

# Тестирование многопоточных структур данных

Никита Коваль

[nkoval@devexperts.com](mailto:nkoval@devexperts.com)

[twitter.com/nkoval\\_](https://twitter.com/nkoval_)



# О себе

- Инженер-исследователь в лаборатории dxLab, Devexperts
- Преподаю курс по многопоточному программированию в ИТМО
- Заканчиваю магистратуру ИТМО



# Concurrency – это сложно!

Concurrency – это сложно!

... но тестировать многопоточные  
алгоритмы не легче!

# Счётчик

```
class Counter {  
    private int x;  
  
    int incAndGet() {  
        return ++x;  
    }  
}
```

Вызовем `incAndGet` параллельно

# Счётчик: исполняем

```
val c = new Counter();
```

---

```
c.incAndGet():  
  int t1 = x;  
  t1 = t1 + 1;  
  x = t1;  
  return t1;
```

```
c.incAndGet():  
  int t2 = x;  
  t2 = t2 + 1;  
  x = t2;  
  return t2;
```

# Счётчик: исполняем

```
val c = new Counter();
```

```
c.incAndGet():
```

```
① int t1 = x; // 0
```

```
③ t1 = t1 + 1; // 1
```

```
④ x = t1; // 1
```

```
⑧ return t1; // 1
```

```
c.incAndGet():
```

```
② int t2 = x; // 0
```

```
⑤ t2 = t2 + 1; // 1
```

```
⑥ x = t2; // 1
```

```
⑦ return t2; // 1
```

# Счётчик: исполняем

```
val c = new Counter();
```

```
c.incAndGet():
```

```
① int t1 = x; // 0
```

```
③ t1 = t1 + 1; // 1
```

```
④ x = t1; // 1
```

```
⑧ return t1; // 1
```

```
c.incAndGet():
```

```
② int t2 = x; // 0
```

```
⑤ t2 = t2 + 1; // 1
```

```
⑥ x = t2; // 1
```

```
⑦ return t2; // 1
```

**НЕКОРРЕКТНО**



# Счётчик: что ожидаем

```
val c = new Counter();
```

---

```
① c.incAndGet(); // 1
```

```
② c.incAndGet(); // 2
```

ИЛИ

```
val c = new Counter();
```

---

```
② c.incAndGet(); // 2
```

```
① c.incAndGet(); // 1
```

# Счётчик: что ожидаем

```
val c = new Counter();
```

```
1 c.incAndGet(); // 1
```

```
2 c.incAndGet(); // 2
```

**АТОМАРНО**

```
val c = new Counter();
```

```
2 c.incAndGet(); // 2
```

```
1 c.incAndGet(); // 1
```

# Корректность

- Последовательный алгоритм
  - Последовательная спецификация операций
- Многопоточный алгоритм
  - Линеаризуемость (ака атомарность)

# Линеаризуемость

Исполнение линеаризуемо  $\Leftrightarrow$   $\exists$  эквивалентное  
последовательное исполнение *(на самом деле чуть сложнее)*

# Линеаризуемость

Исполнение линеаризуемо  $\Leftrightarrow$   $\exists$  эквивалентное последовательное исполнение *(на самом деле чуть сложнее)*

```
val m = new ConcurrentHashMap<Int, Int>();
```

```
m.put(4, 5);
```

```
m.get(2);
```

```
m.put(1, 1);
```

```
m.get(4);
```

```
m.put(1, 3);
```

```
m.put(2, 4);
```

```
m.get(4);
```

```
m.get(1);
```

# Линеаризуемость

Исполнение линеаризуемо  $\Leftrightarrow \exists$  эквивалентное последовательное исполнение *(на самом деле чуть сложнее)*

```
val m = new ConcurrentHashMap<Int, Int>();
```

```
m.put(4, 5); // 0  
m.get(2); // 4  
m.put(1, 1); // 3
```

```
m.get(4); // 0  
m.put(1, 3); // 0
```

```
m.put(2, 4); // 0  
m.get(4); // 5  
m.get(1); // 1
```

# Линеаризуемость

Исполнение линеаризуемо  $\Leftrightarrow \exists$  эквивалентное последовательное исполнение *(на самом деле чуть сложнее)*

```
val m = new ConcurrentHashMap<Int, Int>();
```

```
2 m.put(4, 5); // 0
```

```
6 m.get(2); // 4
```

```
7 m.put(1, 1); // 3
```

```
1 m.get(4); // 0
```

```
5 m.put(1, 3); // 0
```

```
3 m.put(2, 4); // 0
```

```
4 m.get(4); // 5
```

```
8 m.get(1); // 1
```

# JCStress: счётчик

```
@JCStressTest
@Outcome(id = "1, 2", expect = Expect.ACCEPTABLE)
@Outcome(id = "2, 1", expect = Expect.ACCEPTABLE)
@State
public class CounterTest {
    Counter c = new Counter();

    @Actor
    public void actor1(IntResult2 r) { r.r1 = c.incAndGet(); }

    @Actor
    public void actor2(IntResult2 r) { r.r2 = c.incAndGet(); }
}
```



# JCStress: счётчик

```
@JCStressTest
@Outcome(id = "1, 2", expect = Expect.ACCEPTABLE)
@Outcome(id = "2, 1", expect = Expect.ACCEPTABLE)
@State
public class CounterTest {
    Counter c = new Counter();

    @Actor
    public void actor1(IntResult2 r) { r.r1 = c.incAndGet(); }

    @Actor
    public void actor2(IntResult2 r) { r.r2 = c.incAndGet(); }
}
```

# JCStress: счётчик

```
@JCStressTest
@Outcome(id = "1, 2", expect = Expect.ACCEPTABLE)
@Outcome(id = "2, 1", expect = Expect.ACCEPTABLE)
@State
public class CounterTest {
    Counter c = new Counter();

    @Actor
    public void actor1(IntResult2 r) { r.r1 = c.incAndGet(); }

    @Actor
    public void actor2(IntResult2 r) { r.r2 = c.incAndGet(); }
}
```

# JCStress: счётчик

```
@JCStressTest
@Outcome(id = "1, 2", expect = Expect.ACCEPTABLE)
@Outcome(id = "2, 1", expect = Expect.ACCEPTABLE)
@State
public class CounterTest {
    Counter c = new Counter();

    @Actor
    public void actor1(IntResult2 r) { r.r1 = c.incAndGet(); }

    @Actor
    public void actor2(IntResult2 r) { r.r2 = c.incAndGet(); }
}
```

# JCStress: счётчик

```
@JCStressTest
@Outcome(id = "1, 2", expect = Expect.ACCEPTABLE)
@Outcome(id = "2, 1", expect = Expect.ACCEPTABLE)
@State
public class CounterTest {
    Counter c = new Counter();

    @Actor
    public void actor1(IntResult2 r) { r.r1 = c.incAndGet(); }

    @Actor
    public void actor2(IntResult2 r) { r.r2 = c.incAndGet(); }
}
```

# JCStress: счётчик

```
@JCStressTest
@Outcome(id = "1, 2", expect = Expect.ACCEPTABLE)
@Outcome(id = "2, 1", expect = Expect.ACCEPTABLE)
@State
public class CounterTest {
    Counter c = new Counter();

    @Actor
    public void actor1(IntResult2 r) { r.r1 = c.incAndGet(); }

    @Actor
    public void actor2(IntResult2 r) { r.r2 = c.incAndGet(); }
}
```

# JCStress: счётчик

```
@JCStressTest
@Outcome(id = "1, 2", expect = Expect.ACCEPTABLE)
@Outcome(id = "2, 1", expect = Expect.ACCEPTABLE)
@State
public class CounterTest {
    Counter c = new Counter();

    @Actor
    public void actor1(IntResult1 r) { r.r1 = c.incAndGet(); }

    @Actor
    public void actor2(IntResult2 r) { r.r2 = c.incAndGet(); }
}
```

Запустим

# JCStress: счётчик

```
@JCStressTest
@Outcome(id = "1, 2", expect = Expect.ACCEPTABLE)
@Outcome(id = "2, 1", expect = Expect.ACCEPTABLE)
@State
public class Counter {
    @Actor
    public void actor1() {
        @Actor
        public void actor2(IntResult2 r) { r.r2 = c.incAndGet(); }
    }
}
```

State	Occurrences	Expectation
1, 1	5,003,109	FORBIDDEN
1, 2	3,830,773	ACCEPTABLE
2, 1	2,396,268	ACCEPTABLE



# JCStress: тестируем ConcurrentHashMap

## Сценарии:

- Одновременное добавление и удаление по ключу
- Параллельное обновление одного bucket-а
- Параллельное обновление разных bucket-ов
- Параллельное добавление/удаление и перехеширование
- ...

## Возможные исполнения:

- 2 операции x 2 потока  $\Rightarrow$  6 последовательных исполнений
- 2 операции x 3 потока  $\Rightarrow$  **90 (!)** последовательных исполнений



# JCStress: тестируем ConcurrentHashMap

## Сценарии:

- Одновременное добавление и удаление по ключу
- Параллельное обновление одного значения
- Параллельное обновление
- Параллельное добавление
- ...



**A-A-A-A!**

## Возможные результаты:

- 2 операции x 2 потока ⇒ 4 последовательных исполнений
- 2 операции x 3 потока ⇒ **90 (!)** последовательных исполнений

# JCStress: резюмируем

- Проверяет корректность заданного сценария
- Универсален: не только линеаризуемость
  
- Нужно продумать сценарии
  - Это сложно
- Нужно описать корректные исходы
  - Муторно и долго

# JCStress: резюмируем

Автоматизировать?

- Нужно продумать сценарии
  - Это сложно
- Нужно описать корректные исходы
  - Мучительно и долго

# Идеальный инструмент

## 1. Задать состояние

```
public class CHMTest {  
    private Map<Int, Int> m =  
        new ConcurrentHashMap<>();  
  
}
```

# Идеальный инструмент

1. Задать состояние
2. Задать операции

```
public class CHMTest {  
    private Map<Int, Int> m =  
        new ConcurrentHashMap<>();  
  
    @Operation  
    public int put(Int k, Int v) {  
        return m.put(k, v);  
    }  
  
    @Operation  
    public int get(Int k) {  
        return m.get(k);  
    }  
  
}
```

# Идеальный инструмент

1. Задать состояние
2. Задать операции
3. Проверить на  
линеаризуемость

Сделай хорошо!

```
public class CHMTest {
    private Map<Int, Int> m =
        new ConcurrentHashMap<>();

    @Operation
    public int put(Int k, Int v) {
        return m.put(k, v);
    }

    @Operation
    public int get(Int k) {
        return m.get(k);
    }

    public void test() { check(this); }
}
```

# Идеальный инструмент

1. Задать состояние
2. Задать операции
3. Проверить на  
линеаризуемость

Сделай хорошо!

```
public class CHMTest {  
    private Map<Int, Int> m =  
        new ConcurrentHashMap<>();  
  
    @Operation  
    public int put(Int k, Int v) {  
        return m.put(k, v);  
    }  
  
    @Operation  
    public int get(Int k) {  
        return m.get(k);  
    }  
  
    public void test() { check(this); }  
}
```

# Lin-Check

- Lin-Check = **Linearization Checker**
  - <https://github.com/Devexperts/lin-check>
- Генерирует сценарии выполнения, запускает и проверяет на линеаризуемость



# Lin-Check: «Hello World»

```
@StressCTest
public class CounterTest {
    private Counter counter;

    @Reset
    public void reload() { counter = new Counter(); }

    @Operation
    public int incAndGet() {
        return counter.incAndGet();
    }

    @Test // JUnit test
    public void test() { LinChecker.check(CounterTest.class);
}
```

# Lin-Check: «Hello World»

```
@StressCTest
public class CounterTest {
    private Counter counter;
```

```
@Reset
```

```
public void reload() { counter = new Counter(); }
```

```
@Operation
```

```
public int incAndGet() {
    return counter.incAndGet();
}
```

```
@Test // JUnit test
```

```
public void test() { LinChecker.check(CounterTest.class); }
```

Начальное состояние



# Lin-Check: «Hello World»

```
@StressCTest
public class CounterTest {
    private Counter counter;
```

```
    @Reset
    public void reload() { counter = new Counter(); }
```

```
    @Operation
    public int incAndGet() {
        return counter.incAndGet();
    }
```

```
    @Test // JUnit test
    public void test() { LinChecker.check(CounterTest.class); }
```

Операции



# Lin-Check: «Hello World»

```
@StressCTest
public class CounterTest {
    private Counter counter;

    @Reset
    public void reload() { counter = new Counter(); }

    @Operation
    public int incAndGet() {
        return counter.incAndGet();
    }

    @Test // JUnit test
    public void test() { LinChecker.check(CounterTest.class); }
```

Проверка на  
линеаризуемость



# Lin-Check: «Hello World»

```
@StressCTest
public class CounterTest {
    private Counter counter;

    @Reset
    public void reload() { counter = new Counter(); }

    @Operation
    public int incAndGet() {
        return counter.incAndGet();
    }

    @Test // JUnit test
    public void test() { LinChecker.check(CounterTest.class);
}
```



# Lin-Check: «Hello World»

```
@StressCTest
public class CounterTest {
    private Counter counter;

    @Reset
    public void reload() { counter = new Counter(); }

    @Operation
    public int incAndReturn() {
        return counter.inc();
    }

    @Test // JUnit test
    public void test() { LinChecker.check(CounterTest.class);
}
```

Запустим



# Lin-Check: «Hello World»

**= Iteration 1 / 200 =**

**Actors per thread:**

`[incAndGet(), incAndGet(), incAndGet()]`

`[incAndGet(), incAndGet(), incAndGet(), incAndGet()]`

**Non-linearizable execution:**

`[2, 3, 5]`

`[1, 3, 4, 6]`

**Possible linearizable executions:**

`[3, 6, 7]`

`[1, 2, 4, 5]`

`[1, 2, 5]`

`[3, 4, 6, 7]`

`...`

# Lin-Check: «Hello World»

= Iteration 1 / 200 =

## Actors per thread:

[incAndGet(), incAndGet(), incAndGet()]

[incAndGet(), incAndGet(), incAndGet(), incAndGet()]

## Non-linearizable execution:

[2, 3, 5]

[1, 3, 4, 6]

## Possible linearizable executions:

[3, 6, 7]

[1, 2, 4, 5]

[1, 2, 5]

[3, 4, 6, 7]

...

Сценарий выполнения



# Lin-Check: «Hello World»

= Iteration 1 / 200 =

Actors per thread:

```
[incAndGet(), incAndGet(), incAndGet()]
```

```
[incAndGet(), incAndGet(), incAndGet(), incAndGet()]
```

**Non-linearizable execution:**

```
[2, 3, 5]
```

```
[1, 3, 4, 6]
```

Possible linearizable executions:

```
[3, 6, 7]
```

```
[1, 2, 4, 5]
```

```
[1, 2, 5]
```

```
[3, 4, 6, 7]
```

```
...
```

Не могут два incAndGet  
вернуть одно и то же

# Lin-Check: чуть сложнее

```
@<Strategy>CTest
@Param(name = "key", gen = IntGen.class, conf = "1:3")
@Param(name = "value", gen = IntGen.class)
public class CHMTest {
    private Map<Integer, Integer> m;

    @Reset
    public void reload() { m = new ConcurrentHashMap<>(); }

    @Operation(params = {"key", "value"})
    public int put(Integer k, Integer v) { return m.put(k, v); }

    @Operation
    public int get(@Param(name = "key") Integer k) { return m.get(k); }

    @Test // JUnit test
    public void test() throws Exception { LinChecker.check(CHMTest.class); }
}
```

# Lin-Check: чуть сложнее

```
@<Strategy>CTest
@Param(name = "key", gen = IntGen.class, conf = "1:3")
@Param(name = "value", gen = IntGen.class)
public class CHMTest {
    private Map<Integer, Integer> m;

    @Reset
    public void reload() { m = new ConcurrentHashMap<>(); }

    @Operation(params = {"key", "value"})
    public int put(Integer k, Integer v) { return m.put(k, v); }

    @Operation
    public int get(@Param(name = "key") Integer k) { return m.get(k); }

    @Test // JUnit test
    public void test() throws Exception { LinChecker.check(CHMTest.class); }
}
```

Начальное состояние

# Lin-Check: чуть сложнее

```
@<Strategy>CTest
@Param(name = "key", gen = IntGen.class, conf = "1:3")
@Param(name = "value", gen = IntGen.class)
public class CHMTest {
    private Map<Integer, Integer> m;

    @Reset
    public void reload() { m = new ConcurrentHashMap<>(); }

    @Operation(params = {"key", "value"})
    public int put(Integer k, Integer v) { return m.put(k, v); }

    @Operation
    public int get(@Param(name = "key") Integer k) { return m.get(k); }

    @Test // JUnit test
    public void test() throws Exception { LinChecker.check(CHMTest.class); }
}
```

Операции

# Lin-Check: чуть сложнее

```
@<Strategy>CTest
@Param(name = "key", gen = IntGen.class, conf = "1:3")
@Param(name = "value", gen = IntGen.class)
public class CHMTest {
    private Map<Integer, Integer> m;

    @Reset
    public void reload() { m = new ConcurrentHashMap<>(); }

    @Operation(params = {"key", "value"})
    public int put(Integer k, Integer v) { return m.put(k, v); }

    @Operation
    public int get(@Param(name = "key") Integer k) { return m.get(k); }

    @Test // JUnit test
    public void test() throws Exception { LinChecker.check(CHMTest.class); }
}
```

Генераторы  
параметров операций

# Генератор параметров операций

```
/**  
 * The implementation of this interface is used  
 * to generate parameter values for  
 * {@link Operation operations}.  
 */  
public interface ParameterGenerator<T> {  
    T generate();  
}
```

# Lin-Check: чуть сложнее

```
@<Strategy>CTest
@Param(name = "key", gen = IntGen.class, conf = "1:3")
@Param(name = "value", gen = IntGen.class)
public class CHMTest {
    private Map<Integer, Integer> m;

    @Reset
    public void reload() { m = new ConcurrentHashMap<>(); }

    @Operation(params = {"key", "value"})
    public int put(Integer k, Integer v) { return m.put(k, v); }

    @Operation
    public int get(@Param(name = "key") Integer k) { return m.get(k); }

    @Test // JUnit test
    public void test() throws Exception { LinChecker.check(CHMTest.class); }
}
```

Генераторы  
параметров операций

# Lin-Check: чуть сложнее

```
@<Strategy>CTest
@Param(name = "key", gen = IntGen.class, conf = "1:3")
@Param(name = "value", gen = IntGen.class)
public class CHMTest {
    private Map<Integer, Integer> m;

    @Reset
    public void reload() { m = new ConcurrentHashMap<>(); }

    @Operation(params = {"key", "value"})
    public int put(Integer k, Integer v) { return m.put(k, v); }

    @Operation
    public int get(@Param(name = "key") Integer k) { return m.get(k); }

    @Test // JUnit test
    public void test() throws Exception { LinChecker.check(CHMTest.class); }
}
```

Проверка на  
линеаризуемость



# Lin-Check: чуть сложнее

```
@<Strategy>CTest
@Param(name = "key", gen = IntGen.class, conf = "1:3")
@Param(name = "value", gen = IntGen.class)
public class CHMTest {
    private Map<Integer, Integer> m;

    @Reset
    public void reload() { m = new ConcurrentHashMap<>(); }

    @Operation(params = {"key", "value"})
    public int put(Integer k, Integer v) { return m.put(k, v); }

    @Operation
    public int get(@Param(name = "key") Integer k) { return m.get(k); }

    @Test // JUnit test
    public void test() throws Exception { LinChecker.check(CHMTest.class); }
}
```

# Генерация сценариев выполнения

- Тестируем ConcurrentHashMap
- Генерируем байт-код для каждого из потоков (ASM)
- Сохраняем результат каждой операции

```
val m = new ConcurrentHashMap<Int, Int>();
```

```
// start thread #1  
r1[1] = m.put(4, 5);  
r1[2] = m.get(2);  
r1[3] = m.put(1, 1);  
// done
```

```
// start thread #2  
r2[0] = m.get(4);  
r2[1] = m.put(1, 3);  
// done
```

```
// start thread #3  
r3[0] = m.put(2, 4);  
r3[1] = m.get(4);  
r3[2] = m.get(1);  
// done
```

# Проверка корректности

- Хотим проверить, что результаты выполнения операций **(r1, r2, r3)** возможны в линеаризуемом исполнении

# Линеаризуемость

Исполнение линеаризуемо  $\Leftrightarrow$   $\exists$  эквивалентное  
последовательное исполнение *(на самом деле чуть сложнее)*

# Проверка корректности

- Хотим проверить, что результаты выполнения операций (**r1**, **r2**, **r3**) возможны в линеаризуемом исполнении
- Ищем перестановку операций, результат последовательного выполнения которой совпадает с результатом многопоточного исполнения (**r1**, **r2**, **r3**)
- Если такой перестановки нет, нашли ошибку

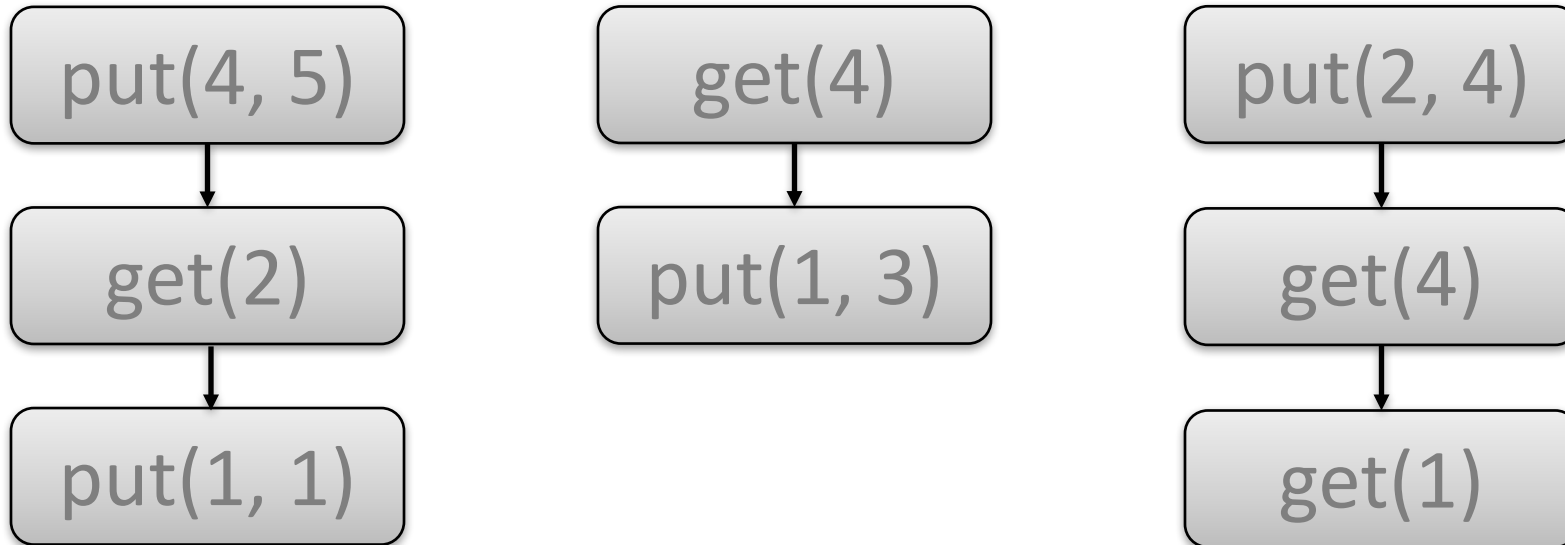
# Как запускать сценарий?

- Стресс режим
  - Запускаем потоки одновременно, ждем завершения и смотрим на результат
  - Делаем это много раз (привет, JCSstress)
- Управляемое исполнение
  - Об этом позже

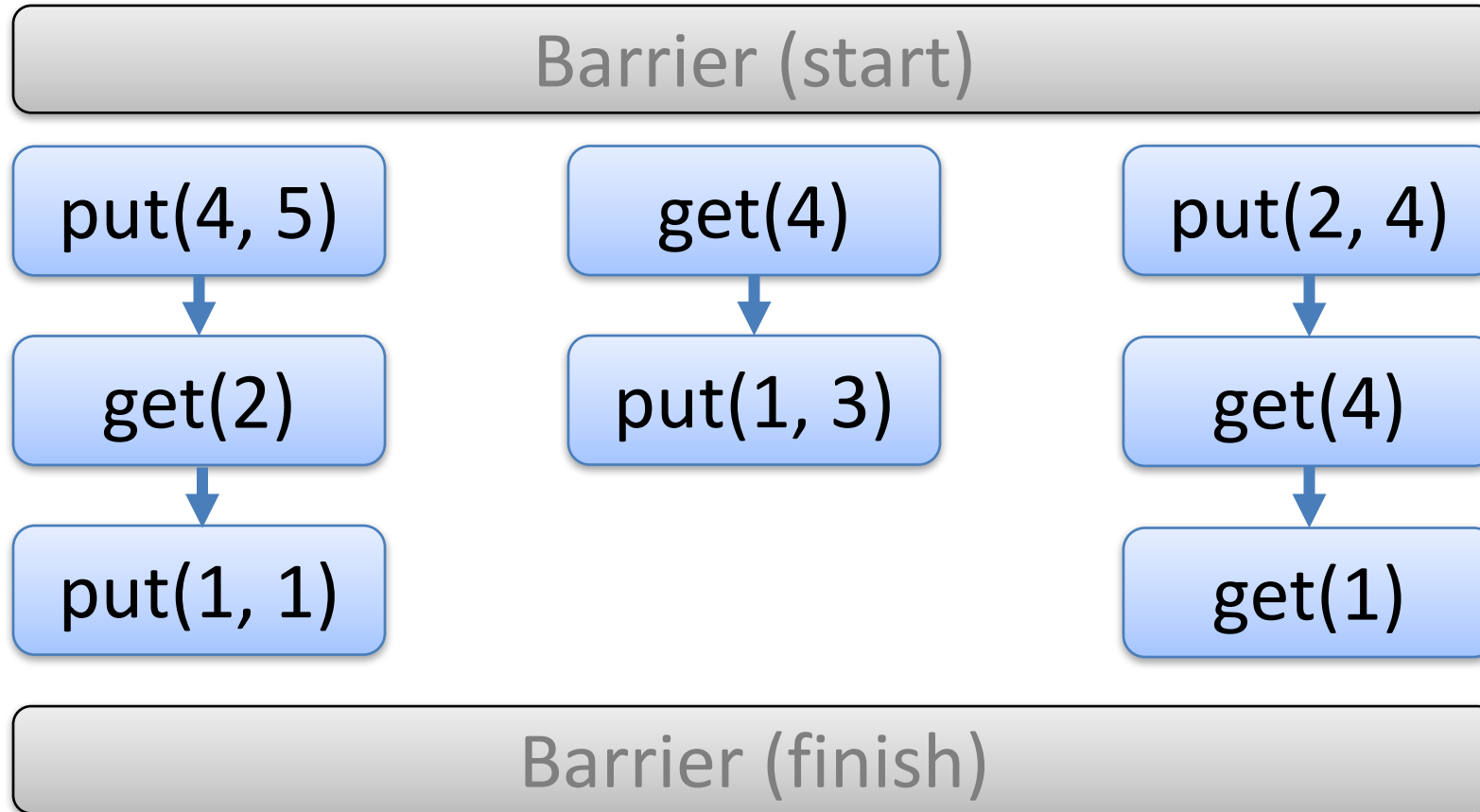
# Стресс-режим: пример

Barrier (start)

«Одновременный» старт



# Стресс-режим: пример

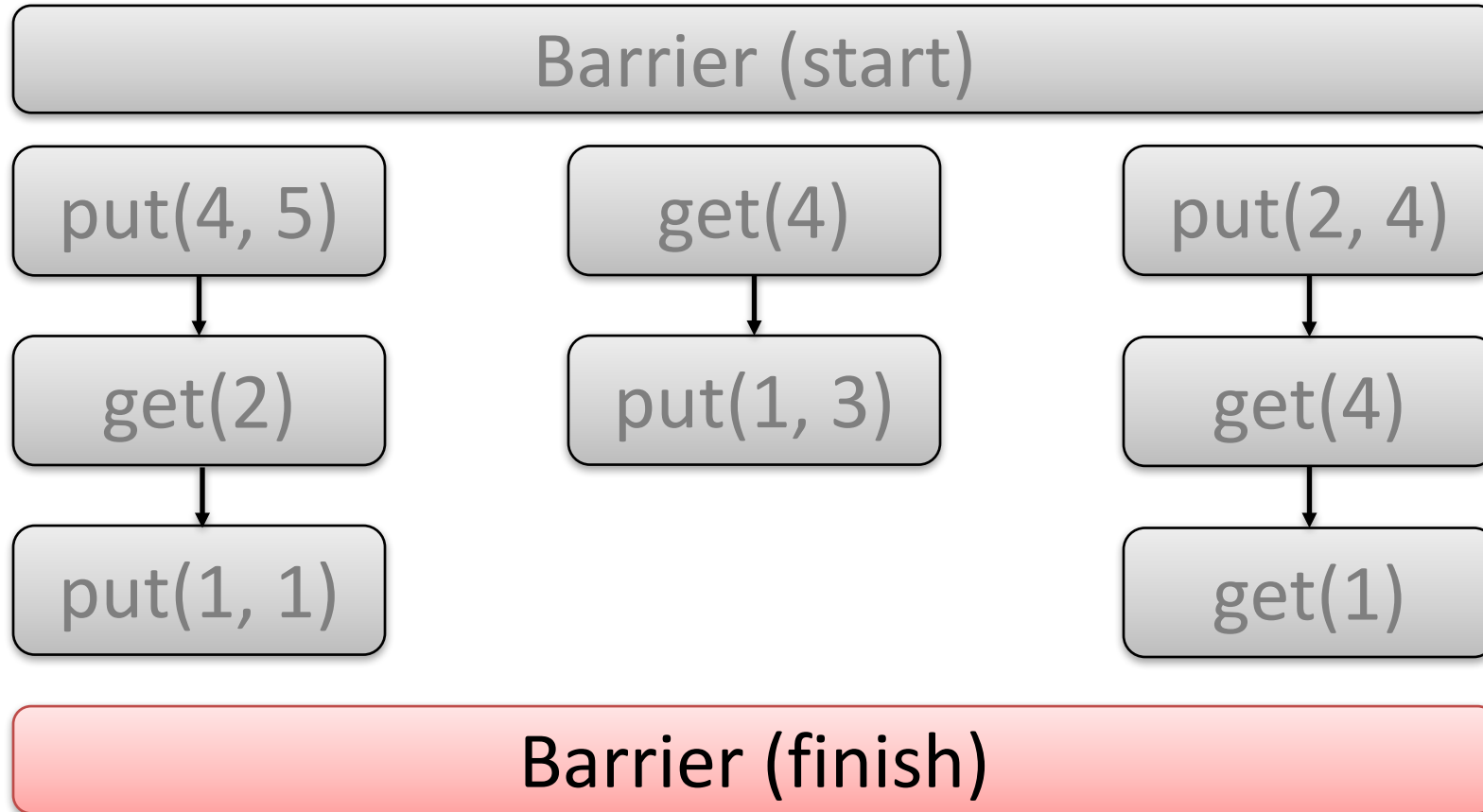


«Одновременный» старт

**Параллельное  
исполнение операций:  
происходят гонки,  
ошибки синхронизации  
и т.д.**



# Стресс-режим: пример



«Одновременный» старт

Параллельное  
исполнение операций:  
гонки, ошибки  
синхронизации и т.д.

Все закончили работу

# Стресс-режим: пример

Barrier (start)

put(4, 5)

get(4)

put(2, 4)

get(2)

put(1, 3)

get(4)

put(1, 1)

get(1)

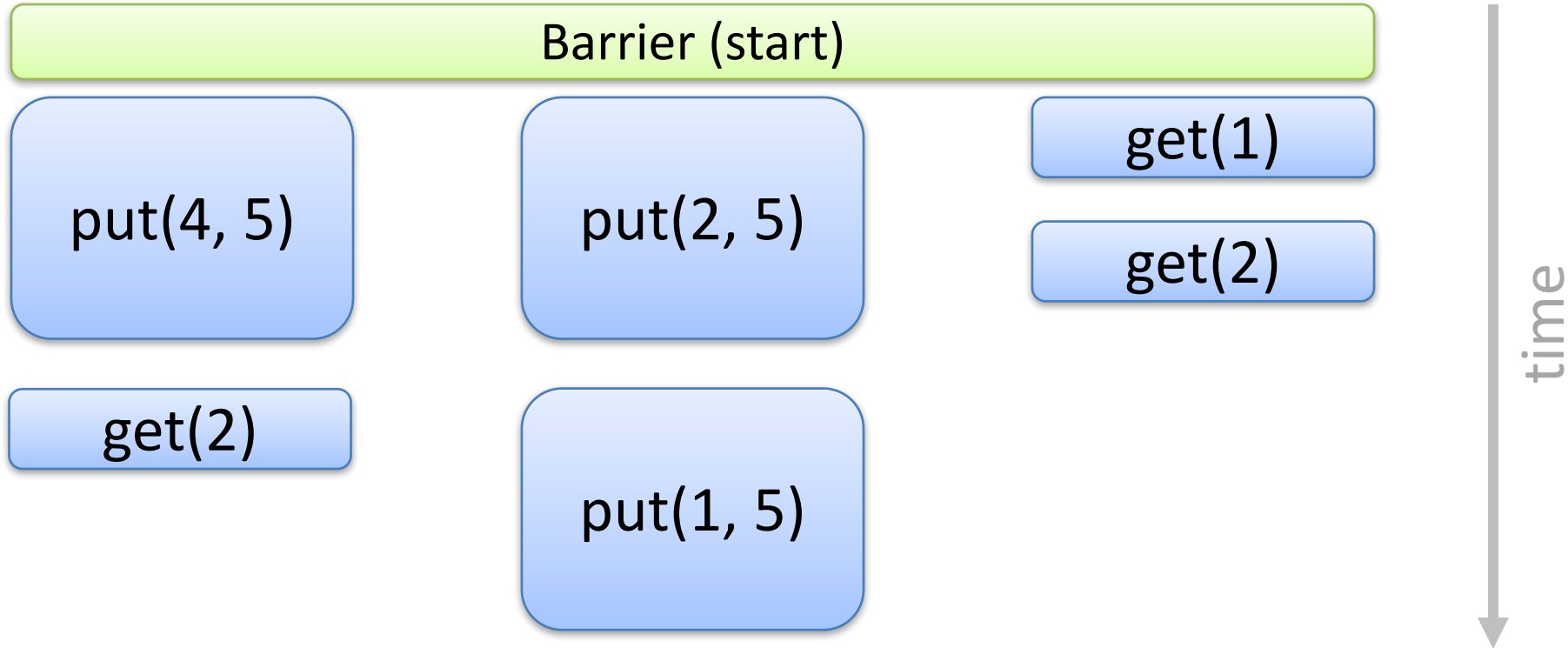
Barrier (finish)

«Одновременный» старт

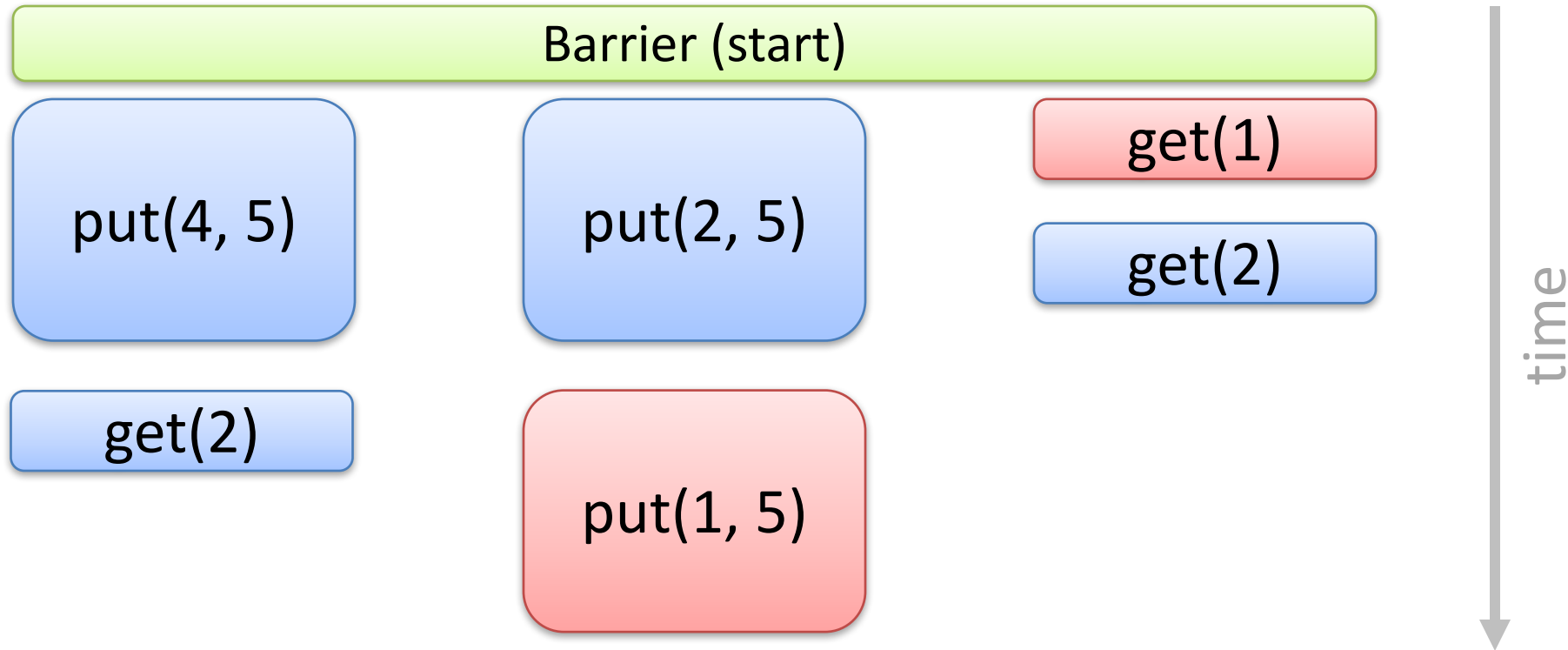
Параллельное  
исполнение операций:  
гонки, ошибки  
синхронизации и т.д.

Все закончили работу

# Стресс-режим: улучшаем покрытие

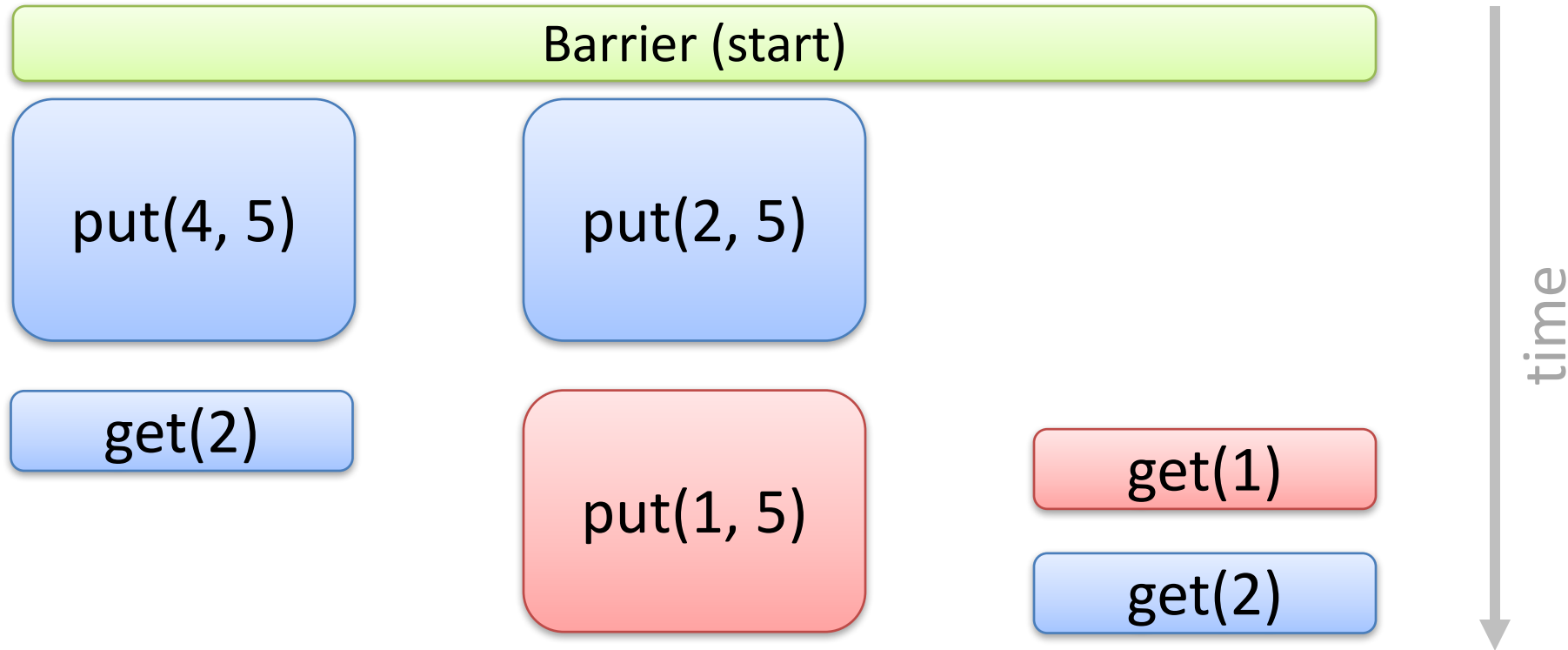


# Стресс-режим: улучшаем покрытие



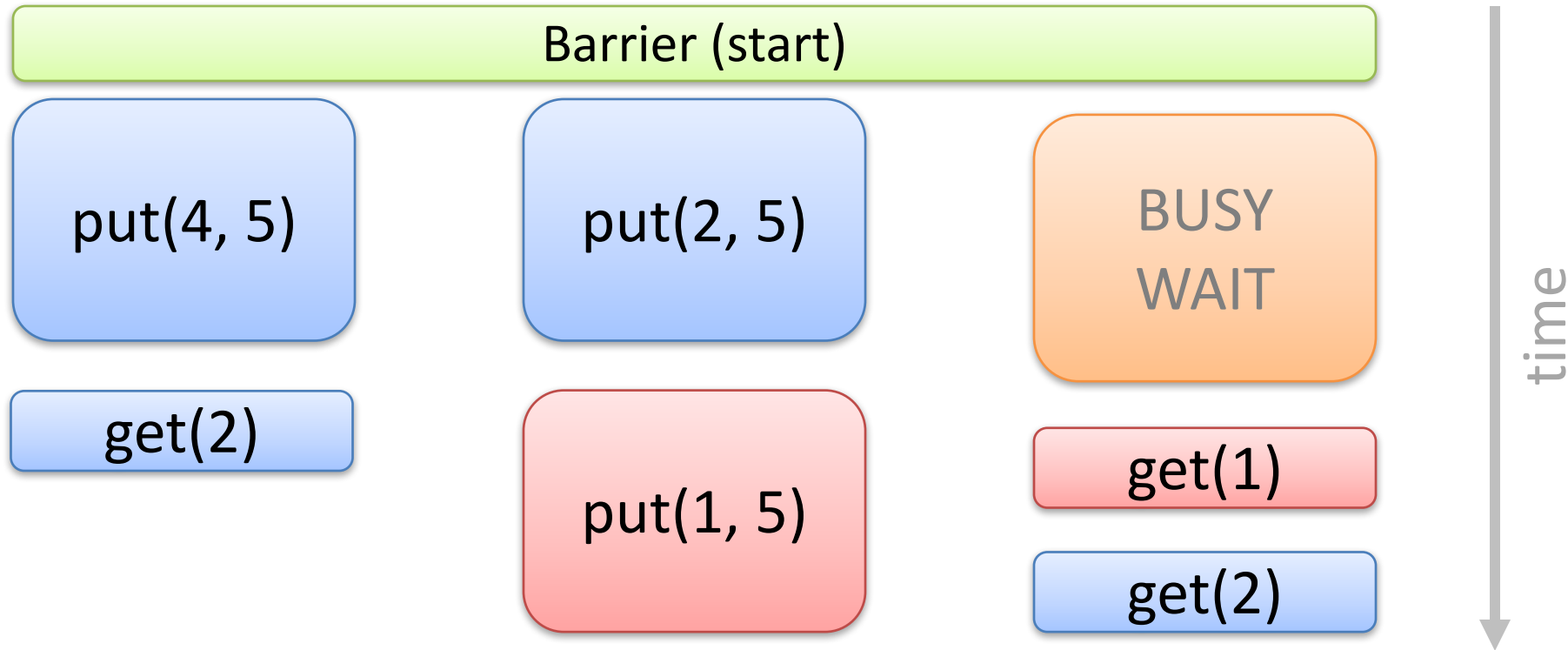
На практике почти всегда `get(1)` до `put(1, 5)`

# Стресс-режим: улучшаем покрытие



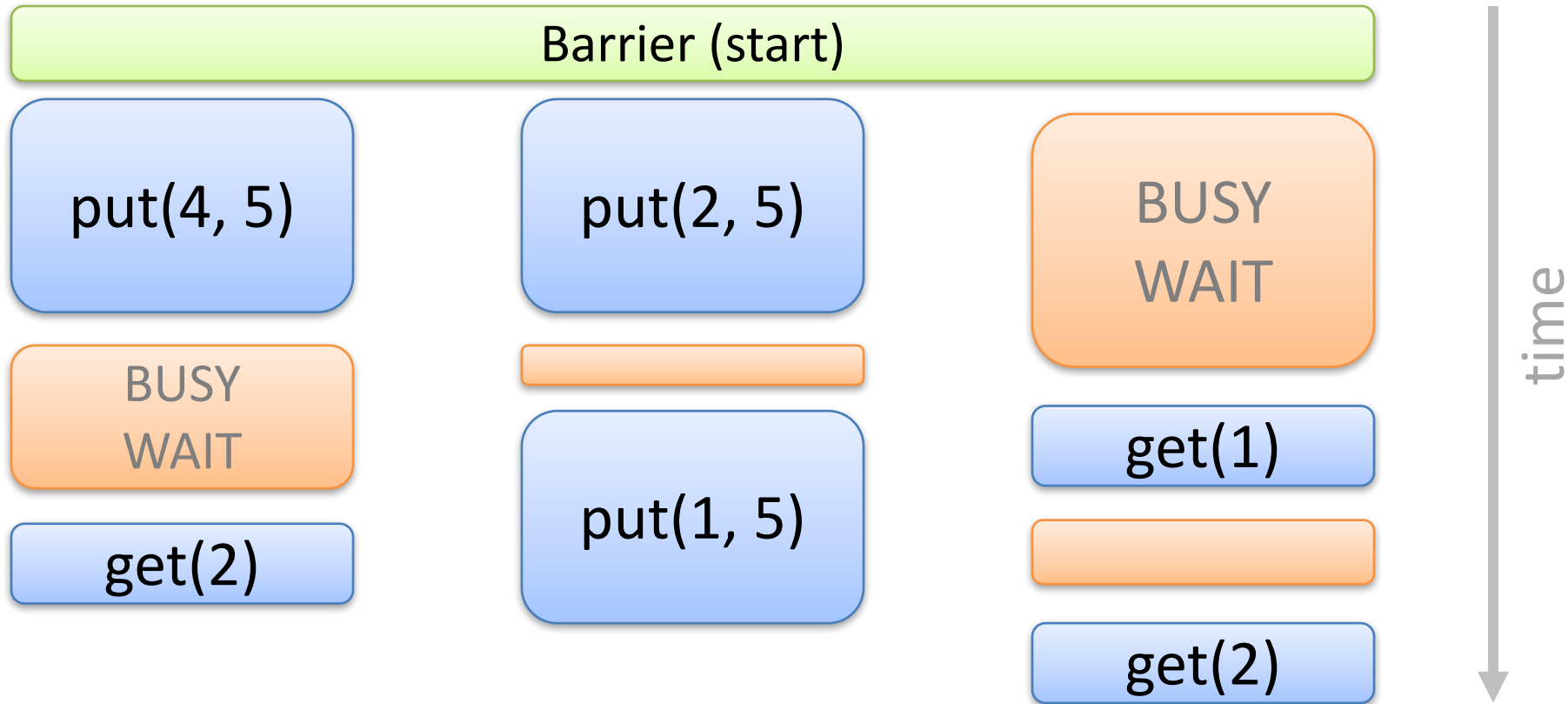
Хотелось бы получить такое исполнение

# Стресс-режим: улучшаем покрытие



Вставим бесполезную работу

# Стресс-режим: улучшаем покрытие



Везде... и случайную!

# Стресс-режим: резюмируем

- Добавляем задержки не только между операциями, но и внутри них
- Пробуем запускать с задержками разной величины



# Стресс-режим: резюмируем

- Добавляем задержки не только между операциями, но и внутри них
- Пробуем запускать с задержками разной величины

Как ещё улучшить  
покрытие?

# Управляемые стратегии

- Не будем надеяться на задержки, сами переключаем потоки в нужных местах
- Есть куча стратегий для переключения
- Позволяют находить логические ошибки
- Исполнение не содержит гонок

# One-switch стратегия

- Есть такой сценарий:

put(4, 5)

put(2, 5)

get(1)

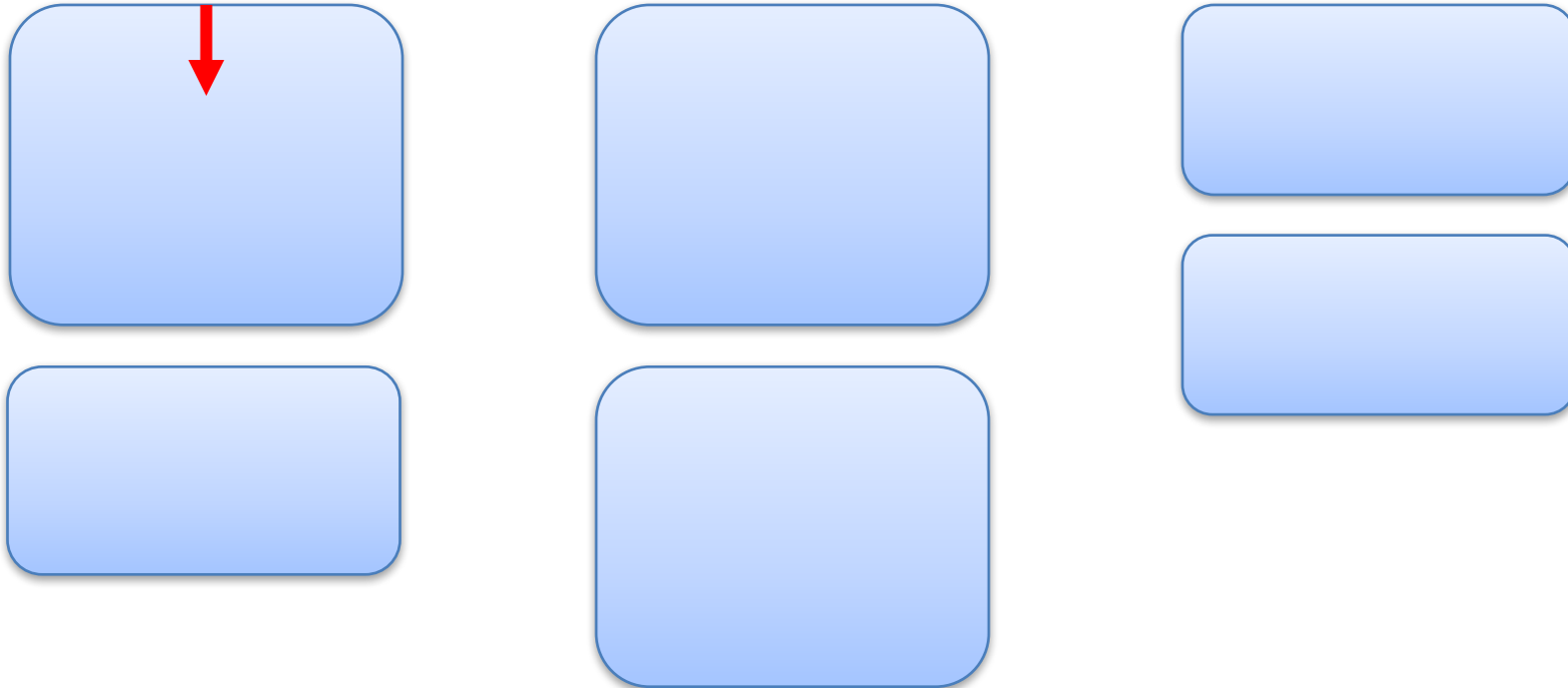
get(2)

put(1, 5)

get(2)

# One-switch стратегия

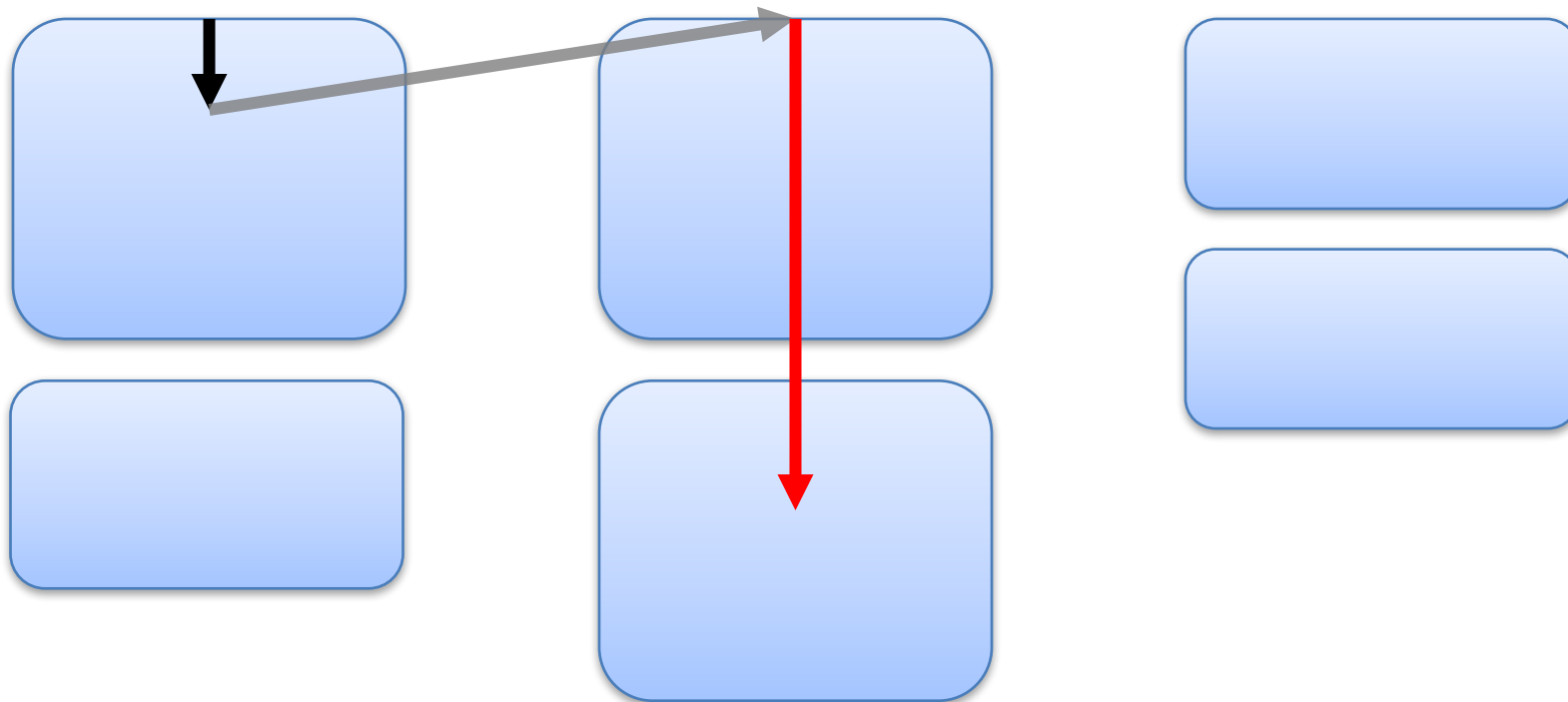
- Хочется уметь исполнять как-то так:



Выполним часть  
первого потока

# One-switch стратегия

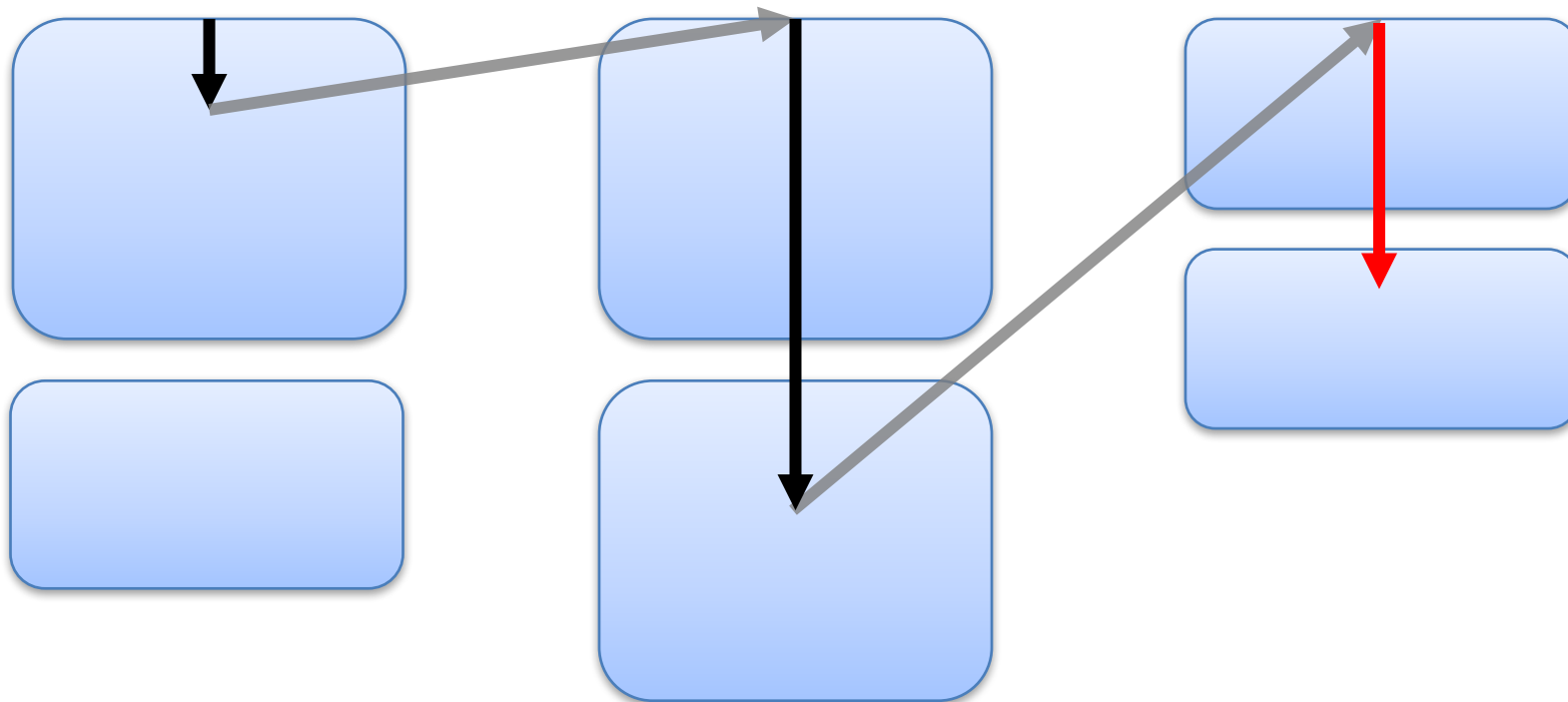
- Хочется уметь исполнять как-то так:



Переключимся на  
второй

# One-switch стратегия

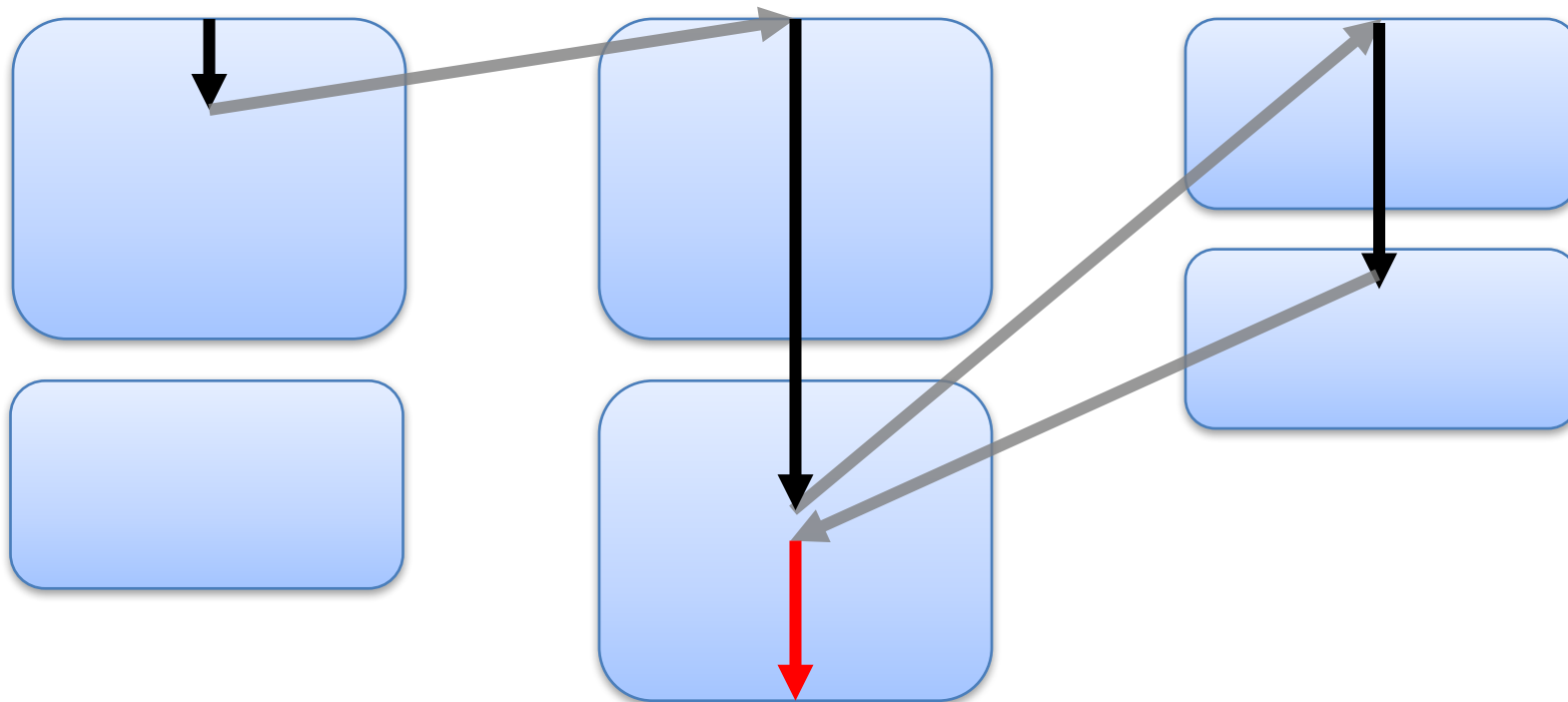
- Хочется уметь исполнять как-то так:



Теперь на третий

# One-switch стратегия

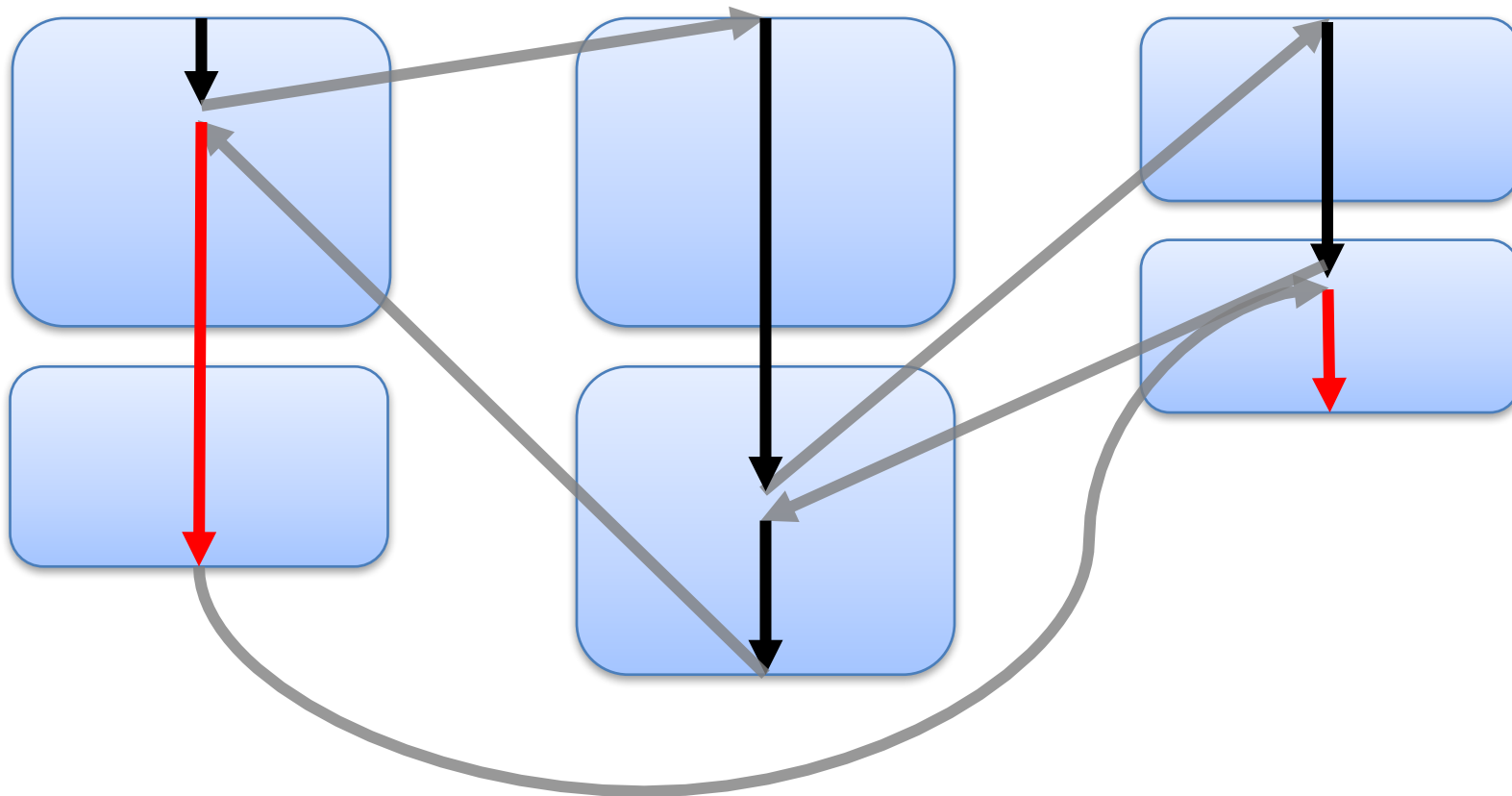
- Хочется уметь исполнять как-то так:



Обратно на второй

# One-switch стратегия

- Хочется уметь исполнять как-то так:

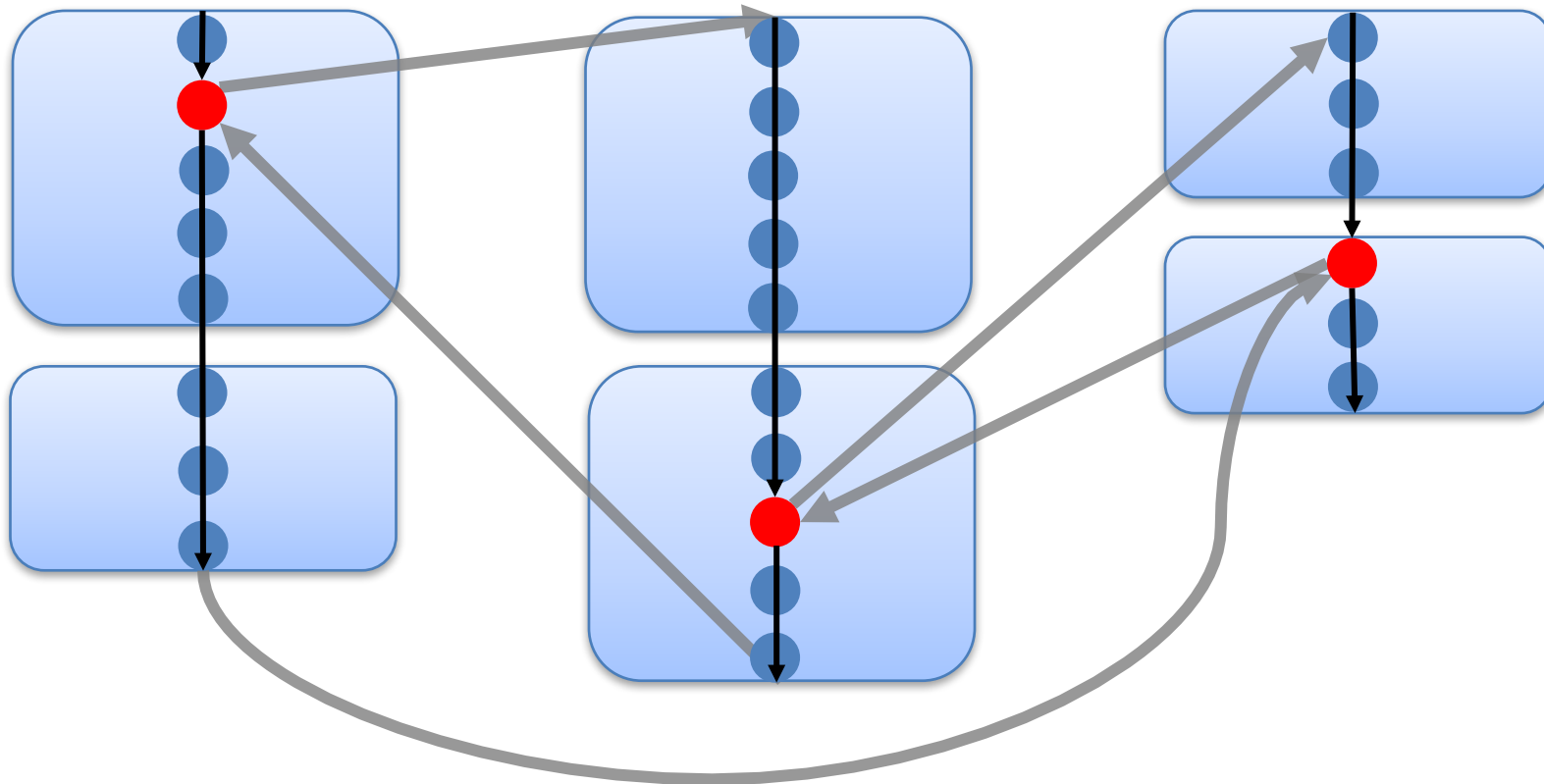


На первый,  
на третий



# One-switch: точки переключения

- Где потенциальные точки переключения?



# One-switch: точки переключения

Доступ к разделяемым переменным:

- Поля
  - `getField`, `putfield`, `getstatic`, `putstatic`
- Элементы массива
  - `Xaload`, `Xastore`, `X={a,i,d,...}`

# One-switch: приоритезация

- Возможных сценариев переключения может быть много
- Хочется ограничить их количество

# One-switch: приоритезация

- Возможных сценариев переключения может быть много
- Хочется ограничить их количество

Ограничим количество исполнений,  
выполняя более «интересные» раньше

# Переключение потоков

- В управляемых стратегиях исполнение де-факто **последовательное**
- Много ресурсов тратится на переключение потоков

# Переключение потоков

- В управляемых стратегиях исполнение де-факто **последовательное**
- Много ресурсов тратится на переключение потоков

Идея: **легковесные потоки!**

# Легковесные потоки

- Fibers, green threads, [co,go,...]routines, ...
- Фреймворк Quasar
  - <http://www.paralleluniverse.co/quasar/>
- При небольшой глубине стека переключение потоков намного дешевле
  - Наш случай!

# Стресс vs Управляемые

- Управляемые стратегии ещё в разработке
- One-switch чуть медленнее Stress режима
- One-switch + Fibers = в 1.5 раза быстрее
- One-switch + Fibers + Parallel = ???



# Стресс vs Управляемые

- Управляемые стратегии ещё в разработке
- One-switch чуть медленнее Stress режима
- One-switch + Fibers = в 1.5 раза быстрее
- One-switch + Fibers + Parallel = ???

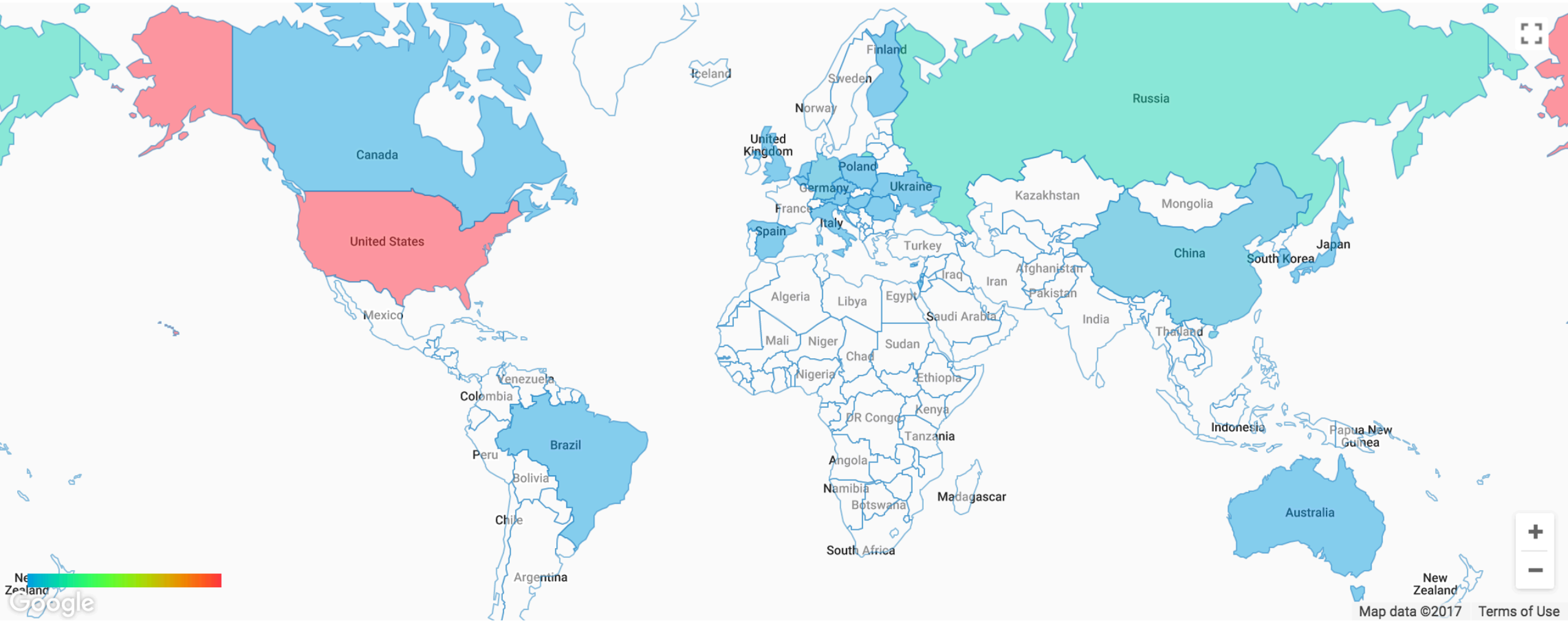
В управляемых  
стратегиях нет гонок!

# Итого

- Тестировать многопоточные программы непросто, но теперь есть Lin-Check
  - <https://github.com/Devexperts/lin-check>
- Используется в Devexperts и JetBrains
  - И еще кто-то скрывается 😊

# lin-check downloads

Total downloads: 2,048



# Спасибо за внимание!

Никита Коваль

[nkoval@devexperts.com](mailto:nkoval@devexperts.com)

[twitter.com/nkoval\\_](https://twitter.com/nkoval_)

