МИР Plat.Form

Не happens-before единым

нестандартные семантики

Обо мне

linkedin.com/in/alantsov

Мир Plat.Form

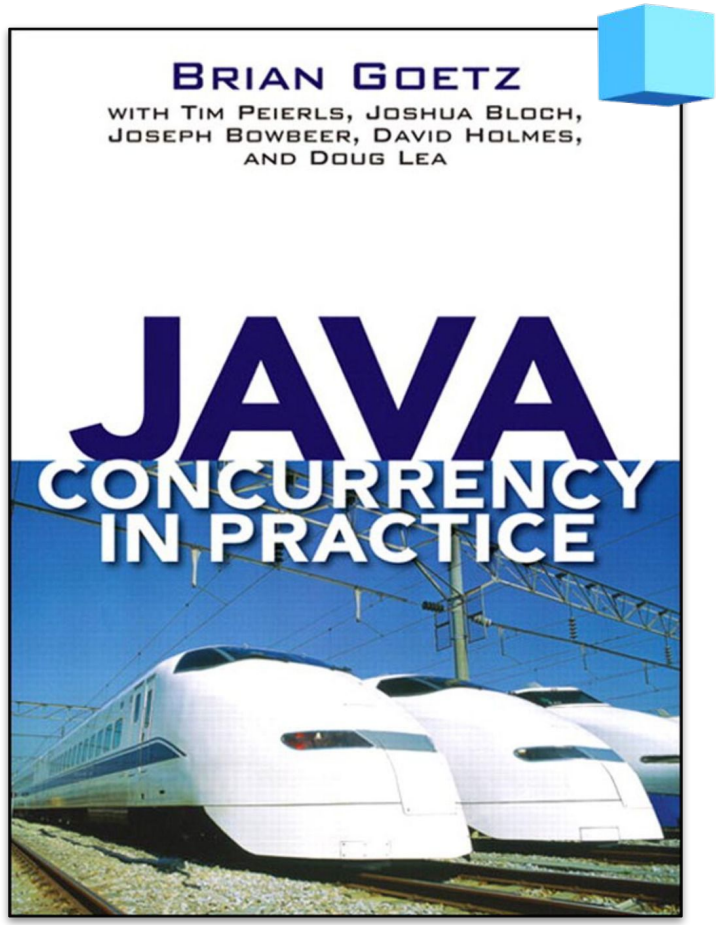
mir-platform.ru

Все примеры и бенчмарки

github.com/lantalex/jpoint-2023-semantics

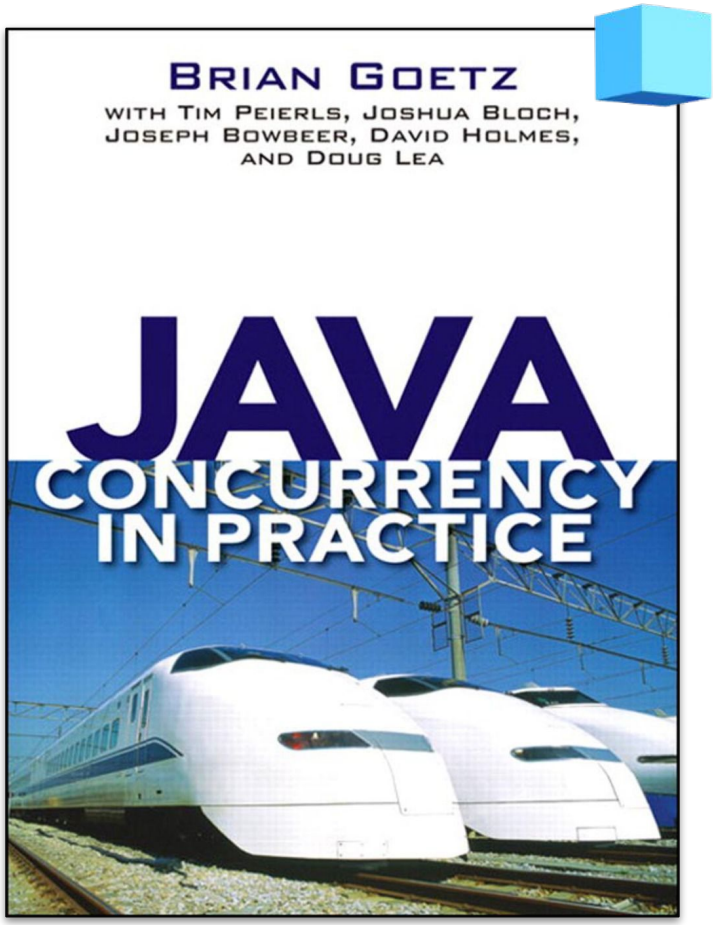
Многопоточность в 2023 году, серьезно?

Ведь столько уже рассказано:



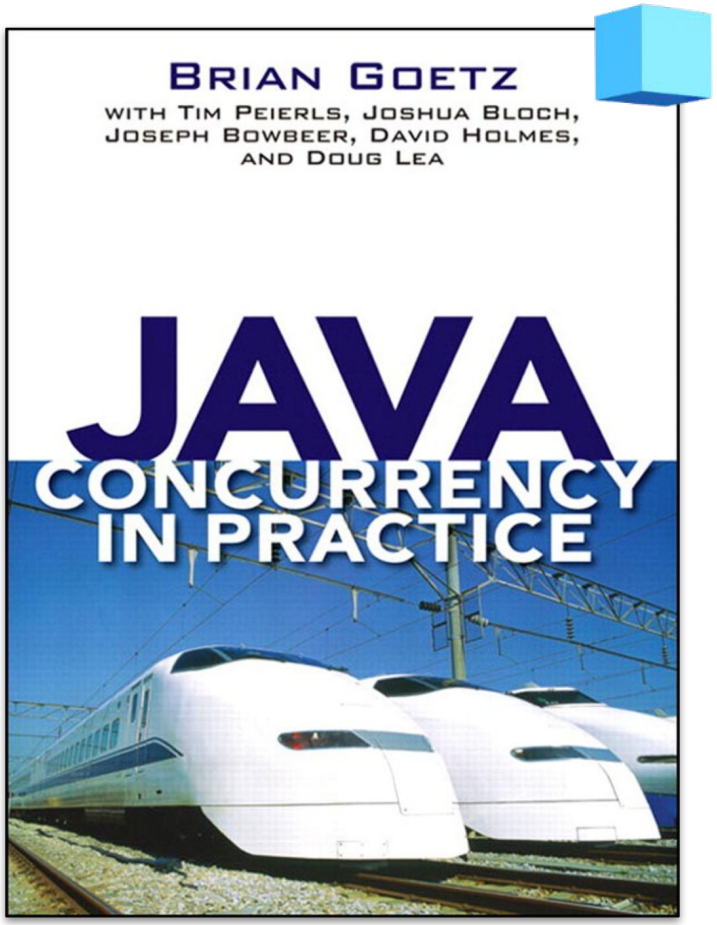
Многопоточность в 2023 году, серьезно?

Ведь столько уже рассказано:



Многопоточность в 2023 году, серьезно?

Ведь столько уже рассказано:



Кризис веры #1

Java 8: в AtomicLong есть методы, которые не описываются в JMM

```
// Eventually sets to the given value

public final void lazySet(long newValue) {
   …
}
```

Кризис веры #2

Многопоточных библиотеки: иногда работают не в терминах JMM

Disruptor: *github.com/LMAX-Exchange/disruptor*

```
public boolean isAvailable(final long sequence) {
    int index = calculateIndex(sequence);
    int flag = calculateAvailabilityFlag(sequence);
    return (int) AVAILABLE_ARRAY.getAcquire(availableBuffer, index) == flag;
}
```

🤔 Почему нельзя написать просто: `availableBuffer[index] == flag`?

Кризис веры #3

Java 9+: добивающий удар от AtomicLong

```
AtomicLong value = new AtomicLong();
value.set
```

```
set(long newValue)                    void
setRelease(long newValue)             void
setOpaque(long newValue)              void
setPlain(long newValue)               void
```

Если это “секретное” API - почему оно доступно даже на базовых классах? 😠

Кризис веры #3

Java 9+: добивающий удар от AtomicLong

```
AtomicLong value = new AtomicLong();
value.get
```

get()	long
getAcquire()	long
getOpaque()	long
getPlain()	long

Если это “секретное” API - почему оно доступно даже на базовых классах? 😠

JEP 193: Variable Handles

Доступен с Java 9; основная мотивация - миграция с *Unsafe*

Описание от Doug Lea: **Using JDK 9 Memory Order Modes**

https://gee.cs.oswego.edu/dl/html/j9mm.html

“Помимо двух существующих режимов упорядочивания доступов к памяти (aka семантик), теперь доступны ещё две промежуточные семантики - opaque и acquire/release”

Using JDK 9 Memory Order Modes

“Помимо двух существующих режимов упорядочивания доступов к памяти (aka семантик), теперь доступны ещё две промежуточные семантики - opaque и acquire/release”

Using JDK 9 Memory Order Modes

“Помимо двух существующих режимов упорядочивания доступов к памяти (aka семантик), теперь доступны ещё две промежуточные семантики - opaque и acquire/release”

1. Что такое «семантики»?

Using JDK 9 Memory Order Modes

“Помимо двух существующих режимов упорядочивания доступов к памяти (aka семантик), теперь доступны ещё две промежуточные семантики - opaque и acquire/release”

1. Что такое «семантики»?
2. Можно ли обойтись без семантик?

Using JDK 9 Memory Order Modes

“Помимо двух существующих режимов упорядочивания доступов к памяти (aka семантик), теперь доступны ещё две промежуточные семантики - opaque и acquire/release”

1. Что такое «семантики»?
2. Можно ли обойтись без семантик?
3. Существующие семантики

Using JDK 9 Memory Order Modes

“Помимо двух существующих режимов упорядочивания доступов к памяти (aka семантик), теперь доступны ещё две промежуточные семантики - opaque и acquire/release”

1. Что такое «семантики»?
2. Можно ли обойтись без семантик?
3. Существующие семантики
4. Как задавать семантики в своем коде

Using JDK 9 Memory Order Modes

“Помимо двух существующих режимов упорядочивания доступов к памяти (aka семантик), теперь доступны ещё две промежуточные семантики - opaque и acquire/release”

1. Что такое «семантики»?
2. Можно ли обойтись без семантик?
3. Существующие семантики
4. Как задавать семантики в своем коде
5. Opaque
6. Acquire/Release

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- Opaque-семантика
- Acquire/Release-семантика

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- Opaque-семантика
- Acquire/Release-семантика

Семантики: смысл термина

Семантики: смысл термина

```
int x, y, z;
volatile boolean ready = false;
```

Семантики: смысл термина

```
int x, y, z;
volatile boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready = true;
```

Семантики: смысл термина

```
int x, y, z;
volatile boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready = true;
```

Thread 2

```
while(!ready) {
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

Семантики: смысл термина

```
int x, y, z;
volatile boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready = true;
```

hb

Thread 2

```
while(!ready) {
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

Семантики: смысл термина

```
int x, y, z;
volatile boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready = true;
```

hb

hb

Thread 2

```
while(!ready) {
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

Семантики: смысл термина

```
int x, y, z;
volatile boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready = true;
```

hb

hb

Thread 2

```
while(!ready) {
    Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

hb

Семантики: смысл термина

```
int x, y, z;
volatile boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready = true;
```

hb

hb

Thread 2

```
while(!ready) {
    Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

hb

Семантики: смысл термина

Операции чтения и записи могут давать дополнительные гарантии

```
ready = true;
```

→ $\mathtt{Write}_{\mathtt{volatile}}(\mathtt{ready}) \leftarrow \mathtt{true}$

```
while(!ready) {};
```

→ $\mathtt{Read}_{\mathtt{volatile}}(\mathtt{ready}) \rightarrow \mathtt{true}$

Семантики: смысл термина

Операции чтения и записи могут давать дополнительные гарантии

```
ready = true;
```

→ $\mathtt{Write}_{\mathtt{volatile}}(\mathtt{ready}) \leftarrow \mathtt{true}$

```
while(!ready) {};
```

→ $\mathtt{Read}_{\mathtt{volatile}}(\mathtt{ready}) \rightarrow \mathtt{true}$

В многопоточности чтения/записи всегда имеют какую-то семантику

$\mathtt{Write}(\mathtt{ready}) \leftarrow \mathtt{true}$ 🤔

Семантики: смысл термина

Операции чтения и записи могут давать дополнительные гарантии

```
ready = true;
```

→ $\text{Write}_{\text{volatile}}(\text{ready}) \leftarrow \text{true}$

```
while(!ready) {};
```

→ $\text{Read}_{\text{volatile}}(\text{ready}) \rightarrow \text{true}$

В многопоточности чтения/записи всегда имеют какую-то семантику

$\text{Write}(\text{ready}) \leftarrow \text{true}$ 🤔

$\text{Write}_{\text{volatile}}(\text{ready}) \leftarrow \text{true}$ 😎

Семантики: смысл термина

Операции чтения и записи могут давать дополнительные гарантии

```
ready = true;
```

→ $\mathrm{Write}_{\mathrm{volatile}}(\mathrm{ready}) \leftarrow \mathrm{true}$

```
while(!ready) {};
```

→ $\mathrm{Read}_{\mathrm{volatile}}(\mathrm{ready}) \rightarrow \mathrm{true}$

В многопоточности чтения/записи всегда имеют какую-то семантику

$\mathrm{Write}(\mathrm{ready}) \leftarrow \mathrm{true}$ 🤔

$\mathrm{Write}_{\mathrm{volatile}}(\mathrm{ready}) \leftarrow \mathrm{true}$ 😎

семантика

Семантики: смысл термина

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

Опираясь на эти дополнительные гарантии, становится возможным писать корректный многопоточный код

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- Opaque-семантика
- Acquire/Release-семантика

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- Opaque-семантика
- Acquire/Release-семантика

Golang

Официальная документация, глава про модель памяти

If you must read the rest of this document to understand the behavior of your program, you are being too clever.

Don't be clever.

Golang

Официальная документация, глава про модель памяти

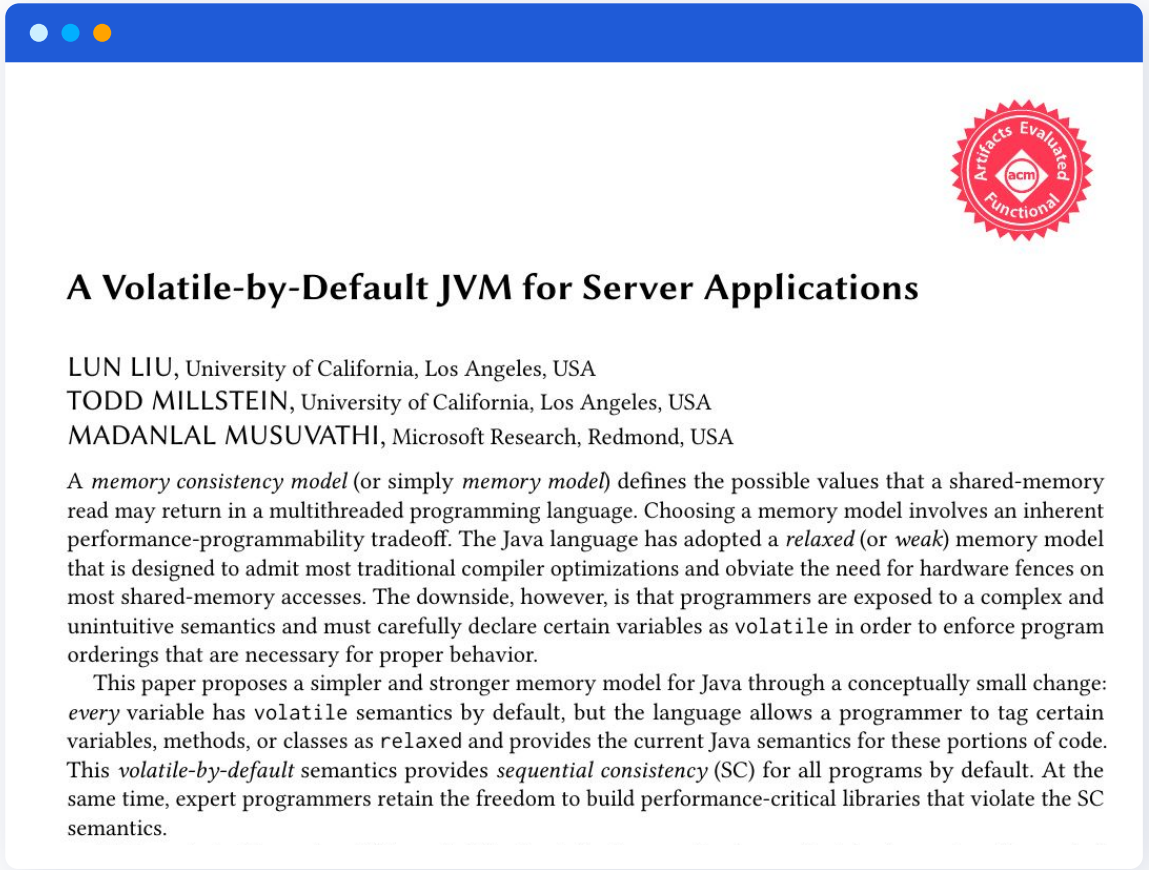
If you must read the rest of this document to understand the behavior of your program, you are being too clever.

Don't be clever.

у нас такую тему не уважают

A volatile-by-default JVM for server applications

by Lun Liu, Todd Millstein, Madanlal Musuvathi: *https://dl.acm.org/doi/10.1145/3133873*



A Volatile-by-Default JVM for Server Applications

LUN LIU, University of California, Los Angeles, USA
TODD MILLSTEIN, University of California, Los Angeles, USA
MADANLAL MUSUVATHI, Microsoft Research, Redmond, USA

A *memory consistency model* (or simply *memory model*) defines the possible values that a shared-memory read may return in a multithreaded programming language. Choosing a memory model involves an inherent performance-programmability tradeoff. The Java language has adopted a *relaxed* (or *weak*) memory model that is designed to admit most traditional compiler optimizations and obviate the need for hardware fences on most shared-memory accesses. The downside, however, is that programmers are exposed to a complex and unintuitive semantics and must carefully declare certain variables as `volatile` in order to enforce program orderings that are necessary for proper behavior.

This paper proposes a simpler and stronger memory model for Java through a conceptually small change: *every* variable has `volatile` semantics by default, but the language allows a programmer to tag certain variables, methods, or classes as `relaxed` and provides the current Java semantics for these portions of code. This *volatile-by-default* semantics provides *sequential consistency* (SC) for all programs by default. At the same time, expert programmers retain the freedom to build performance-critical libraries that violate the SC semantics.

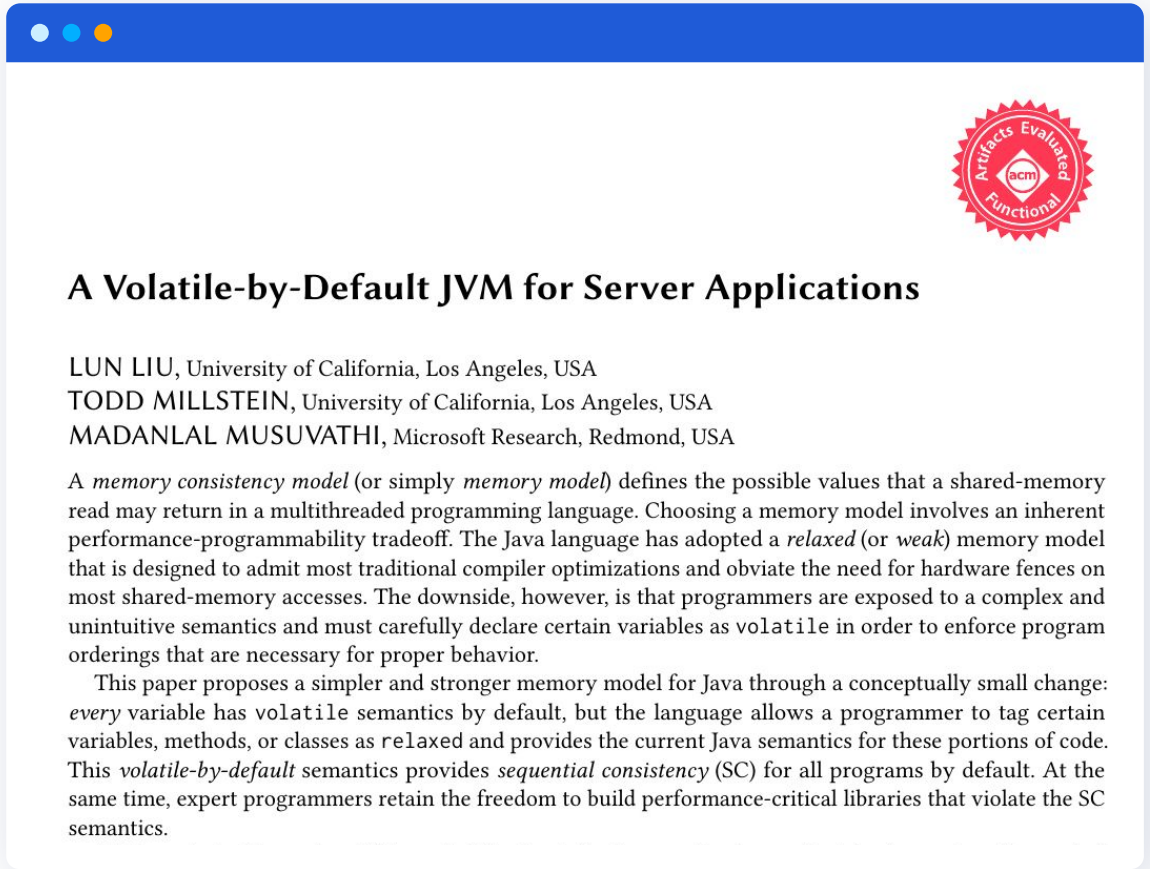
A volatile-by-default JVM for server applications

by Lun Liu, Todd Millstein, Madanlal Musuvathi: *https://dl.acm.org/doi/10.1145/3133873*

Любая переменная работает так, как будто бы объявлена с спецификатором `volatile`

Только одна, самая сильная семантика, максимальное количество гарантий

Но что с производительностью?



A Volatile-by-Default JVM for Server Applications

LUN LIU, University of California, Los Angeles, USA
TODD MILLSTEIN, University of California, Los Angeles, USA
MADANLAL MUSUVATHI, Microsoft Research, Redmond, USA

A *memory consistency model* (or simply *memory model*) defines the possible values that a shared-memory read may return in a multithreaded programming language. Choosing a memory model involves an inherent performance-programmability tradeoff. The Java language has adopted a *relaxed* (or *weak*) memory model that is designed to admit most traditional compiler optimizations and obviate the need for hardware fences on most shared-memory accesses. The downside, however, is that programmers are exposed to a complex and unintuitive semantics and must carefully declare certain variables as `volatile` in order to enforce program orderings that are necessary for proper behavior.

This paper proposes a simpler and stronger memory model for Java through a conceptually small change: *every* variable has `volatile` semantics by default, but the language allows a programmer to tag certain variables, methods, or classes as `relaxed` and provides the current Java semantics for these portions of code. This *volatile-by-default* semantics provides *sequential consistency* (SC) for all programs by default. At the same time, expert programmers retain the freedom to build performance-critical libraries that violate the SC semantics.

A volatile-by-default JVM for server applications

два бенчмарка: `DaCapo` и `spark-perf`, измеряем насколько замедлились

A volatile-by-default JVM for server applications

два бенчмарка: `DaCapo` и `spark-perf`, измеряем насколько замедлились

x86 : в среднем в **1,5-2** раза ; в худшем в **2-3** раза

A volatile-by-default JVM for server applications

два бенчмарка: `DaCapo` и `spark-perf`, измеряем насколько замедлились

x86 : в среднем в **1,5-2** раза ; в худшем в **2-3** раза

ARM : в среднем в **1,5-2** раза ; в худшем в **3-5+** раз

A volatile-by-default JVM for server applications

два бенчмарка: `DaCapo` и `spark-perf`, измеряем насколько замедлились

`x86`: в среднем в **1,5-2** раза ; в худшем в **2-3** раза

`ARM`: в среднем в **1,5-2** раза ; в худшем в **3-5+** раз

Если производительность не критична, то щедрая обсыпка сомнительного кода `volatile` - вполне нормальное решение

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- Opaque-семантика
- Acquire/Release-семантика

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- Opaque-семантика
- Acquire/Release-семантика

Существующие семантики

Были доступны до JEP 193:

```
plain | volatile
```

Существующие семантики

Были доступны до JEP 193:

`plain` | `volatile`

После JEP 193:

`plain` | | `volatile`

Существующие семантики

Были доступны до JEP 193:

```
plain | volatile
```

После JEP 193:

```
plain | opaque | acquire-release | volatile
```

Существующие семантики

Были доступны до JEP 193:

`plain` | `volatile`

После JEP 193:

`plain` | `opaque` | `acquire-release` | `volatile`

Нет новых семантик (opaque, acquire-release) - поведение программы полностью описывается текущей JMM

Карта семантик

`plain` `opaque` `acquire-release` `volatile`

Карта семантик

`plain` ⊆ `opaque` ⊆ `acquire-release` ⊆ `volatile`

Карта семантик

`plain` ⊆ `opaque` ⊆ `acquire-release` ⊆ `volatile`

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

Карта семантик

`plain` ⊆ `opaque` ⊆ `acquire-release` ⊆ `volatile`

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

Любая семантика включает в себя все гарантии более слабых семантик

`plain` - самая слабая семантика (почти никаких доп. гарантий, зато дешевая)

`volatile` - самая сильная семантика (максимальные гарантии ценой ухудшения производительности)

Plain – семантика по умолчанию

Нет многопоточности - все работает как ожидается

```
boolean done = false;
```

$\mathtt{Write}_{\mathtt{plain}}(\mathtt{done}) \leftarrow \mathtt{false}$

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1

```
while(!done) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1	Thread 2

`while(!done) {`
`  doSomeWork();`
`}`
	`Thread.sleep(100);`
	`done = true;`

$\mathbf{Write}_{\mathbf{plain}}(\mathbf{done}) \leftarrow \mathbf{true}$

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1

```
while(!done) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1	Thread 2

`while(!done) {`
`  doSomeWork();`
`}`
	`Thread.sleep(100);`
	`done = true;`

$\mathbf{Read}_{\mathbf{plain}}(\mathbf{done}) \leftarrow \mathbf{false}$

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1	Thread 2

`while(!done) {`
`  doSomeWork();`
`}`
	`Thread.sleep(100);`
	`done = true;`

$Read_{plain}(done) \leftarrow false$; $Read_{plain}(done) \leftarrow false$

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1	Thread 2

`while(!done) {`
`  doSomeWork();`
`}`
	`Thread.sleep(100);`
	`done = true;`

$\mathbf{Read}_{\mathbf{plain}}$(done)←**false** ; $\mathbf{Read}_{\mathbf{plain}}$(done)←**false** ; … ;

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1	Thread 2

`while(!done) {`
`  doSomeWork();`
`}`
	`Thread.sleep(100);`
	`done = true;`

$\text{Read}_{\text{plain}}(\text{done}) \leftarrow \text{false}$; $\text{Read}_{\text{plain}}(\text{done}) \leftarrow \text{false}$; … ; $\text{Read}_{\text{plain}}(\text{done}) \leftarrow \text{true}$

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1

```
while(!done) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1

```
while(!done) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

Да почему зависает то?!

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1

```
while(!done) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

Да почему зависает то?!

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1

```
boolean localDone = done;
while(!localDone) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

Plain – семантика по умолчанию

Есть многопоточность - нужно корректно синхронизировать

```
boolean done = false;
```

Thread 1

```
boolean localDone = done;
while(!localDone) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

Очень трудно сопоставить исходный код и результат после оптимизаций!

Plain – семантика по умолчанию

Без дополнительных синхронизаций в случае конкурентных записей - **data race**

Только две гарантии:

- type safety: **data-race** не приводит к *Undefined Behavior*

- out-of-thin-air safety*: не может быть создано новых значений без каких-либо операций записи или чтения даже при **data-race** (см. *youtu.be/C6b_dFtujKo*)

*возможно это не всегда верно: www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html

Карта семантик

`plain` ⊆ `opaque` ⊆ `acq-rel` ⊆ `volatile`

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

Карта семантик

plain ⊆ **opaque** ⊆ **acq-rel** ⊆ **volatile**

- type-safety
- OoTA-safety*

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- Opaque-семантика
- Acquire/Release-семантика

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- **Как задавать семантики в своем коде**
- **Opaque-семантика**
- **Acquire/Release-семантика**

Несколько слов об API

```
int x;
x = 42;
int copyX = x;
```

Несколько слов об API

```
int x;
x = 42;
int copyX = x;
```

plain-семантика

$\mathtt{Write}_{\mathtt{plain}}(x) \leftarrow 42$

$\mathtt{Read}_{\mathtt{plain}}(x) \rightarrow 42$

$\mathtt{Write}_{\mathtt{plain}}(\mathtt{copyX}) \leftarrow 42$

Несколько слов об API

```
volatile int x;
x = 42;
int copyX = x;
```

Несколько слов об API

```
volatile int x;
x = 42;
int copyX = x;
```

volatile-семантика

$\mathtt{Write}_{\mathtt{volatile}}(\mathtt{x}) \leftarrow 42$

$\mathtt{Read}_{\mathtt{volatile}}(\mathtt{x}) \rightarrow 42$

plain-семантика

$\mathtt{Write}_{\mathtt{plain}}(\mathtt{copyX}) \leftarrow 42$

VarHandle API

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;
```

VarHandle API

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {
  try {

  } catch (ReflectiveOperationException e) {
    throw new Error(e);
  }
}
```

VarHandle API

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {
  try {
    X_HANDLE =
  } catch (ReflectiveOperationException e) {
    throw new Error(e);
  }
}
```

VarHandle API

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {
  try {
    X_HANDLE = MethodHandles.lookup().findVarHandle(
  } catch (ReflectiveOperationException e) {
    throw new Error(e);
  }
}
```

VarHandle API

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {
  try {
    X_HANDLE = MethodHandles.lookup().findVarHandle(MyClass.class
  } catch (ReflectiveOperationException e) {
    throw new Error(e);
  }
}
```

VarHandle API

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {
  try {
    X_HANDLE = MethodHandles.lookup().findVarHandle(MyClass.class, "x", int.class);
  } catch (ReflectiveOperationException e) {
    throw new Error(e);
  }
}
```

VarHandle API

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {
  try {
    X_HANDLE = MethodHandles.lookup().findVarHandle(MyClass.class, "x", int.class);
  } catch (ReflectiveOperationException e) {
    throw new Error(e);
  }
}
```

VarHandle API

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API: использование

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.setOpaque(m, 42);
```

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.setOpaque(m, 42);
```

$\mathbf{Write}_{\mathbf{opaque}}(\mathbf{m.x}) \leftarrow \mathbf{42}$

opaque-семантика

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.setOpaque(m, 42);

int copyX = (int) X_HANDLE.getOpaque(m);
```

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.setOpaque(m, 42);

int copyX = (int) X_HANDLE.getOpaque(m);
```

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

$\mathbf{Read}_{\mathbf{opaque}}(\mathbf{m.x}) \rightarrow \mathbf{42}$

opaque-семантика

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.setOpaque(m, 42);

int copyX = (int) X_HANDLE.getOpaque(m);
```

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

$\mathbf{Write}_{\mathbf{plain}}(\mathbf{copyX}) \leftarrow \mathbf{42}$

plain-семантика

$\mathbf{Read}_{\mathbf{opaque}}(\mathbf{m.x}) \rightarrow \mathbf{42}$

opaque-семантика

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.setOpaque(m, 42);

int copyX = (int) X_HANDLE.getOpaque(m);
```

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.setVolatile(m, 42);

int copyX = (int) X_HANDLE.getVolatile(m);
```

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.set(m, 42);

int copyX = (int) X_HANDLE.get(m);
```

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.setRelease(m, 42);

int copyX = (int) X_HANDLE.getAcquire(m);
```

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.setOpaque(m, 42);

int copyX = (int) X_HANDLE.getOpaque(m);
```

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API: использование

```
MyClass m = new MyClass(...);

X_HANDLE.setOpaque(m, 42);

int copyX = (int) X_HANDLE.getOpaque(m);
```

MyClass.java

```
int x;

public static final VarHandle X_HANDLE;

static {...}
```

VarHandle API - позволяет выбирать семантику при операциях записи/чтения

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- **Как задавать семантики в своем коде**
- **Opaque-семантика**
- **Acquire/Release-семантика**

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- **Opaque-семантика**
- **Acquire/Release-семантика**

Opaque

```
boolean done = false;
```

Thread 1

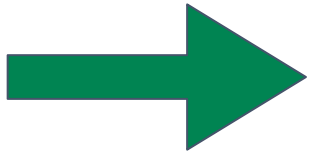
```
while(!done) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

Opaque

```
while(!done) {
   doSomeWork();
}
```



```
boolean localDone = done;
while(!localDone) {
   doSomeWork();
}
```

Opaque

```
while(!done) {
   doSomeWork();
}
```

```
boolean localDone = done;
while(!localDone) {
   doSomeWork();
}
```

Opaque

Запрещает компилятору некоторые оптимизации чтения/записи переменной

Операция есть в исходном коде ⇒ операция есть в оптимизированном коде

```
while(!done) {
   doSomeWork();
}
```

```
boolean localDone = done;
while(!localDone) {
   doSomeWork();
}
```

Opaque

```
boolean done = false;
VarHandle DONE = ...;
```

Thread 2
`Thread.sleep(100);` `done = true;`

Opaque

```
boolean done = false;
VarHandle DONE = ...;
```

Thread 1

```
while(!DONE.getOpaque()) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

Opaque

```
boolean done = false;
VarHandle DONE = ...;
```

Thread 1

```
while(!DONE.getOpaque()) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

$\text{Read}_{\text{opaque}}(\text{done}) \leftarrow \text{false}$

Opaque

```
boolean done = false;
VarHandle DONE = ...;
```

Thread 1

```
while(!DONE.getOpaque()) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

$\text{Read}_{\text{opaque}}(\text{done}) \leftarrow \text{false}$; $\text{Read}_{\text{opaque}}(\text{done}) \leftarrow \text{false}$

Opaque

```
boolean done = false;
VarHandle DONE = ...;
```

Thread 1

```
while(!DONE.getOpaque()) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

$\text{Read}_{\text{opaque}}(\text{done}) \leftarrow \text{false}$; $\text{Read}_{\text{opaque}}(\text{done}) \leftarrow \text{false}$; … ; $\text{Read}_{\text{opaque}}(\text{done}) \leftarrow \text{true}$

Opaque

```
boolean done = false;
VarHandle DONE = ...;
```

Thread 1

```
while(!DONE.getOpaque()) {
   doSomeWork();
}
```

Thread 2

```
Thread.sleep(100);
done = true;
```

🙂 Теперь работает как нужно

Opaque

```
boolean done = false;
VarHandle DONE = ...;
```

Thread 1	Thread 2

`while(!DONE.getOpaque()) {`
`  doSomeWork();`
`}`
	`Thread.sleep(100);`
	`done = true;`

🙂 Теперь работает как нужно

Progress: записи в конечном итоге видны *последующим* чтениям

Карта семантик

plain ⊆ **opaque** ⊆ **acq-rel** ⊆ **volatile**

- type-safety
- OoTA-safety*

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

Карта семантик

plain	⊆	opaque	⊆	acq-rel	⊆	volatile
▪ type-safety ▪ OoTA-safety*		▪ progress				

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

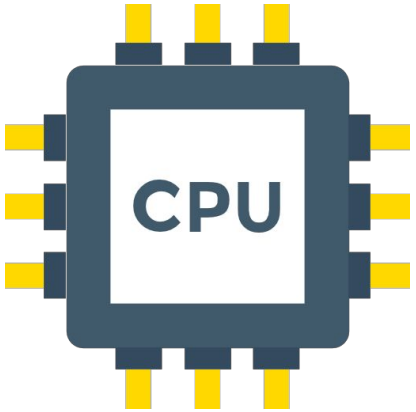
Opaque

Не только компилятор может переупорядочить операции чтения/записи

```
int a = 0;
int b = 0;
```

Thread 1

`a.setOpaque(1);`
 	
`b.setOpaque(1);`



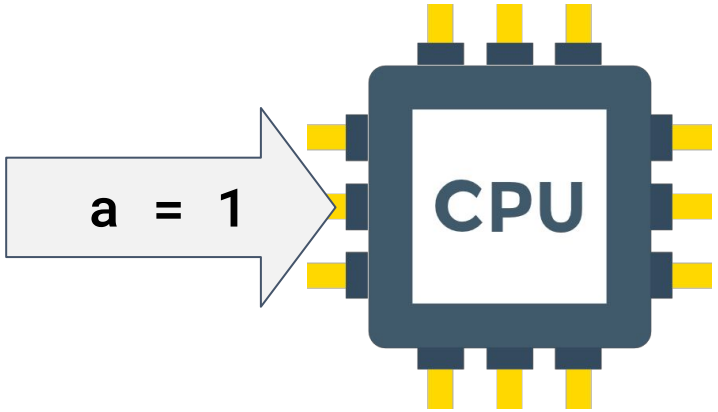
Opaque

Не только компилятор может переупорядочить операции чтения/записи

```
int a = 0;
int b = 0;
```

Thread 1

`a.setOpaque(1);`
 	
`b.setOpaque(1);`



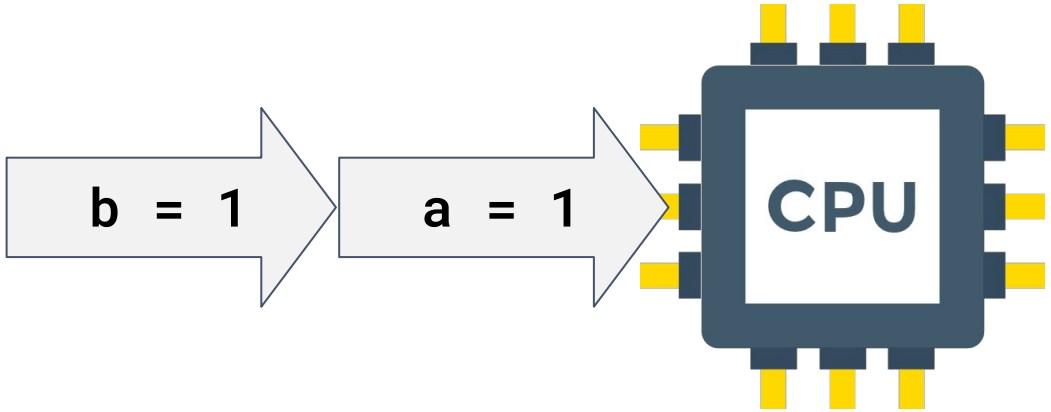
Opaque

Не только компилятор может переупорядочить операции чтения/записи

```
int a = 0;
int b = 0;
```

Thread 1

`a.setOpaque(1);`
 	
`b.setOpaque(1);`



Opaque

Не только компилятор может переупорядочить операции чтения/записи

```
int a = 0;
int b = 0;
```

Thread 1	Thread 2

`a.setOpaque(1);`
 	
`b.setOpaque(1);`	

Opaque

Не только компилятор может переупорядочить операции чтения/записи

```
int a = 0;
int b = 0;
```

Thread 1

`a.setOpaque(1);`
 	
`b.setOpaque(1);`

Thread 2

`if (b.getOpaque() == 1 &&`
`  a.getOpaque() == 0)  {`
 	
`}`

Opaque

Не только компилятор может переупорядочить операции чтения/записи

```
int a = 0;
int b = 0;
```

Thread 1

`a.setOpaque(1);`
 	
`b.setOpaque(1);`

Thread 2

`if (b.getOpaque() == 1 &&`
`  a.getOpaque() == 0)  {`
 	
`}`

$\text{Read}_{\text{opaque}}(b) \leftarrow 1$; $\text{Read}_{\text{opaque}}(a) \leftarrow 0$

Opaque

Не только компилятор может переупорядочить операции чтения/записи

```
int a = 0;
int b = 0;
```

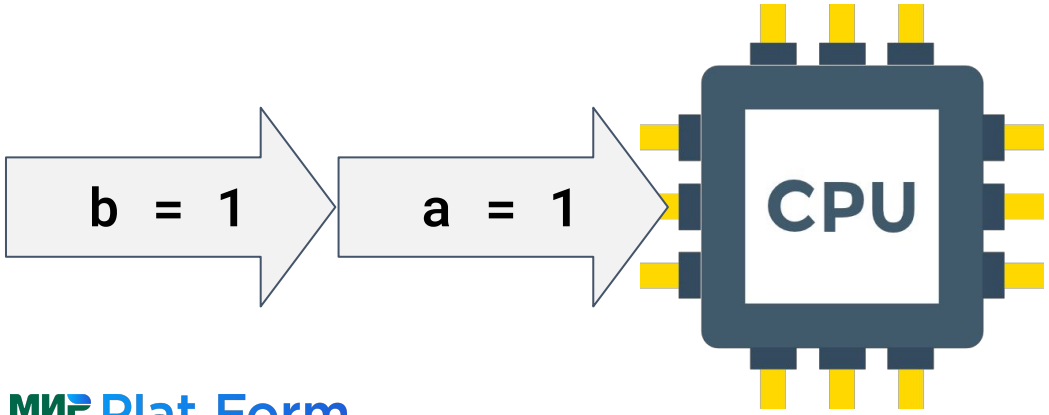
Thread 1

`a.setOpaque(1);`
 	
`b.setOpaque(1);`

Thread 2

`if (b.getOpaque() == 1 &&`
`  a.getOpaque() == 0) {`
 	
`}`

$\text{Read}_{\text{opaque}}(b) \leftarrow 1 \quad ; \quad \text{Read}_{\text{opaque}}(a) \leftarrow 0$



Opaque

Не только компилятор может переупорядочить операции чтения/записи

```
int a = 0;
int b = 0;
```

Thread 1

`a.setOpaque(1);`
 	
`b.setOpaque(1);`

Thread 2

`if (b.getOpaque() == 1 &&`
`  a.getOpaque() == 0)  {`
 	
`  throw new Error("Bad CPU")`
 	
`}`

Opaque

Может ли произойти исключение?

```
a.setOpaque(1);
b.setOpaque(1);
```

```
if (b.getOpaque() == 1 &&
    a.getOpaque() == 0)   {
    throw new Error("Bad CPU")
}
```

Opaque

Может ли произойти исключение?

```
a.setOpaque(1);
b.setOpaque(1);
```

```
if (b.getOpaque() == 1 &&
    a.getOpaque() == 0)   {
    throw new Error("Bad CPU")
}
```

x86: нет, процессор не переупорядочивает операции записи

Opaque

Может ли произойти исключение?

```
a.setOpaque(1);
b.setOpaque(1);
```

```
if (b.getOpaque() == 1 &&
    a.getOpaque() == 0)   {
    throw new Error("Bad CPU")
}
```

x86: нет, процессор не переупорядочивает операции записи

ARM: да, процессор может переупорядочить *независимые* операции

Opaque

Не пригодна для упорядочивания доступов к разным переменным

Opaque

Не пригодна для упорядочивания доступов к разным переменным

Coherence: все операции чтения/записи по одной конкретной переменной выполняются в порядке, консистентном исходному коду

Применение *opaque*-семантики:
флаги состояний, метрики из одной переменной, индикаторы прогресса

Карта семантик

plain	⊆	opaque	⊆	acq-rel	⊆	volatile
▪ type-safety ▪ OoTA-safety*		▪ progress				

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

Карта семантик

plain	⊆	opaque	⊆	acq-rel	⊆	volatile
▪ type-safety ▪ OoTA-safety*		▪ progress ▪ coherence				

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

Opaque

Bitwise Atomicity: все чтения/записи атомарны

```
long a = 0;
```

Opaque

Bitwise Atomicity: все чтения/записи атомарны

```
long a = 0;
```

Thread 1	Thread 2
`a.setOpaque(0x00000000ffffffff)`	`a.setOpaque(0xffffffff00000000)`

Opaque

Bitwise Atomicity: все чтения/записи атомарны

```
long a = 0;
```

Thread 1
`a.setOpaque(0x00000000ffffffff)`

Thread 2
`a.setOpaque(0xffffffff00000000)`

Thread 3
`long copyA = a.getOpaque();`

Opaque

Bitwise Atomicity: все чтения/записи атомарны

```
long a = 0;
```

Thread 1
`a.setOpaque(0x00000000ffffffff)`

Thread 2
`a.setOpaque(0xffffffff00000000)`

Thread 3

`long copyA = a.getOpaque();`
 	
`assert`
`copyA == 0x0000000000000000 ||`
`copyA == 0x00000000ffffffff ||`
`copyA == 0xffffffff00000000`

Opaque

Bitwise Atomicity: все чтения/записи атомарны

```
long a = 0;
```

Thread 1
`a.setOpaque(0x00000000ffffffff)`

Thread 2
`a.setOpaque(0xffffffff00000000)`

Thread 3

`long copyA = a.getOpaque();`
 	
`assert`
`copyA == 0x0000000000000000 ||`
`copyA == 0x00000000ffffffff ||`
`copyA == 0xffffffff00000000`

Запрещенный вариант:
`copyA == 0xffffffffffffffff`

Карта семантик

plain	⊆	opaque	⊆	acq-rel	⊆	volatile
▪ type-safety ▪ OoTA-safety*		▪ progress ▪ coherence				

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

Карта семантик

plain	⊆	opaque	⊆	acq-rel	⊆	volatile
▪ type-safety ▪ OoTA-safety*		▪ progress ▪ coherence ▪ bitwise atomicity				

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- Opaque
- Acquire/Release

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- Opaque
- Acquire/Release

Acquire/Release

```
int x, y, z;
volatile boolean ready = false;
```

Acquire/Release

```
int x, y, z;
volatile boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready = true;
```

Thread 2

```
while(!ready) {
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

Acquire/Release

```
int x, y, z;
volatile boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready = true;
```

hb

hb

Thread 2

```
while(!ready) {
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

hb

Acquire/Release

```
int x, y, z;
volatile boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready = true;
```

Thread 2

```
while(!ready) {
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

Acquire/Release

```
int x, y, z;
boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
```

Thread 2

```
while(                    ){
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

Acquire/Release

```
int x, y, z;
boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready.setRelease(true)
```

Thread 2

```
while(!ready.getAcquire()){
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

Acquire/Release

```
int x, y, z;
boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready.setRelease(true)
```

Thread 2

```
while(!ready.getAcquire()){
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

Упадут ли проверки?

Acquire/Release

```
int x, y, z;
boolean ready = false;
```

Thread 1

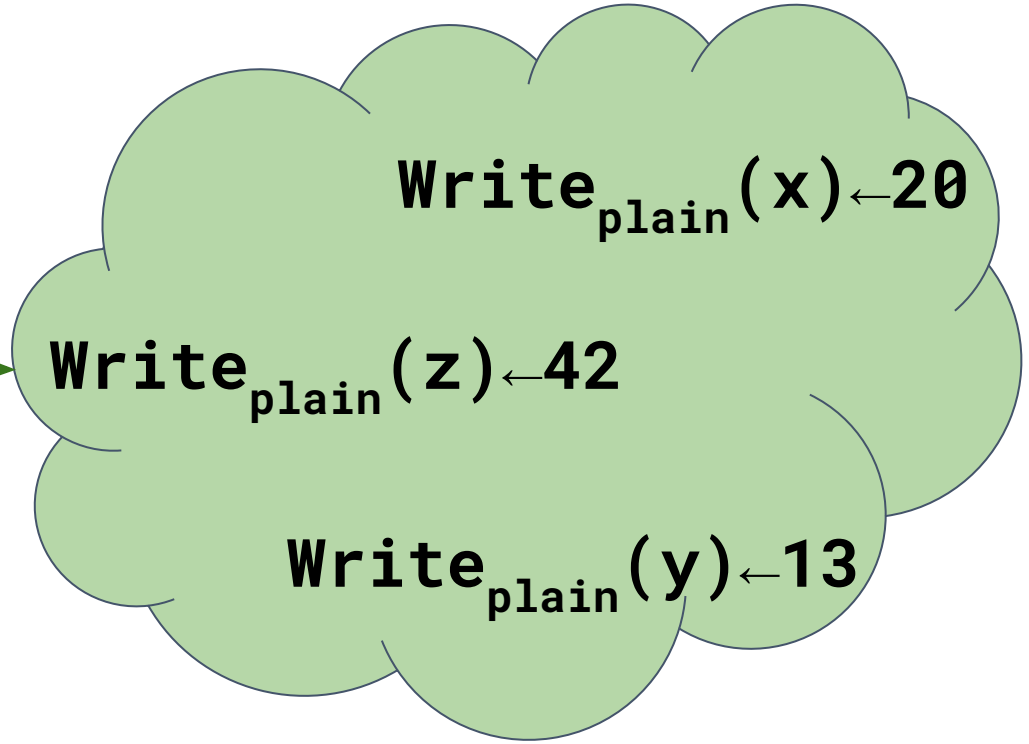
`x = 20;`
`y = 13;`
`z = 42;`
`ready.setRelease(true)`

Acquire/Release

```
int x, y, z;
boolean ready = false;
```

Thread 1

```
x = 20;
y = 13;
z = 42;
ready.setRelease(true)
```



Acquire/Release

```
int x, y, z;
boolean ready = false;
```

Thread 1


```
ready.setRelease(true)
```

Acquire/Release

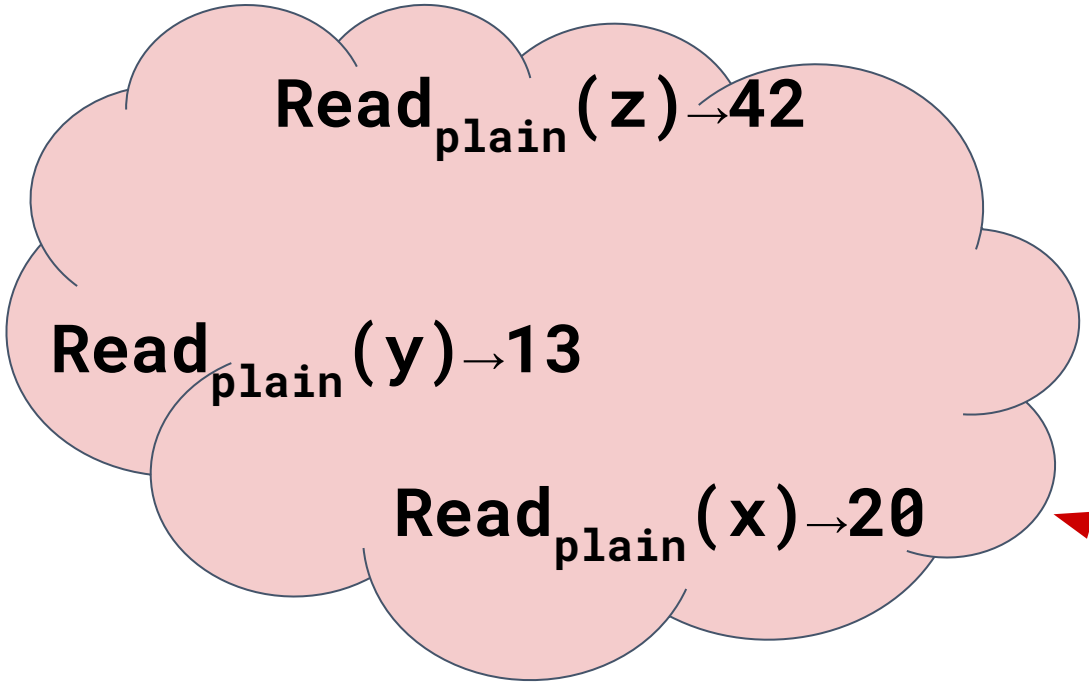
```
int x, y, z;
boolean ready = false;
```

Thread 2

```
while(!ready.getAcquire()){
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

Acquire/Release

```
int x, y, z;
boolean ready = false;
```



Thread 2

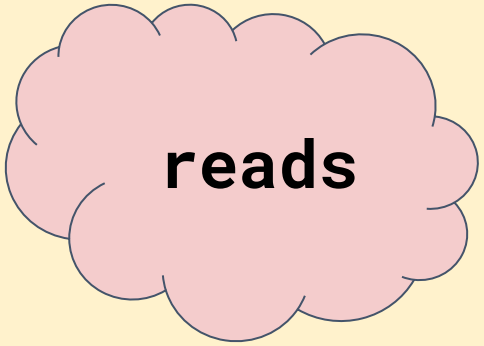
```
while(!ready.getAcquire()){
   Thread.onSpinWait();
}

assert x == 20;
assert y == 13;
assert z == 42;
```

Acquire/Release

```
int x, y, z;
boolean ready = false;
```

Thread 2

```
while(!ready.getAcquire()){
   Thread.onSpinWait();
}
```



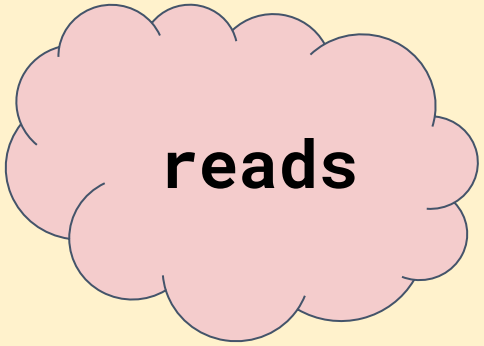
Acquire/Release

Thread 1


```
ready.setRelease(true)
```

Thread 2

```
while(!ready.getAcquire()){
   Thread.onSpinWait();
}
```



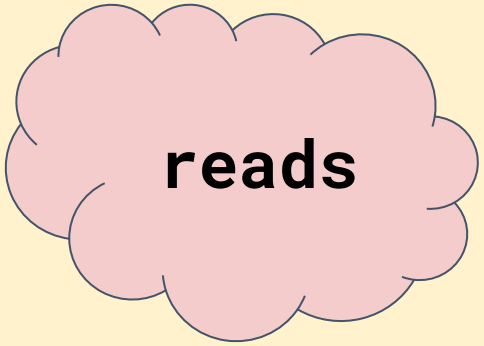
Acquire/Release

Thread 1

$\mathtt{Write}_{\mathtt{acq/rel}}(\mathtt{ready}) \leftarrow \mathtt{true}$

Thread 2

```
while(!ready.getAcquire()){
   Thread.onSpinWait();
}
```

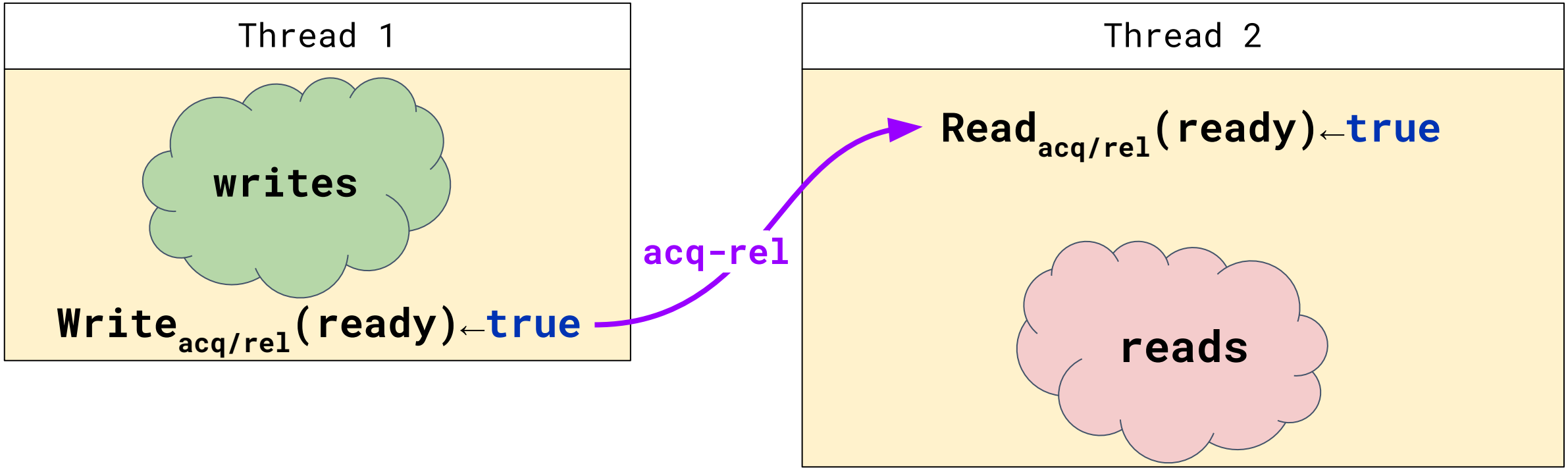


Acquire/Release

Thread 1
writes $\mathtt{Write}_{\mathtt{acq/rel}}(\mathtt{ready}) \leftarrow \mathtt{true}$

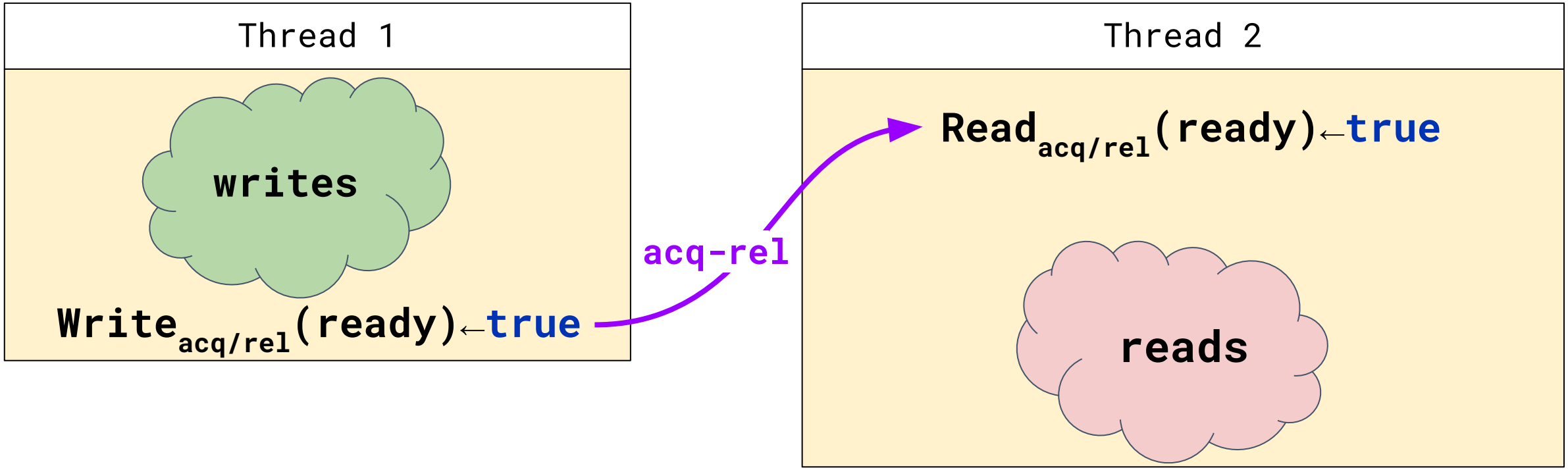
Thread 2
$\mathtt{Read}_{\mathtt{acq/rel}}(\mathtt{ready}) \leftarrow \mathtt{true}$ reads

Acquire/Release



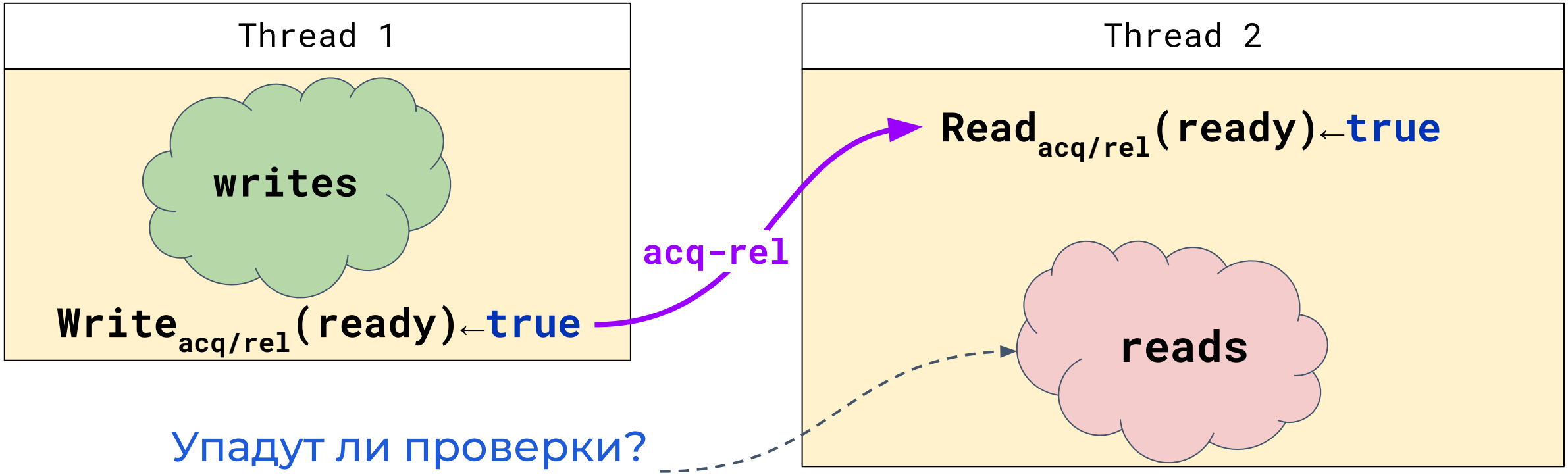
Acquire/Release

Causality: все операции, предшествующие release-записи, видны для всех операций, идущих после парного acquire-чтения



Acquire/Release

Causality: все операции, предшествующие release-записи, видны для всех операций, идущих после парного acquire-чтения



Упадут ли проверки?
Нет

Карта семантик

plain	⊆	opaque	⊆	acq-rel	⊆	volatile
▪ type-safety ▪ OoTA-safety*		▪ progress ▪ coherence ▪ bitwise atomicity				

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

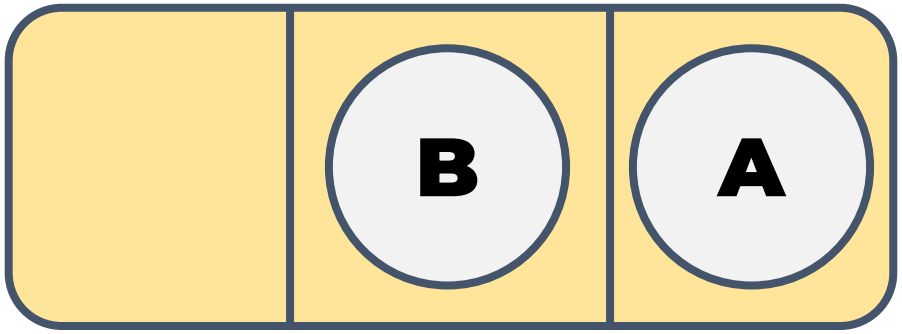
Карта семантик

plain	⊆	opaque	⊆	acq-rel	⊆	volatile
▪ type-safety ▪ OoTA-safety*		▪ progress ▪ coherence ▪ bitwise atomicity		▪ causality		

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

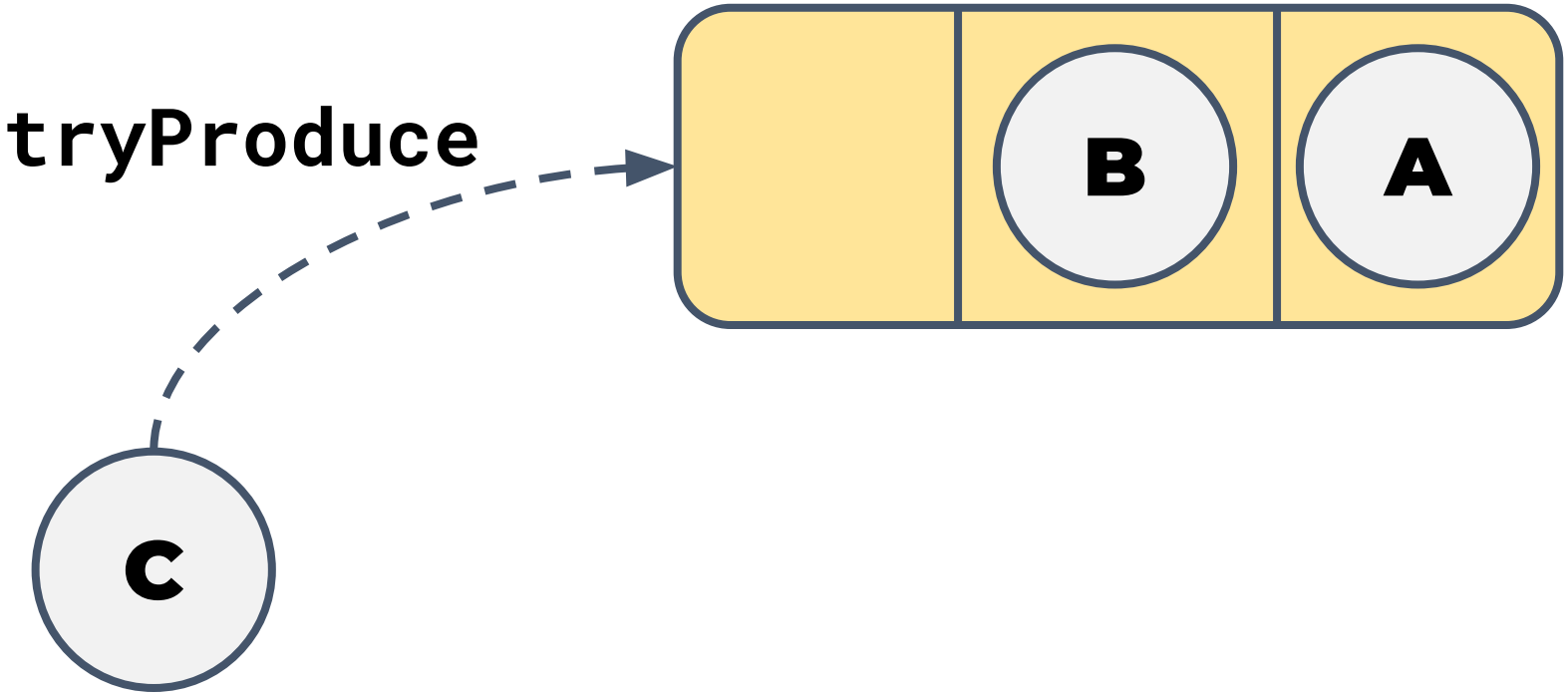
Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue



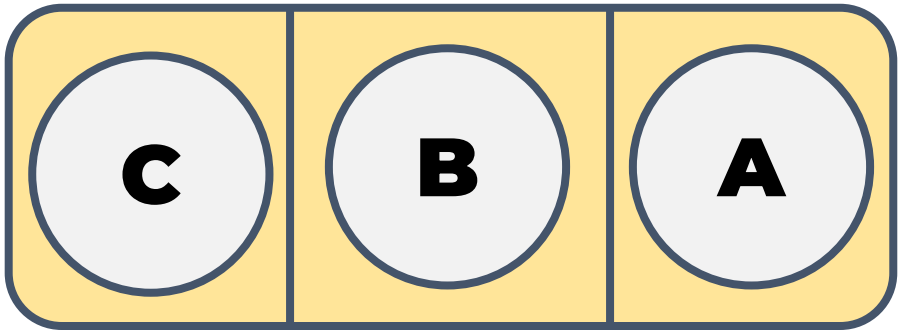
Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue



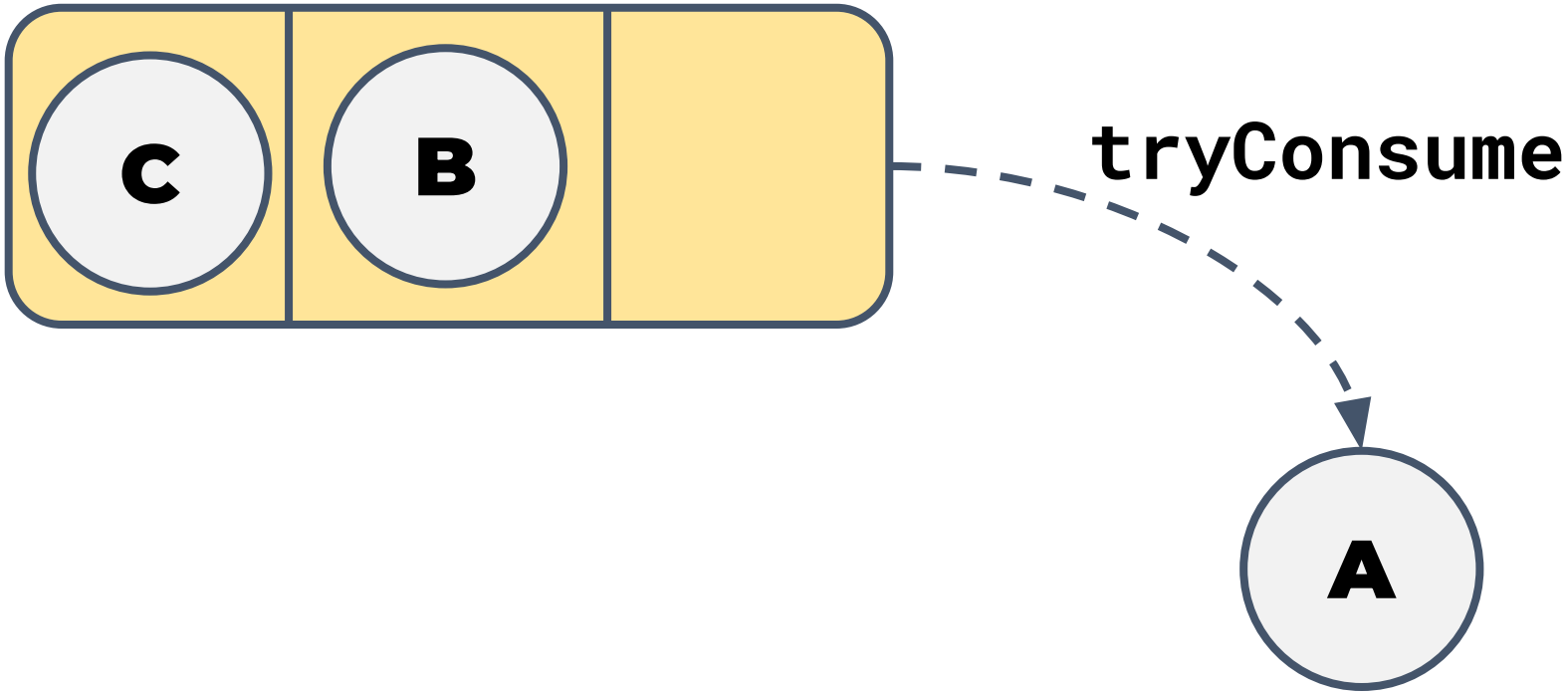
Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue



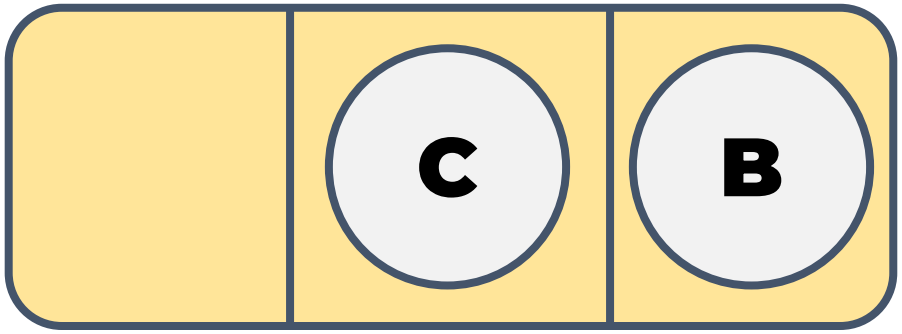
Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue



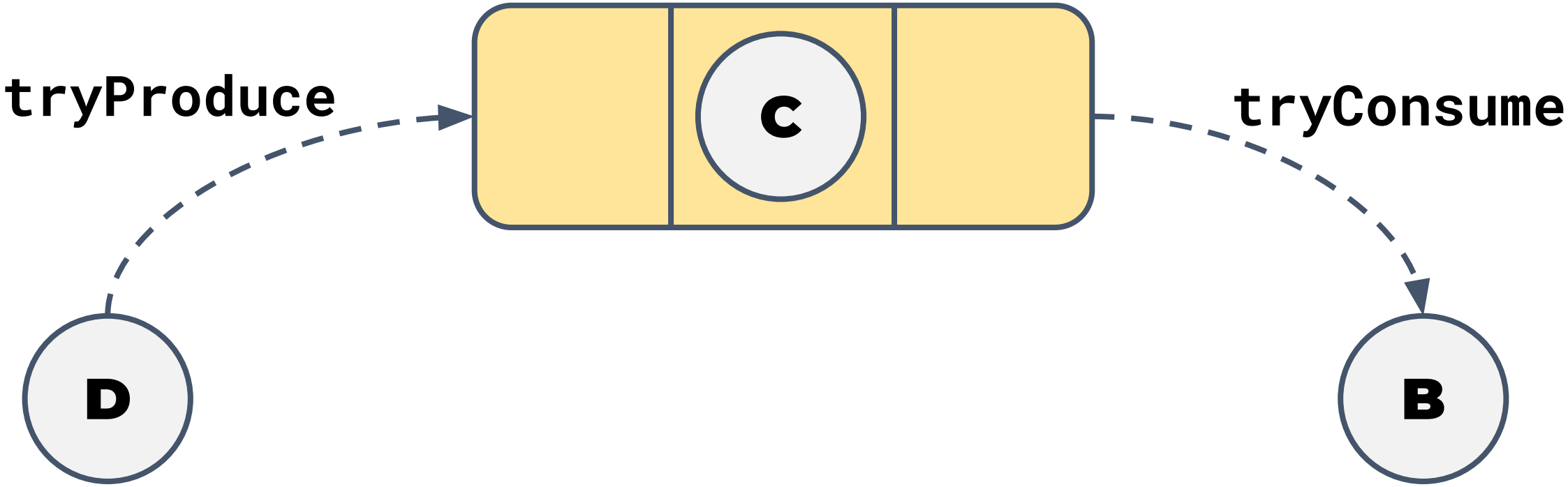
Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue



Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue



Строго один поток-производитель, строго один поток-потребитель!

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

SPSC_BoundedQueue.java

```
//метод для производителя
boolean tryProduce(E value);

//метод для потребителя
boolean tryConsume(Consumer<E> consumer);
```

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

Consuming thread

```
queue.tryConsume(element ->
    print(element)
);
```

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

Producing thread

Consuming thread
`queue.tryConsume(element -> print(element));`

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

Producing thread

`E element = new E(...);`
 	
`init(element);`

Consuming thread

`queue.tryConsume(element ->`
`  print(element)`
`);`

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

Producing thread

```
E element = new E(...);

init(element);

queue.tryProduce(element);
```

Consuming thread

```
queue.tryConsume(element ->
    print(element)
);
```

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

Producing thread

```
E element = new E(...);

init(element);

queue.tryProduce(element);
```

Consuming thread

```
queue.tryConsume(element ->
    print(element)
);
```

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

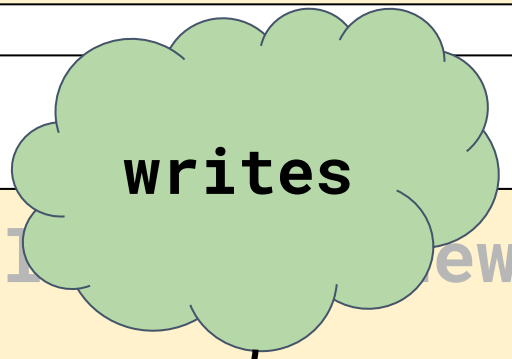
```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

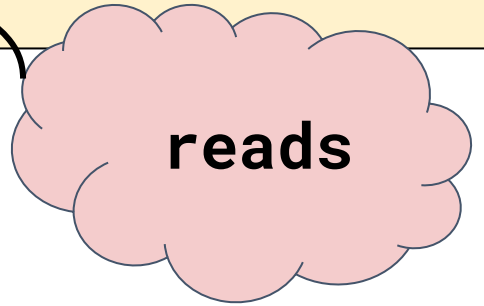
thread

```
E el ew E(...);

init(element);

queue.tryProduce(element);
```

Consuming thread

```
queue.tryConsume(element ->
    print(element)
);
```





потребитель должен видеть объект в полностью готовом виде...

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

Producing thread

`E element = new E(...);`
 	
`init(element);`
 	
`queue.tryProduce(element);`

Consuming thread

`queue.tryConsume(element ->`
`  print(element)`
`);`

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

Producing thread

```
E element = new E(...);

init(element);

queue.tryProduce(element);
```

Consuming thread

```
queue.tryConsume(element ->
    print(element)
);
```

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

Producing thread

```
E element = new E(...);

init(element);

queue.tryProduce(element);
```

Consuming thread

```
queue.tryConsume(element ->
    print(element)
);
```

***happens-before* (используя volatile-семантику)**

Acquire/Release: применение

Single Producer/Single Consumer Bounded Queue

```
SPSC_BoundedQueue<E> queue = new SPSC_BoundedQueue<>(...)
```

***causality* (используя acq-rel-семантику)**

Producing thread	Consuming thread

`E element = new E(...);`
`init(element);`
`queue.tryProduce(element);`
	`queue.tryConsume(element ->`
	`  print(element)`
	`);`

***happens-before* (используя volatile-семантику)**

Acquire/Release: применение

Бенчмарки и примеры тут: *github.com/lantalex/jpoint-2023-semantics*

Acquire/Release: применение

Две возможные реализации SPSC Bounded Queue

используя **volatile**-семантику: `SPSC_VolatileQueue`

используя **acq/rel**-семантику: `SPSC_AcqRelQueue`

Acquire/Release: применение

Две возможные реализации SPSC Bounded Queue

используя **volatile**-семантику: **SPSC_VolatileQueue**

используя **acq/rel**-семантику: **SPSC_AcqRelQueue**

ARMv8: **SPSC_AcqRelQueue** производительнее **SPSC_VolatileQueue** на ~30-40%

Acquire/Release: применение

Две возможные реализации SPSC Bounded Queue

используя **volatile**-семантику: **SPSC_VolatileQueue**

используя **acq/rel**-семантику: **SPSC_AcqRelQueue**

ARMv8: **SPSC_AcqRelQueue** производительнее **SPSC_VolatileQueue** на ~30-40%

x86: **SPSC_AcqRelQueue** производительнее **SPSC_VolatileQueue** в 2-3 раза

Acquire/Release: библиотеки

JCTools: *github.com/JCTools/JCTools*

SpscAtomicArrayQueue.java
MpmcAtomicArrayQueue.java

Disruptor: *github.com/LMAX-Exchange/disruptor*

SingleProducerSequencer.java
MultiProducerSequencer.java

Agrona: *github.com/real-logic/agrona*

OneToOneRingBuffer.java
ManyToOneRingBuffer.java

Acquire/Release: библиотеки

Используйте специализированные версии многопоточных структур данных, если есть для этого возможность (один производитель и/или один потребитель)

JCTools: *github.com/JCTools/JCTools*

`SpscAtomicArrayQueue.java`
`MpmcAtomicArrayQueue.java`

Disruptor: *github.com/LMAX-Exchange/disruptor*

`SingleProducerSequencer.java`
`MultiProducerSequencer.java`

Agrona: *github.com/real-logic/agrona*

`OneToOneRingBuffer.java`
`ManyToOneRingBuffer.java`

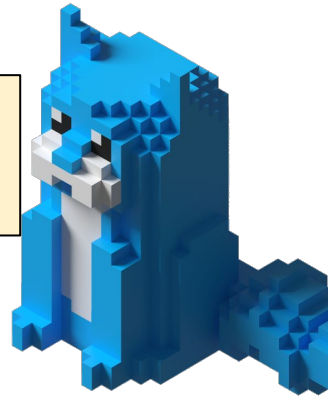
Acquire/Release vs Volatile

Кот, спорим, сегодня я быстрее тебя пробегу стометровку?

Это мы еще посмотрим!

Acquire/Release vs Volatile

```
int   red = 0;
int  blue = 0;
```

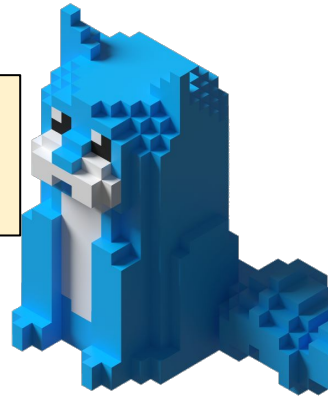


Acquire/Release vs Volatile

```
int   red = 0;
int  blue = 0;
```

Thread ‘Red Panda’

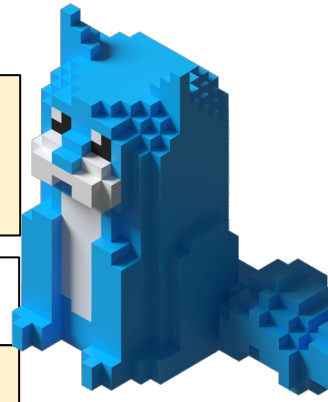
```
//панда добежала до финиша
red.setVolatile(1);
```



Acquire/Release vs Volatile

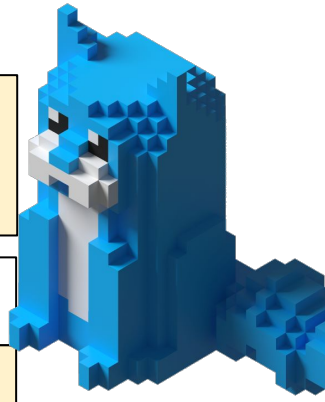
```
int   red = 0;
int  blue = 0;
```

Thread ‘Red Panda’	Thread ‘Blue Cat’
`//панда добежала до финиша` `red.setVolatile(1);`	`//котик добежал до финиша` `blue.setVolatile(1);`



Acquire/Release vs Volatile

```
int   red = 0;
int  blue = 0;
```



Thread ‘Red Panda’

```
//панда добежала до финиша
red.setVolatile(1);
```

Thread ‘Blue Cat’

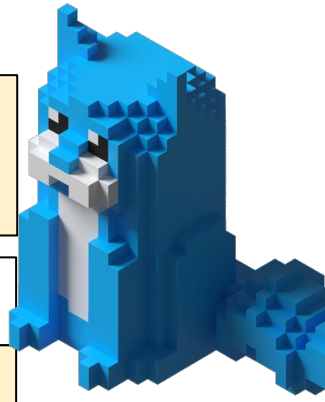
```
//котик добежал до финиша
blue.setVolatile(1);
```

Thread ‘Referee #1’

Thread ‘Referee #2’

Acquire/Release vs Volatile

```
int   red = 0;
int  blue = 0;
```



Thread ‘Red Panda’

```
//панда добежала до финиша
red.setVolatile(1);
```

Thread ‘Blue Cat’

```
//котик добежал до финиша
blue.setVolatile(1);
```

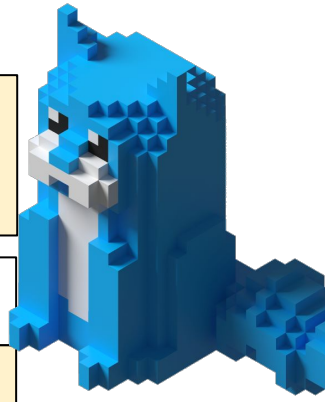
Thread ‘Referee #1’

```
if ( red.getVolatile() == 1 &&
     blue.getVolatile() == 0 ) {

     print(“Панда победила,
            отдаем приз ей”)
}
```

Thread ‘Referee #2’

Acquire/Release vs Volatile

```
int   red = 0;
int  blue = 0;
```



Thread ‘Red Panda’

```
//панда добежала до финиша
red.setVolatile(1);
```

Thread ‘Blue Cat’

```
//котик добежал до финиша
blue.setVolatile(1);
```

Thread ‘Referee #1’

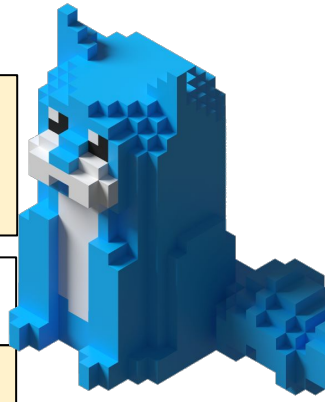
```
if ( red.getVolatile() == 1 &&
     blue.getVolatile() == 0 ) {

    print(“Панда победила,
           отдаем приз ей”)
}
```

Thread ‘Referee #2’

```
if (blue.getVolatile() == 1 &&
      red.getVolatile() == 0 ) {

    print(“Котик победил,
           отдаем приз ему”)
}
```

Acquire/Release vs Volatile

```
int   red = 0;
int  blue = 0;
```



Thread ‘Red Panda’

```
//панда добежала до финиша
red.setVolatile(1);
```

Thread ‘Blue Cat’

```
//котик добежал до финиша
blue.setVolatile(1);
```

Thread ‘Referee #1’

```
if ( red.getVolatile() == 1 &&
     blue.getVolatile() == 0 ) {

    print(“Панда победила,
            отдаем приз ей”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Thread ‘Referee #2’

```
if (blue.getVolatile() == 1 &&
      red.getVolatile() == 0 ) {

    print(“Котик победил,
            отдаем приз ему”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Acquire/Release vs Volatile

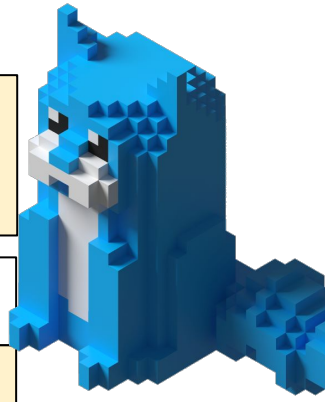
```
Referee #1: “Панда победила, отдаем приз ей”
Referee #2: “Я не знаю, кто победил”
```

```
Referee #1: “Я не знаю, кто победил”
Referee #2: “Котик победил, отдаем приз ему”
```

```
Referee #1: “Я не знаю, кто победил”
Referee #2: “Я не знаю, кто победил”
```

Acquire/Release vs Volatile

```
int   red = 0;
int  blue = 0;
```



Thread ‘Red Panda’

```
//панда добежала до финиша
red.setVolatile(1);
```

Thread ‘Blue Cat’

```
//котик добежал до финиша
blue.setVolatile(1);
```

Thread ‘Referee #1’

```
if ( red.getVolatile() == 1 &&
     blue.getVolatile() == 0 ) {

    print(“Панда победила,
            отдаем приз ей”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Thread ‘Referee #2’

```
if (blue.getVolatile() == 1 &&
      red.getVolatile() == 0 ) {

    print(“Котик победил,
            отдаем приз ему”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Acquire/Release vs Volatile

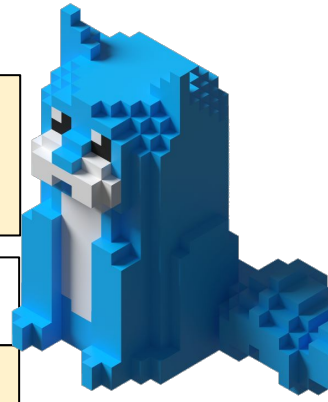
```
int   red = 0;
int  blue = 0;
```

Thread ‘Red Panda’

```
//панда добежала до финиша
red.setRelease(1) ;
```

Thread ‘Blue Cat’

```
//котик добежал до финиша
blue.setRelease(1) ;
```



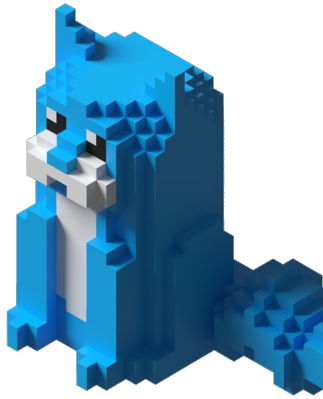
Thread ‘Referee #1’

```
if ( red.getAcquire()  == 1 &&
     blue.getAcquire()  == 0 ) {

    print(“Панда победила,
            отдаем приз ей”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Thread ‘Referee #2’

```
if (blue.getAcquire()  == 1 &&
     red.getAcquire()  == 0 ) {

    print(“Котик победил,
            отдаем приз ему”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Acquire/Release vs Volatile



```
Referee #1: “Панда победила, отдаем приз ей”
Referee #2: “Я не знаю, кто победил”
```

```
Referee #1: “Я не знаю, кто победил”
Referee #2: “Котик победил, отдаем приз ему”
```

```
Referee #1: “Я не знаю, кто победил”
Referee #2: “Я не знаю, кто победил”
```

Acquire/Release vs Volatile

```
Referee #1: “Панда победила, отдаем приз ей”
Referee #2: “Котик победил, отдаем приз ему”
```

```
Referee #1: “Панда победила, отдаем приз ей”
Referee #2: “Я не знаю, кто победил”
```

```
Referee #1: “Я не знаю, кто победил”
Referee #2: “Котик победил, отдаем приз ему”
```

```
Referee #1: “Я не знаю, кто победил”
Referee #2: “Я не знаю, кто победил”
```

Acquire/Release vs Volatile

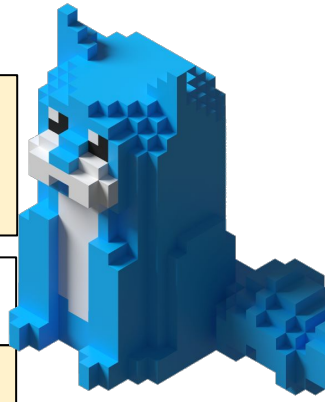
```
Referee #1: “Панда победила, отдаем приз ей”
Referee #2: “Котик победил, отдаем приз ему”
```

Referee #1:

Referee #2:

Acquire/Release vs Volatile

```
int   red = 0;
int  blue = 0;
```



Thread ‘Red Panda’

```
//панда добежала до финиша
red.setRelease(1);
```

Thread ‘Blue Cat’

```
//котик добежал до финиша
blue.setRelease(1);
```

Thread ‘Referee #1’

```
if ( red.getAcquire() == 1 &&
     blue.getAcquire() == 0 ) {

    print(“Панда победила,
            отдаем приз ей”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Thread ‘Referee #2’

```
if (blue.getAcquire() == 1 &&
      red.getAcquire() == 0 ) {

    print(“Котик победил,
            отдаем приз ему”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Acquire/Release vs Volatile

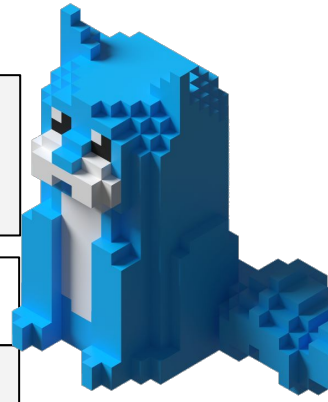
```
int   red = 0;
int  blue = 0;
```

Thread ‘Red Panda’

```
//панда добежала до финиша
red.setRelease(1);
```

Thread ‘Blue Cat’

```
//котик добежал до финиша
blue.setRelease(1);
```



Thread ‘Referee #1’

```
if ( red.getAcquire() == 1 &&
     blue.getAcquire() == 0 ) {

    print(“Панда победила,
            отдаем приз ей”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Thread ‘Referee #2’

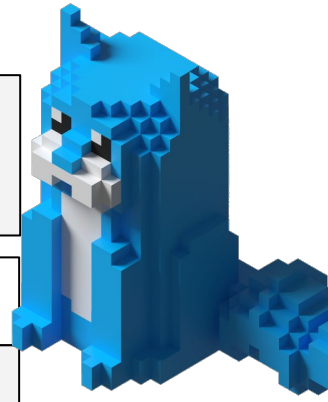
```
if (blue.getAcquire() == 1 &&
      red.getAcquire() == 0 ) {

    print(“Котик победил,
            отдаем приз ему”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Acquire/Release vs Volatile

```
int   red = 0;
int  blue = 0;
```

Panda

Cat



Thread ‘Referee #1’

```
if ( red.getAcquire() == 1 &&
     blue.getAcquire() == 0 ) {

    print(“Панда победила,
            отдаем приз ей”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Thread ‘Referee #2’

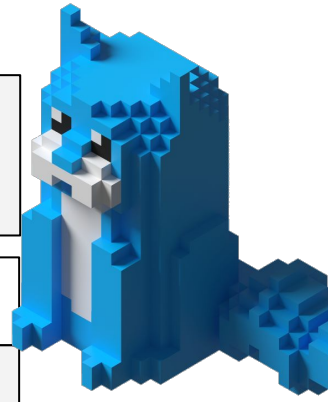
```
if (blue.getAcquire() == 1 &&
      red.getAcquire() == 0 ) {

    print(“Котик победил,
            отдаем приз ему”)
} else {
   print(“Я не знаю, кто победил”)
}
```

Acquire/Release vs Volatile

```
int   red = 0;
int  blue = 0;
```

Panda

Cat



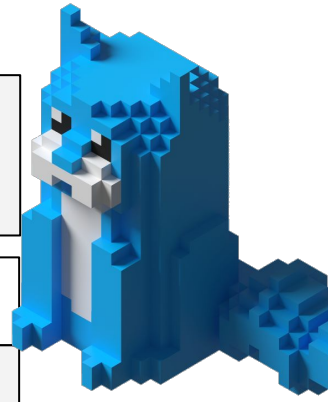
Thread ‘Referee #1’

Referee 1

Thread ‘Referee #2’

Referee 2

Acquire/Release vs Volatile



```
int   red = 0;
int  blue = 0;
```

Panda

Cat

Относительность - история событий не абсолютна, а зависит от конкретного читателя!

Referee 1

Referee 2

Volatile: гарантия консенсуса

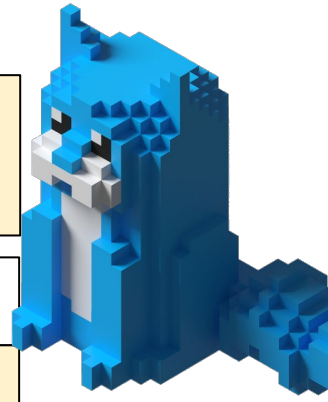
```
int   red = 0;
int  blue = 0;
```

Thread ‘Red Panda’

```
//панда добежала до финиша
red.setVolatile(1);
```

Thread ‘Blue Cat’

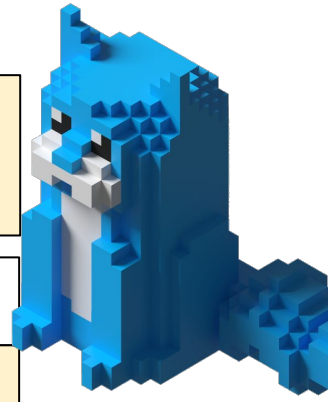
```
//котик добежал до финиша
blue.setVolatile(1);
```



Volatile: гарантия консенсуса

```
int   red = 0;
int  blue = 0;
```

Thread ‘Red Panda’	Thread ‘Blue Cat’
`//панда добежала до финиша` `red.setVolatile(1);`	`//котик добежал до финиша` `blue.setVolatile(1);`



Consensus: все операции чтения/записи с volatile-семантикой глобально упорядочены

Многопоточные библиотеки

Используйте специализированные версии многопоточных структур данных, если есть для этого возможность (один производитель и/или один потребитель)

JCTools: *github.com/JCTools/JCTools*

`SpscAtomicArrayQueue.java`
`MpmcAtomicArrayQueue.java`

Disruptor: *github.com/LMAX-Exchange/disruptor*

`SingleProducerSequencer.java`
`MultiProducerSequencer.java`

Agrona: *github.com/real-logic/agrona*

`OneToOneRingBuffer.java`
`ManyToOneRingBuffer.java`

Карта семантик

plain	⊆	opaque	⊆	acq-rel	⊆	volatile
▪ type-safety ▪ OoTA-safety*		▪ progress ▪ coherence ▪ bitwise atomicity		▪ causality		

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

Карта семантик

plain	⊆	opaque	⊆	acq-rel	⊆	volatile
▪ type-safety ▪ OoTA-safety*		▪ progress ▪ coherence ▪ bitwise atomicity		▪ causality		▪ consensus

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

Карта семантик

plain	⊆ opaque	⊆ acq-rel	⊆ volatile
▪ type-safety ▪ OoTA-safety*	▪ progress ▪ coherence ▪ bitwise atomicity	▪ causality	▪ consensus *happens-before*

Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)

- Введение
- Что такое «семантики»?
- Можно ли обойтись без семантик?
- Существующие семантики
- Как задавать семантики в своем коде
- Opaque
- Acquire/Release

Итоги

- Если производительность не критична, то щедрая обсыпка сомнительного кода **`volatile`** - вполне нормальное решение
- Семантика - спецификатор операции чтения/записи, который дает дополнительные гарантии (какие именно - зависит от выбранной семантики)
- Если нет новых семантик - поведение программы полностью описывается JMM
- **`VarHandle`** - позволяет выбирать семантику при операциях записи/чтения
- **`Opaque`**-семантика ограничивает оптимизации компилятора, но не CPU
- **`Acquire/Release`** - “почти” happens-before, подходит для случаев одного писателя
- Используйте специализированные версии многопоточных структур данных, если есть для этого возможность (один производитель и/или один потребитель)

Спасибо за внимание!

Все примеры и бенчмарки

github.com/lantalex/jpoint-2023-semantics

МИР Plat.Form работает на российской платформе *AXIOM JDK*