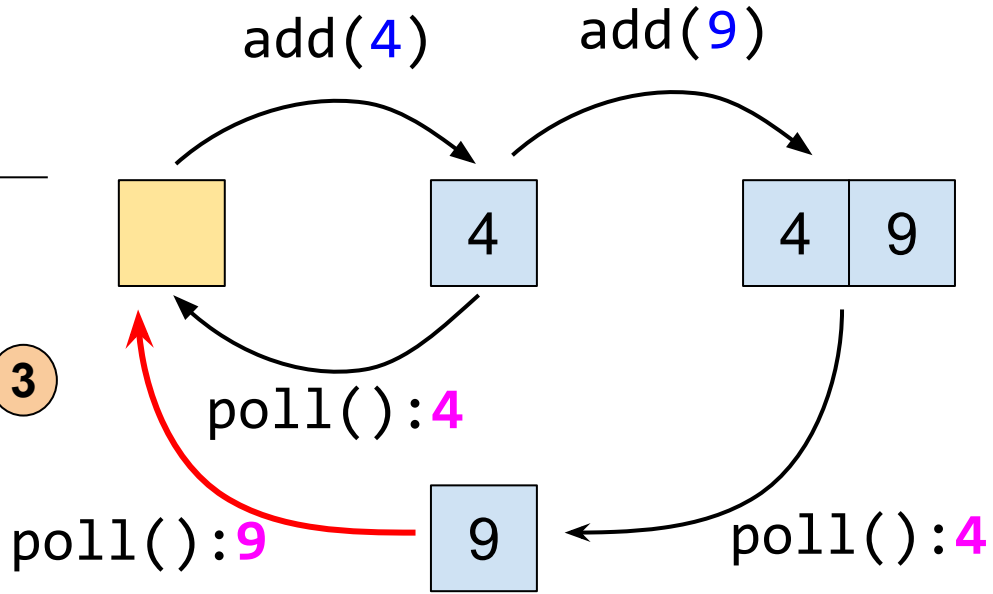
1 `q.add(4)`	
	`q.add(9)` 2
	`q.poll(): 4` 3
`q.poll(): 9`	



Верификация с помощью LTS

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

1 `q.add(4)`	
	`q.add(9)` 2
	`q.poll(): 4` 3
4 `q.poll(): 9`	

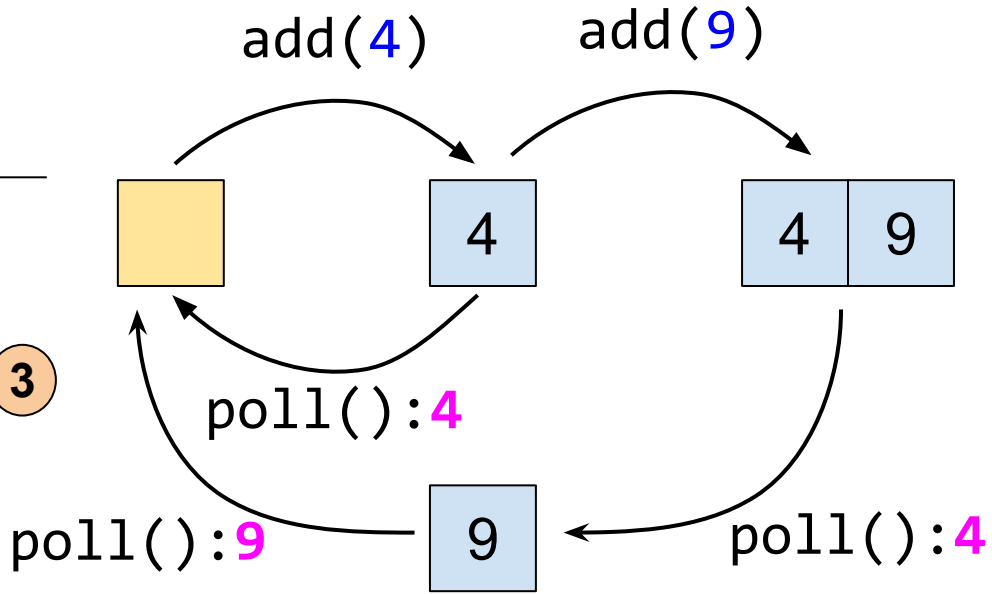


Верификация с помощью LTS

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

1 `q.add(4)`	
	`q.add(9)` 2
	`q.poll(): 4` 3
4 `q.poll(): 9`	

Путь найден ⇒
результаты корректны

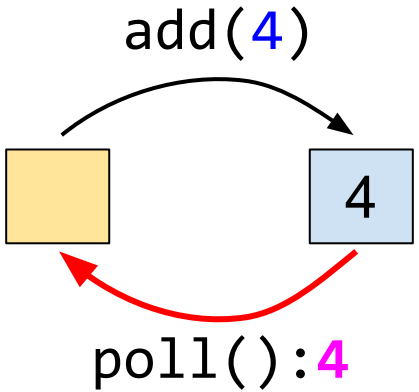


Построение LTS

- Строим лениво
- Используем последовательную спецификацию

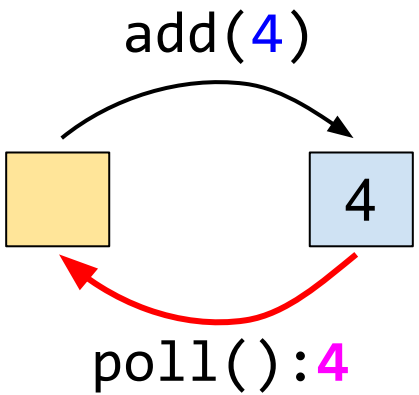
Построение LTS

- Строим лениво
- Используем последовательную спецификацию



Построение LTS

- Строим лениво
- Используем последовательную спецификацию
- Задаем эквивалентность состояний через определение `equals/hashcode`




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

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

Последовательная спецификация

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

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

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

Последовательная спецификация

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

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

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

Последовательная спецификация

```
class MyQueueTest {
    val q = MyQueue<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
}
```

Что если в структуре данных есть **блокирующие** методы?

Rendezvous Channels

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

`c.send(4)`	`c.receive(): suspended + 4`

`send` ждет **`receive`** и наоборот

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)
}
```

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

Rendezvous Channels

Client 1

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

Client 2

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

Засыпает, пока какой-нибудь **Client** не отправит **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)
```

Client 2

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

Worker


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

1

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

Rendezvous Channels

Rendezvous!
send будит **receive**

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)
```

Client 2

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

Worker

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

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

Rendezvous Channels

Client 1

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

Client 2

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

Worker

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

Засыпает, пока **Worker** не вызовет **receive**

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

Rendezvous Channels

Client 1

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

Client 2


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

Worker

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

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

Rendezvous Channels

Client 1

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

Client 2

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

Worker

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

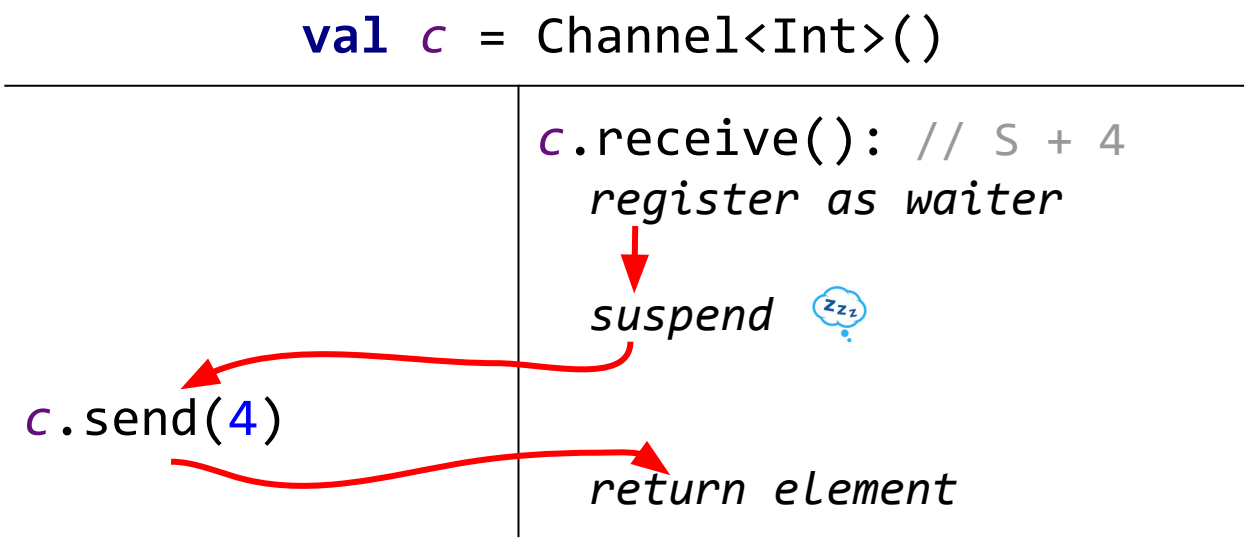
Rendezvous!
receive будит **send**

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

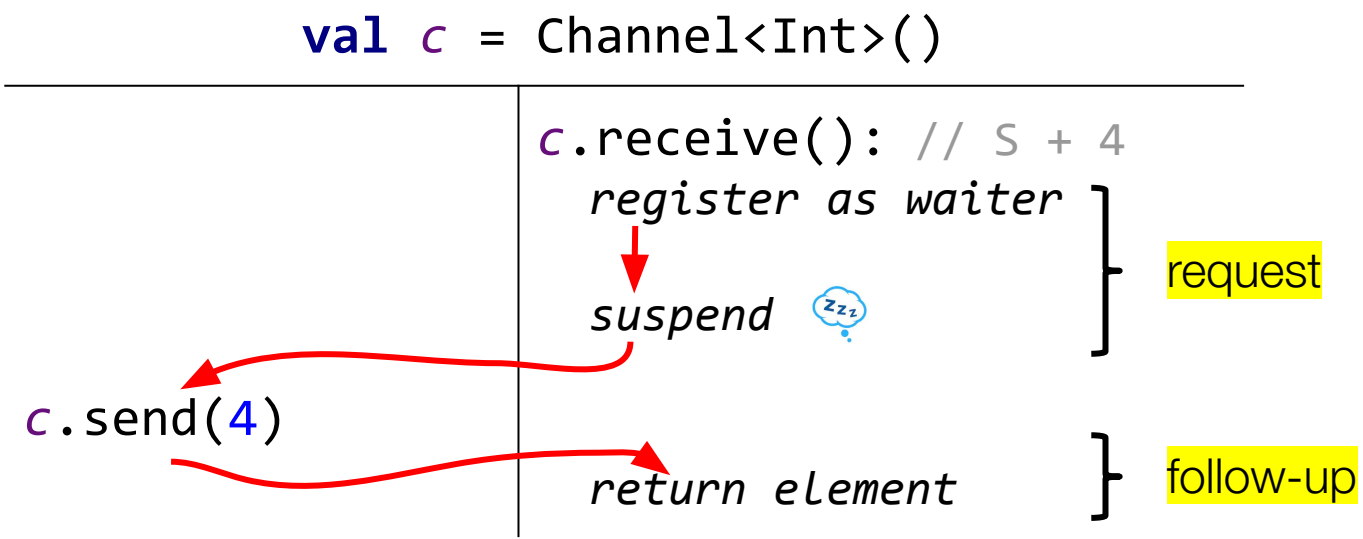
```
val c = Channel<Int>()
```

`c.send(4)`	`c.receive(): suspended + 4`

Нелинеаризуемо


val c = Channel<Int>()
c.receive(): // S + 4
register as 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 LTS

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

`c.send(4)`	`c.receive(): suspended + 4`

Dual LTS

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

`c.send(4)`	`c.receive(): suspended + 4` $receive^{req}():s_1$ $receive^{fu}(s_1):4$

Dual LTS

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

`c.send(4)`	`c.receive(): suspended` + 4 $receive^{req}():s_1$ $receive^{fu}(s_1):4$

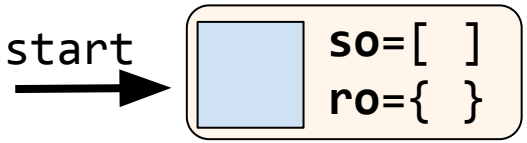
Отношение эквивалентности между состояниями LTS:

1. **equals/hashCode** на состоянии структуры данных
2. Список уснувших операций (**so**)
3. Множество разбуженных операций (**ro**)

Dual LTS

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

`c.send(4)`	`c.receive(): suspended` + 4
	$receive^{req}():s_1$
	$receive^{fu}(s_1):4$



Отношение эквивалентности между состояниями LTS:

1. **equals/hashCode** на состоянии структуры данных
2. Список уснувших операций (**so**)
3. Множество разбуженных операций (**ro**)

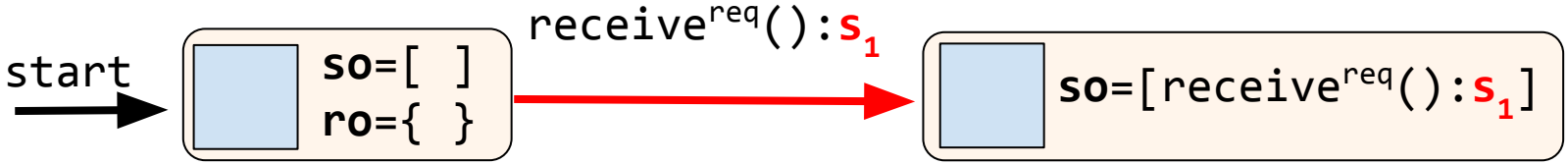
Dual LTS

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

`c.send(4)`	`c.receive(): suspended` + 4
	$receive^{req}():s_1$
	$receive^{fu}(s_1):4$

Отношение эквивалентности между состояниями LTS:

1. **equals/hashCode** на состоянии структуры данных
2. Список уснувших операций (**so**)
3. Множество разбуженных операций (**ro**)



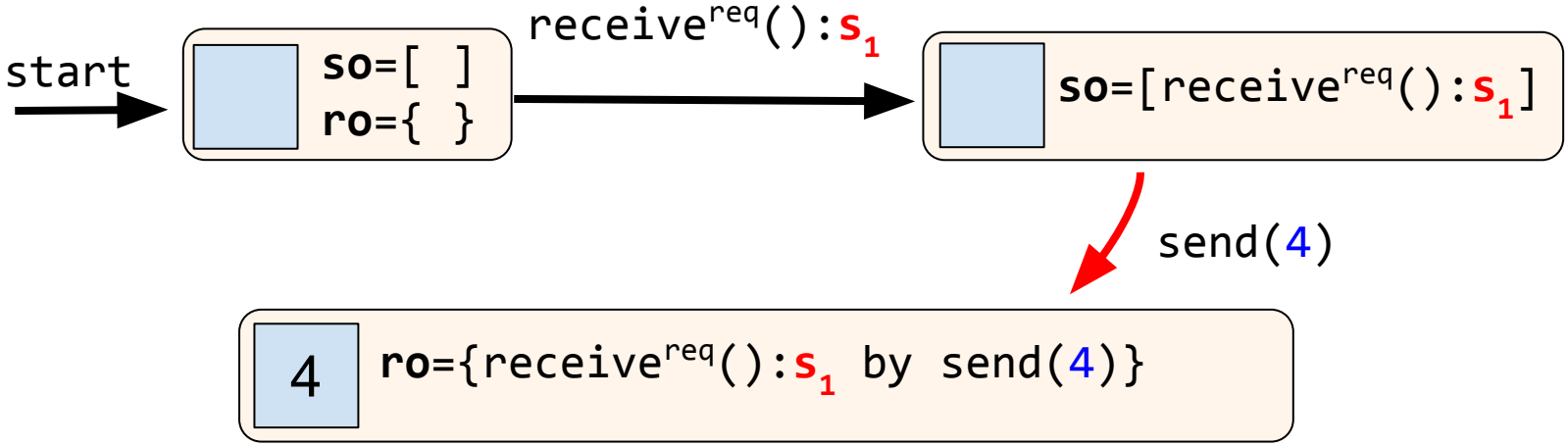
Dual LTS

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

`c.send(4)`	`c.receive(): suspended + 4`
	$receive^{req}():s_1$
	$receive^{fu}(s_1):4$

Отношение эквивалентности между состояниями LTS:

1. **equals/hashCode** на состоянии структуры данных
2. Список уснувших операций (**so**)
3. Множество разбуженных операций (**ro**)



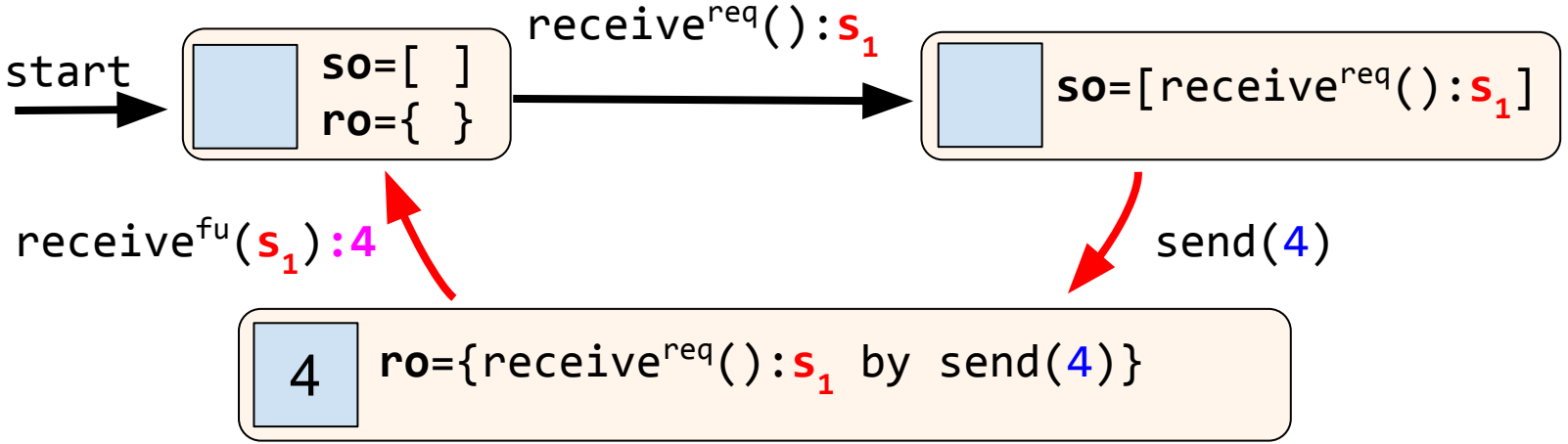
Dual LTS

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

`c.send(4)`	`c.receive(): suspended + 4`
	$receive^{req}():s_1$
	$receive^{fu}(s_1):4$

Отношение эквивалентности между состояниями LTS:

1. **equals/hashCode** на состоянии структуры данных
2. Список уснувших операций (**so**)
3. Множество разбуженных операций (**ro**)



Rendezvous Channel Test

```
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
}
```

Lincheck на практике

Параметры операций

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

Откуда берутся параметры операций?

Генерация параметров

Генераторы параметров

```
class MyQueueTest {
    ...
    @Operation fun add(@Param(gen = IntGen::class,
                              conf = "-10:10") x: Int) = ...
    @Operation fun poll() = ...
}
```

Генерация параметров

```
class MyQueueTest {
    ...
    @Operation fun add(@Param(gen = IntGen::class,
                              conf = "-10:10") x: Int) = ...
    @Operation fun poll() = ...
}
```

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

Свой генератор параметров

Run once

```
class MyQueueTest {
    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() = ...
}
```

Run once

```
class MyQueueTest {
    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() = ...
}
```

По контракту очереди **`close`** можно вызвать только **один раз**

Run once

```
class MyQueueTest {
    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() = ...
}
```

По контракту очереди **close** можно вызвать только **один раз**

Single reader / writer

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

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

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

Single reader / writer

```
class MyQueueTest {
    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   |
```

Single reader / writer

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

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

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

Такой сценарий даст некорректные результаты для **single-consumer** очереди

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

Single reader / writer

```
@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() = ...
}
```

Single reader / writer

```
@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() = ...
}
```

Задание своих сценариев

```
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)
    }
  }
}
```

Задание своих сценариев

```
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)
    }
  }
}
```

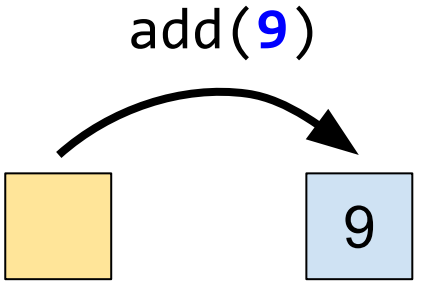
```
class MyQueueTest {
  ...
  @Test fun test() = StressOptions()
    .addCustomScenario(s)
    .check(this::class)
}
```

Результат теста с lincheck

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

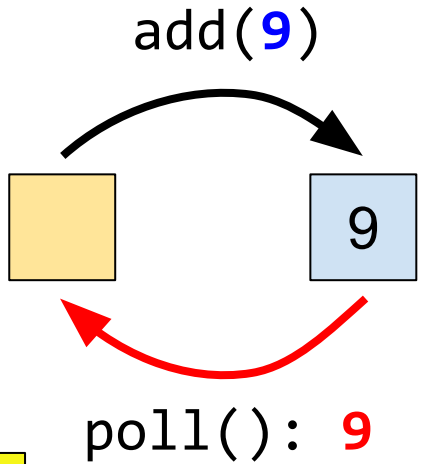
Результат теста с lincheck

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



Результат теста с lincheck

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



Как понять, почему произошла ошибка?

Минимизация сценария с ошибкой

```
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 пытается жадно удалять операции из сценария:

см. `Options.minimizeFailedScenario(..)`

- **Lincheck** позволяет легко тестировать многопоточные структуры данных
- Поддерживает разные контракты корректности
  - single reader/writer, dual data structures
  - serializability, quiescent consistency
- Используем в Kotlin Coroutines

Вопросы?

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