




# Java Memory Model

**ORACLE<sup>®</sup>**

Sergey Kuksenko  
Java Platform Performance

6+



The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.

The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# Agenda

- Для чего?
- Из-за чего?
- Как
- Примеры
- Сколько стоит



# "Requirements for Programming Language Memory Models"

Jeremy Manson and William Pugh


“If programmers don’t know what their code is doing, programmers won’t be able to know what their code is doing wrong.”

# Agenda

- Для чего?
- Из-за чего?
- Как
- Примеры
- Сколько стоит



# Memory wall

- 1980
  - скорость одной операции CPU ~ скорость одного доступа к памяти
- 2010
  - скорость CPU выросла более чем в **10000** раз
  - скорость памяти выросла в **~10** раз 

# The Free Lunch Is Over

- 1980
  - скорость одной операции CPU ~ скорость одного доступа к памяти
- 2010
  - скорость CPU выросла более чем в **10000** раз
  - скорость памяти выросла в **~10** раз 😞
- 2005 Herb Sutter “The Free Lunch Is Over”
  - <http://www.gotw.ca/publications/concurrency-ddj.htm>
  - увеличиваем количество CPU

## В итоге

- Сложная архитектура современных CPU
  - многоядерность
  - OoO (Out of Order execution)
  - Store buffers (+ store forwarding)
  - Сложная иерархия кешей
  - Протоколы когерентности кешей (MESI, MOESI, MESIF, ...)
  - NUMA, ccNUMA
- “Умные” компиляторы



# Пример 1

Начальные значения:  $A == B == 0$

Thread 1	Thread 2
$r1 = B;$ $A = 1;$	$r2 = A;$ $B = 1;$

# Пример 1

Начальные значения:  $A == B == 0$

Thread 1	Thread 2
$r1 = B;$ $A = 1;$	$r2 = A;$ $B = 1;$

Какие значения могут быть для  $r1$  и  $r2$ ?

# Пример 1

Начальные значения:  $A == B == 0$

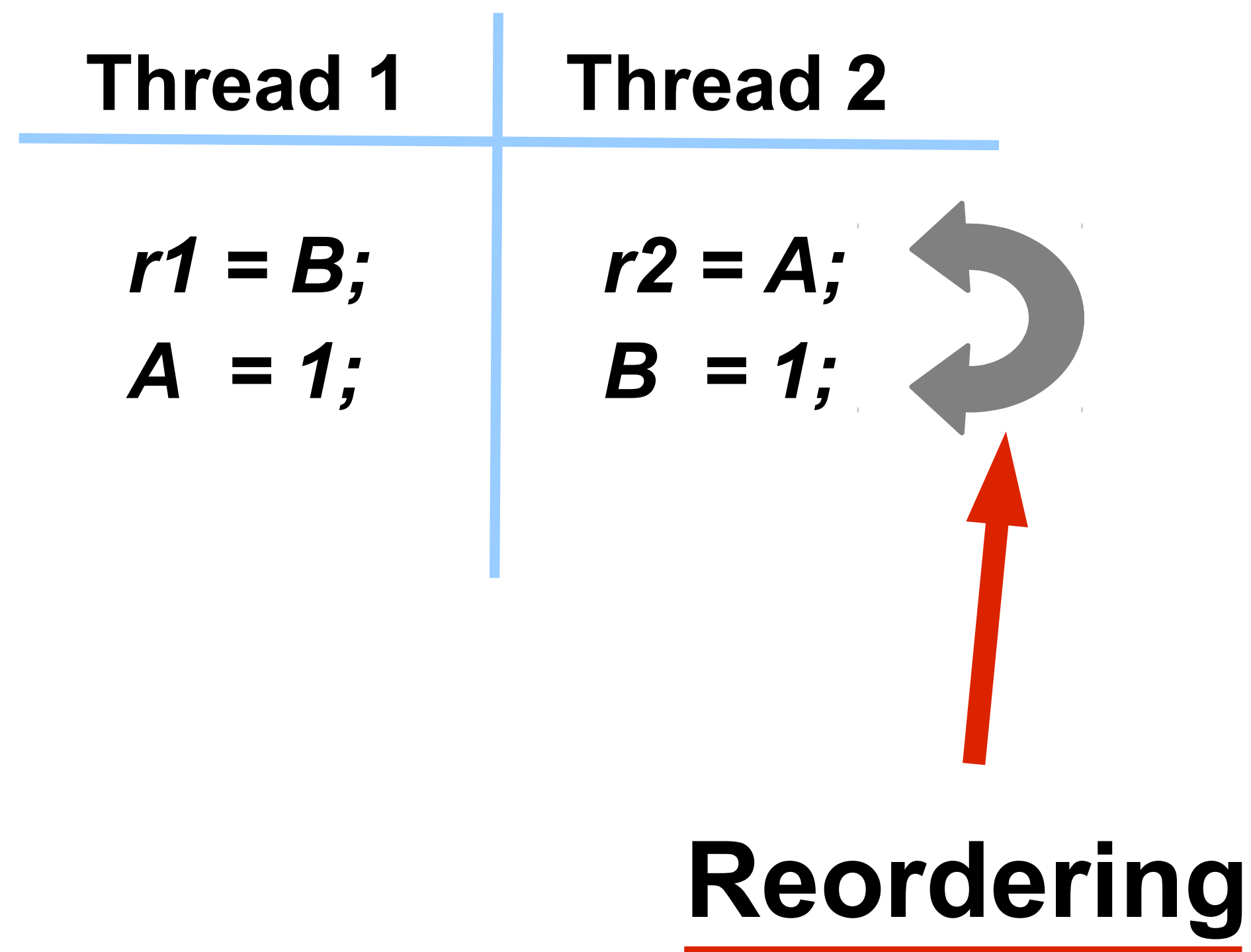
Thread 1	Thread 2
$r1 = B;$ $A = 1;$	$r2 = A;$ $B = 1;$

Какие значения могут быть для  $r1$  и  $r2$ ?

- $\langle r1, r2 \rangle$
- 1)  $\langle 0, 0 \rangle ?$
  - 2)  $\langle 0, 1 \rangle ?$
  - 3)  $\langle 1, 0 \rangle ?$
  - 4)  $\langle 1, 1 \rangle ?$

# Пример 1

Начальные значения:  $A == B == 0$



Какие значения могут быть для  $r1$  и  $r2$ ?

- $\langle r1, r2 \rangle$
- 1)  $\langle 0, 0 \rangle +$
  - 2)  $\langle 0, 1 \rangle +$
  - 3)  $\langle 1, 0 \rangle +$
  - 4)  $\langle 1, 1 \rangle +$

# Пример 2

Начальные значения:  $A == B == 0$

Thread 1	Thread 2	Thread 3
$A = 1;$	$while( A \neq 1 );$ $B = 1;$	$while( B \neq 1 );$ $r1 = A;$

## Пример 2

Начальные значения:  $A == B == 0$

Thread 1	Thread 2	Thread 3
$A = 1;$	$while( A != 1 );$ $B = 1;$	$while( B != 1 );$ $r1 = A;$

Результат:  
 $r1 == 1$   
или (xor) ?  
 $r1 == 0$

# Пример 2

Начальные значения:  $A == B == 0$

Thread 1	Thread 2	Thread 3
$A = 1;$	$while( A != 1 );$ $B = 1;$	$while( B != 1 );$ $r1 = A;$

Результат:  
 $r1 == 1$   
или (or) !  
 $r1 == 0$

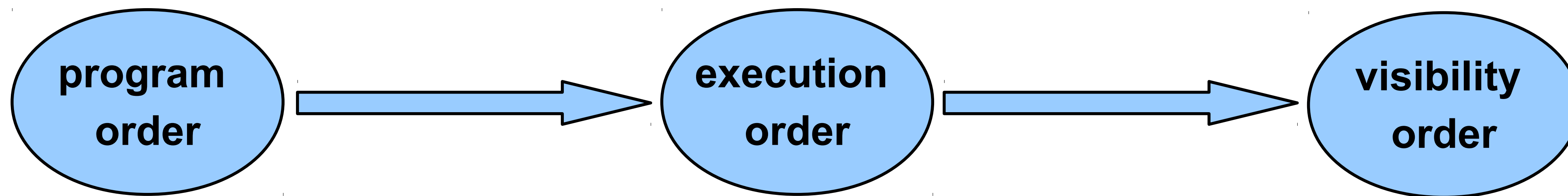
## Visibility

Возможно:

Thread 2 видит write A первым потоком

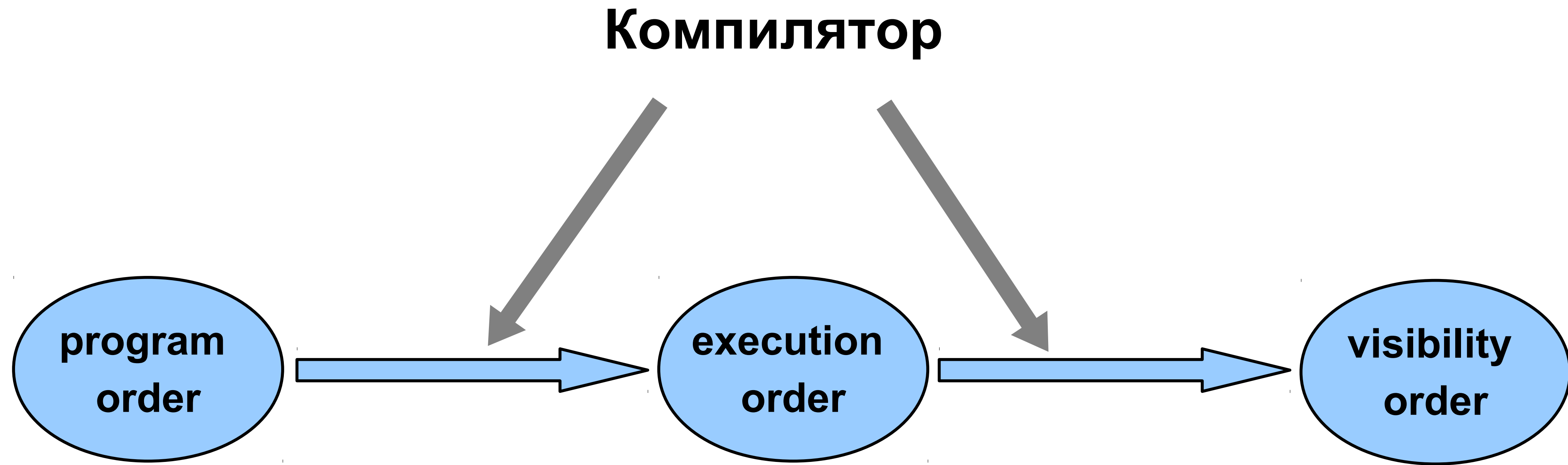
Thread 3 видит write B вторым потоком **ПЕРЕД** тем как увидит write A

# КТО ВИНОВАТ?

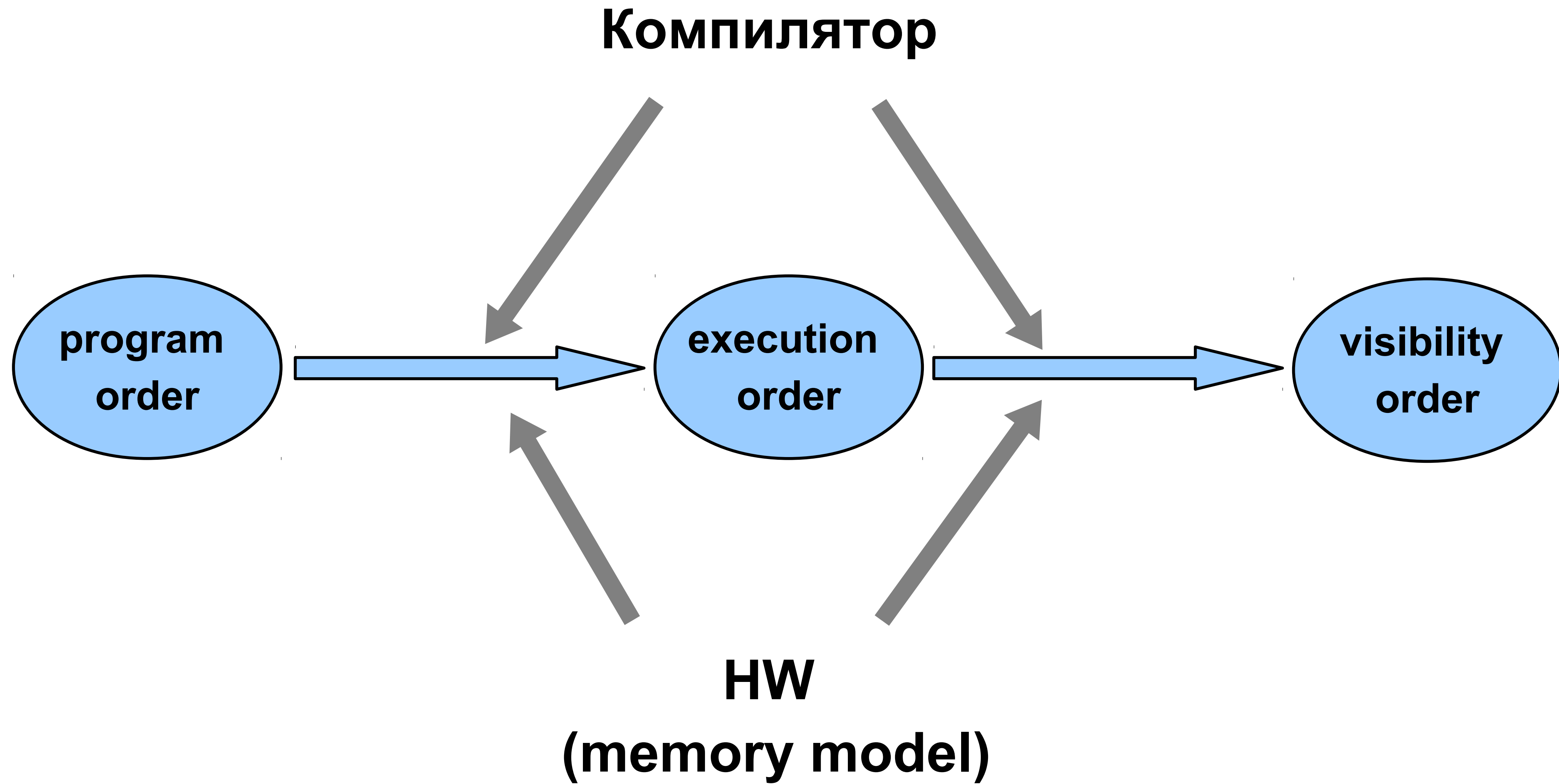




# Кто виноват?



# Кто виноват?



**Def: “операция завершилась”**

**A=1;**

**...**

**Как определить,  
что “операция завершилась”?**

**Def: “операция завершилась”**

**A=1;**

**...**

**Как определить,  
что “операция завершилась”?**

- А) Началось исполнение следующей операции.**
- Б) Значение оказалось в основной памяти.**
- Ц) Результат записи виден в точке использования.**

~~Def: “операция завершилась”~~

A=1;

...

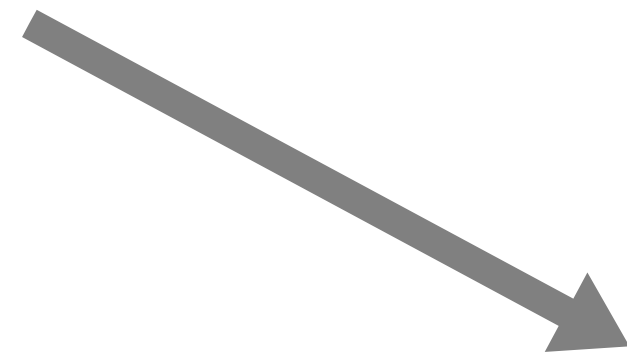
~~Как определить,  
что “операция завершилась”?~~

- ~~А) Началось исполнение следующей операции.~~
- ~~Б) Значение оказалось в основной памяти.~~
- Ц) Результат записи виден в точке использования.

# Атомарность

- Atomicity

*int A = 0xABCDFFFF;*



## Нарушения атомарности:

- Целевая платформа не содержит операции 32–битной записи (e.g. только 16 бит) => отдельная запись старшей и младшей частей.
- x86: адрес A не выровнен относительно размера.

# Основные свойства модели памяти

- Atomicity (атомарность)
- Visibility (видимость)
- Ordering (порядок)

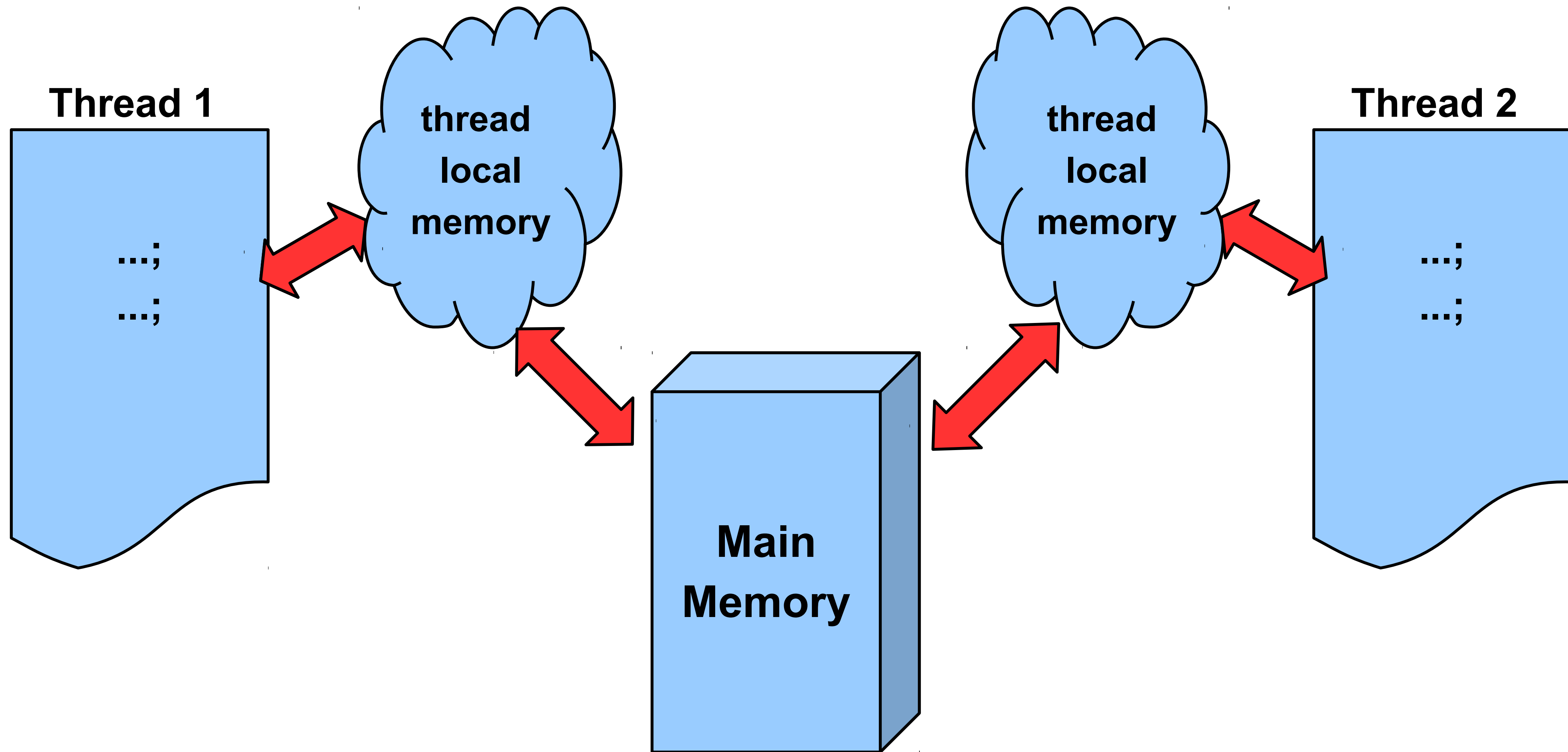
# Agenda

- **Для чего?**
- **Из-за чего?**
- **Как!**
- **Примеры**
- **Сколько стоит**

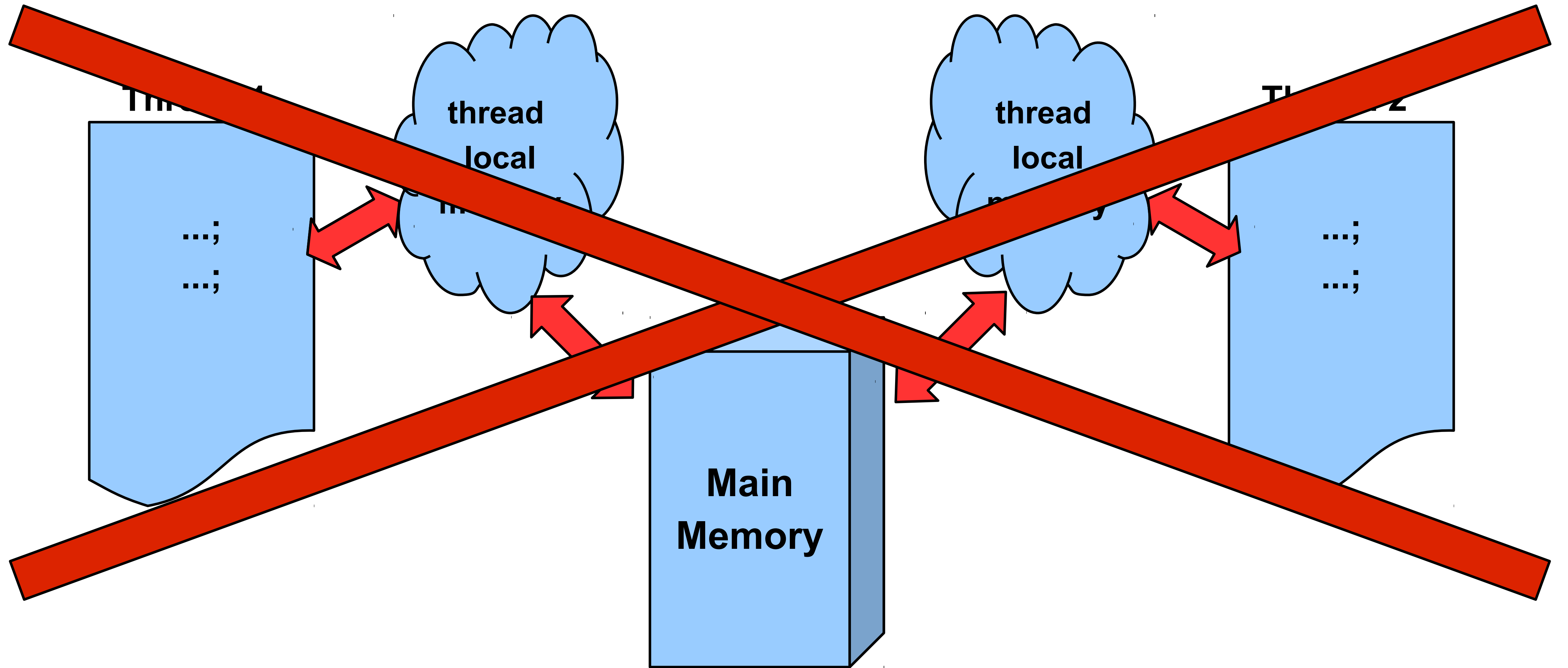




# Старая JMM



# Старая JMM



# JMM

- Переменные:
  - static field, instance field, array element
- Операции:
  - чтение/запись обычных переменных (`read/write`)
  - чтение/запись `volatile` переменных (`volatile read/write`)
  - синхронизация (`lock/unlock`)

# volatile arrays?

- `volatile A[] array;`
- `volatile` – не транзитивно:
  - `...=array;`                      – volatile read
  - `array=...;`                      – volatile write
  - `array[i]=...;`                  – обычный write
- А если очень нужно?
  - *java.util.concurrent.atomic*  
(*AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray*)

# Атомарность

- Операции чтения/записи являются атомарными.
- No out of thin-air values:
  - Всякое чтение переменной возвратит либо значение по умолчанию либо значение записанное (где-либо) в эту переменную.

# Атомарность

- Исключение:
  - Допускается неатомарное чтение/запись для типов `long/double`.
  - Чтение/запись `volatile long/double` **обязано** быть атомарным.

# Атомарность

- Часто встречаемая ошибка:
  - Для `volatile long/double` **ТОЛЬКО** операции чтения/записи являются атомарными!
  - `v++`, `v--` – неатомарные операции!
- Что делать?
  - *synchronized*
  - *java.util.concurrent.atomic*

# Visibility

- Отношение *happens-before*:
  - Если  $X$  *happens-before*  $Y$ , то  ~~$X$  выполнится раньше~~ и  $Y$  будет видеть результат  $X$ .



# Visibility (happens-before в пределах потока)

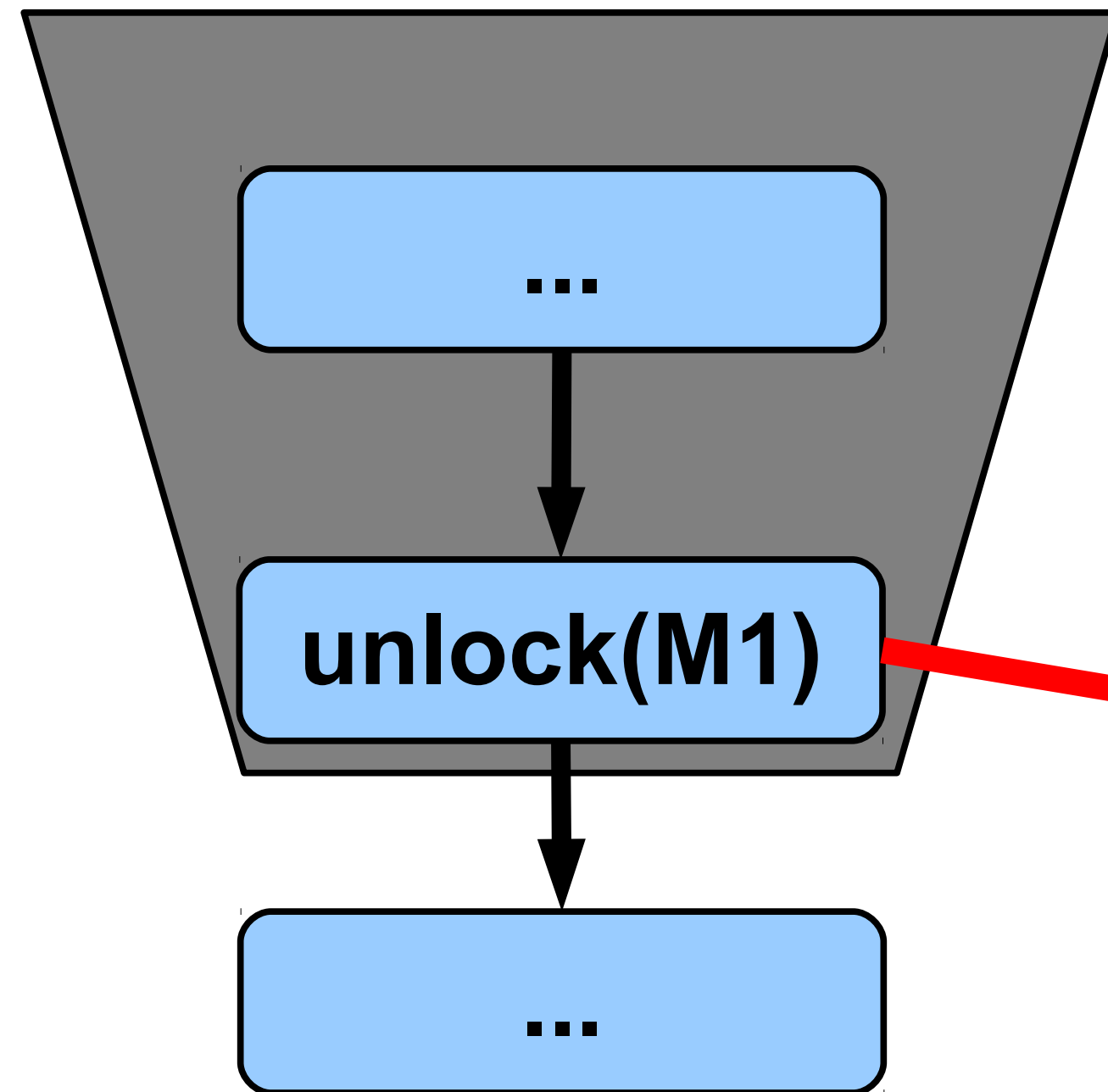
- В пределах одного потока:
  - Все операции имеют отношение *happens-before* в соответствии с *program order*

# Visibility (happens-before)

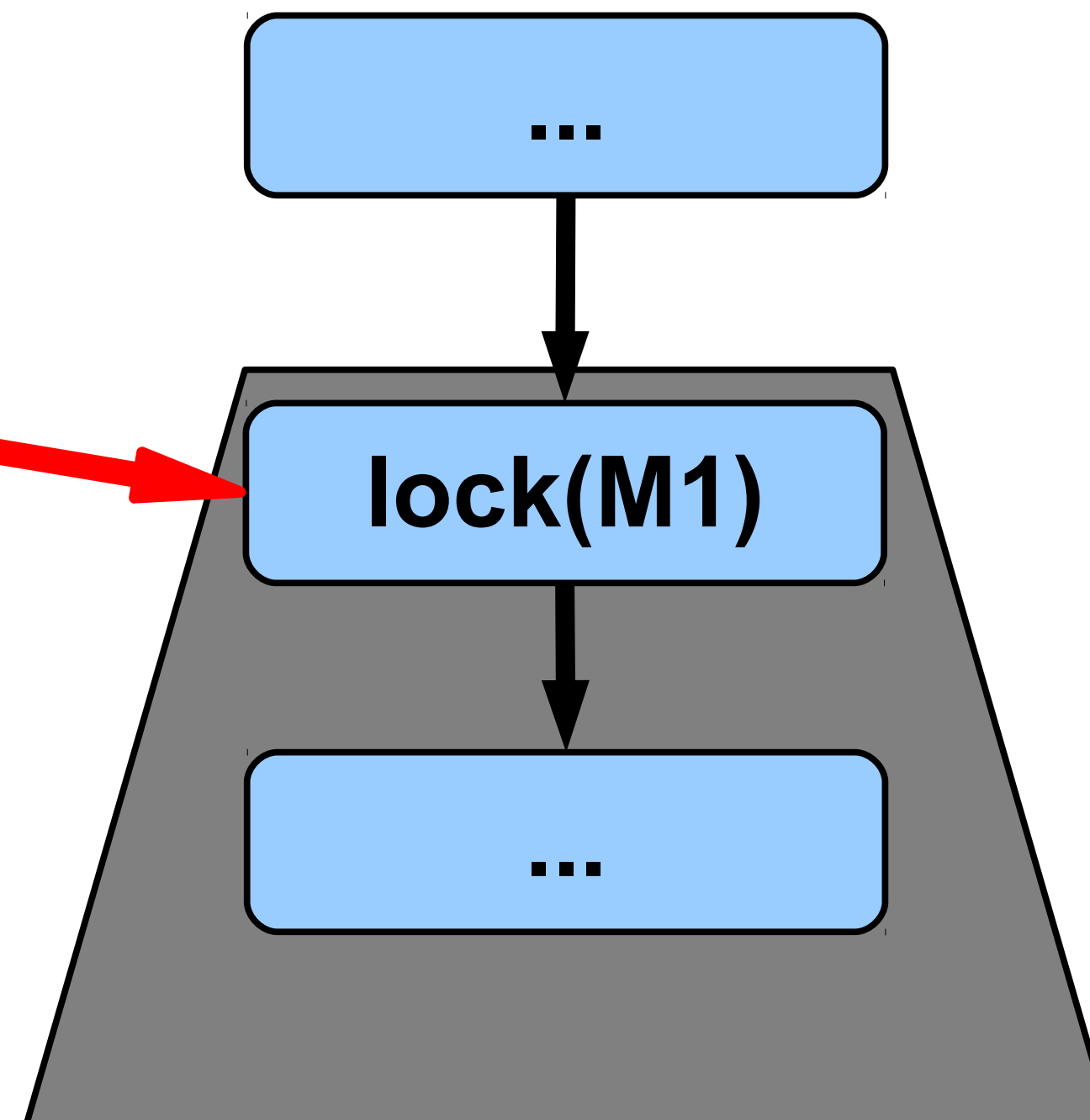
- happens-before транзитивно:
  - Если  $X$  happens-before  $Y$  и  $Y$  happens-before  $Z$ , то  $X$  happens-before  $Z$ .

# Visibility (happens-before между потоками)

Thread 1



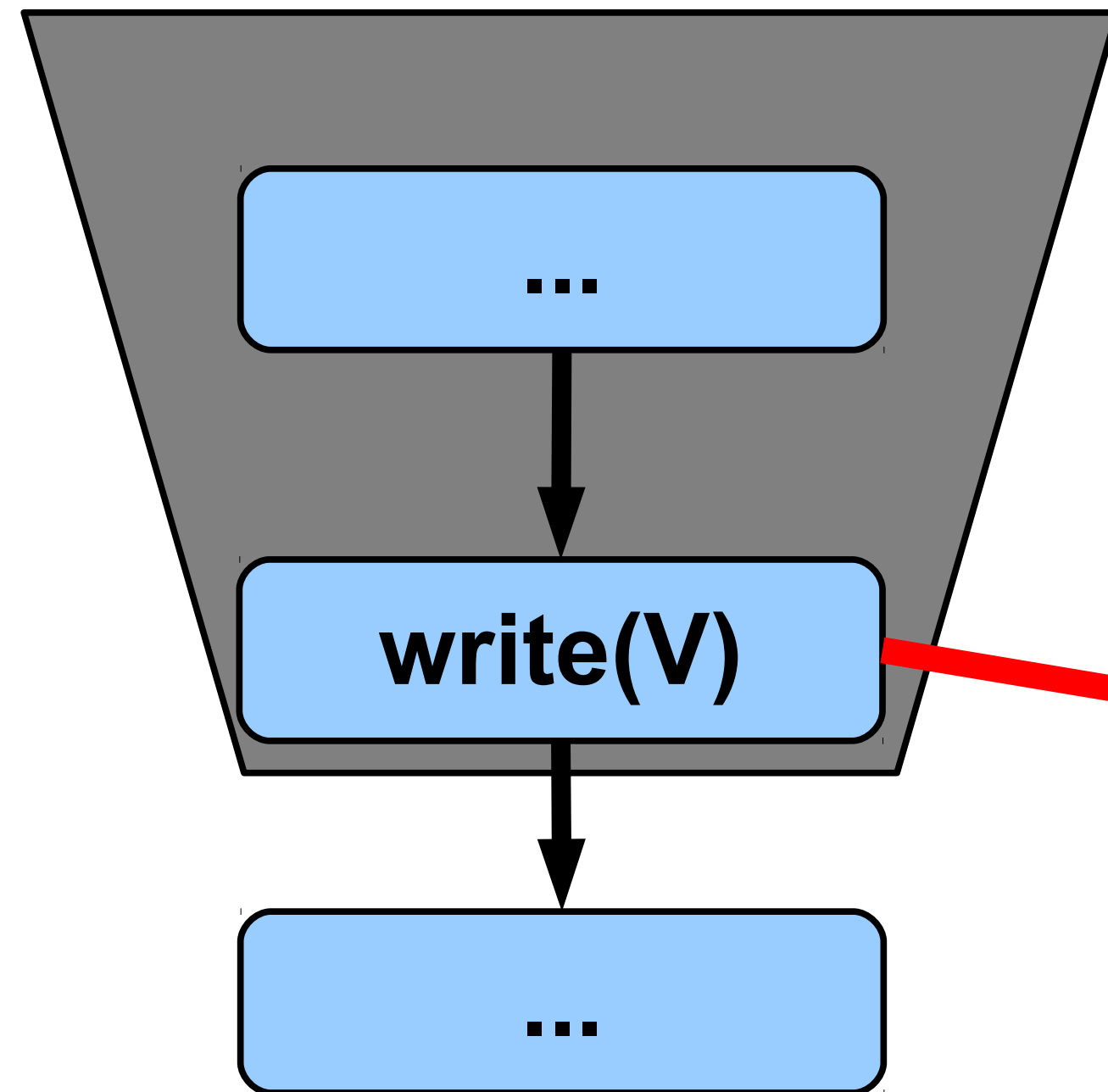
Thread 2



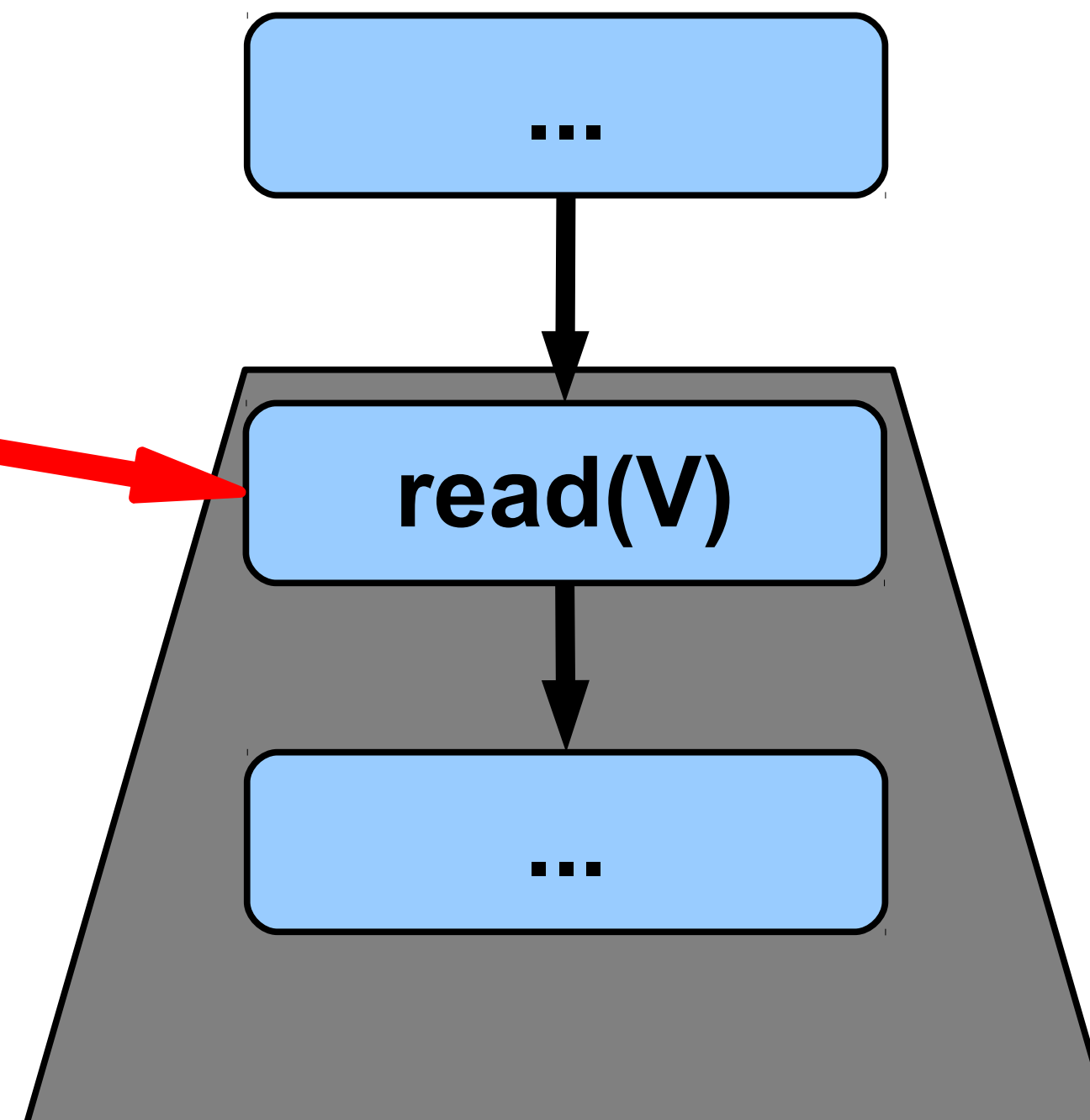
**На одном  
и том же  
мониторе!**

# Visibility (happens-before между потоками)

Thread 1



Thread 2



**V** - volatile

**На одной  
и той же  
переменной!**

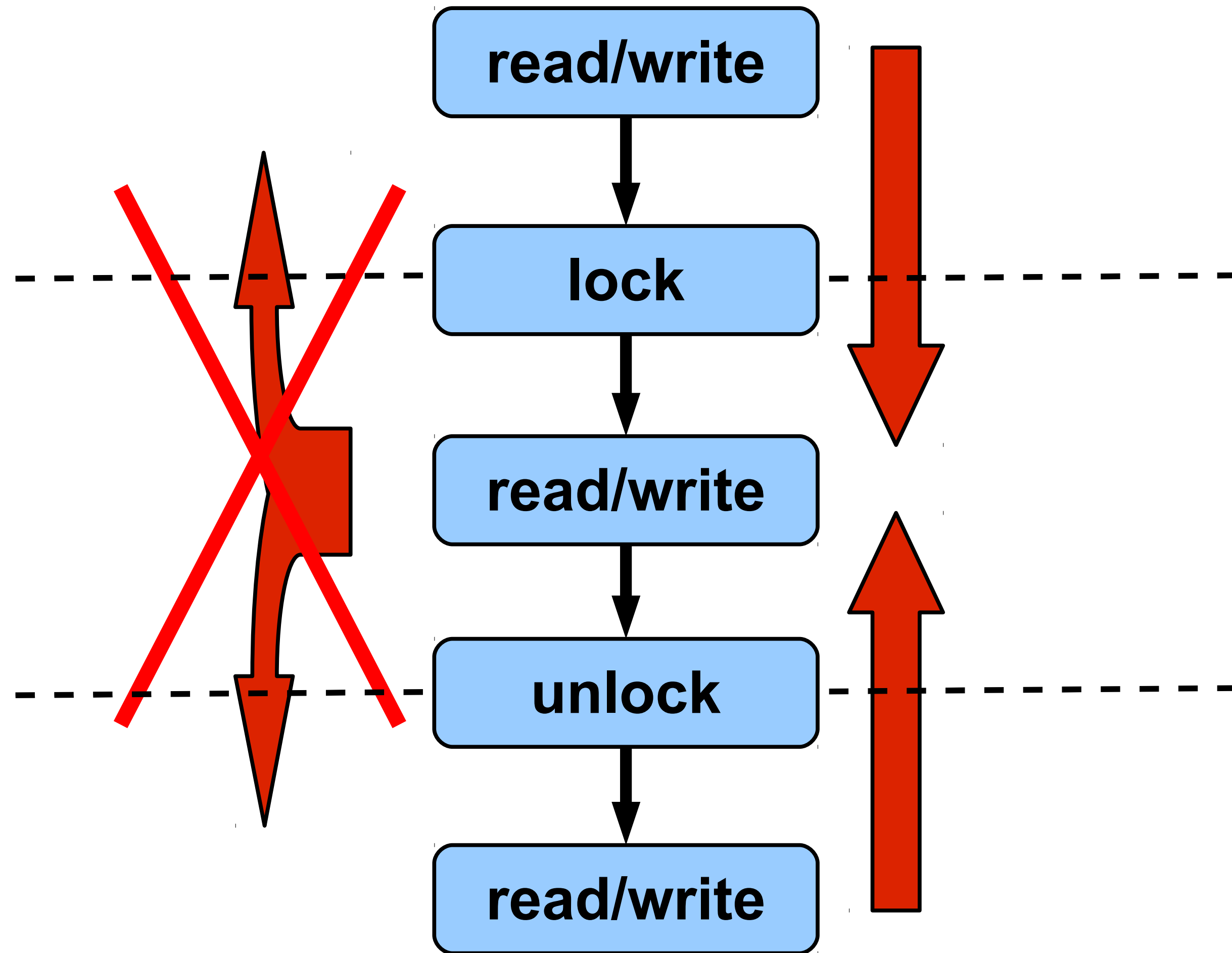
# Visibility (happens-before между потоками)

- *thread.start()* → первое действие в потоке
- Последнее действие в потоке → *join()*, *isAlive()*
- Запись *default value* в любую переменную → первое действие в потоке

# Reordering

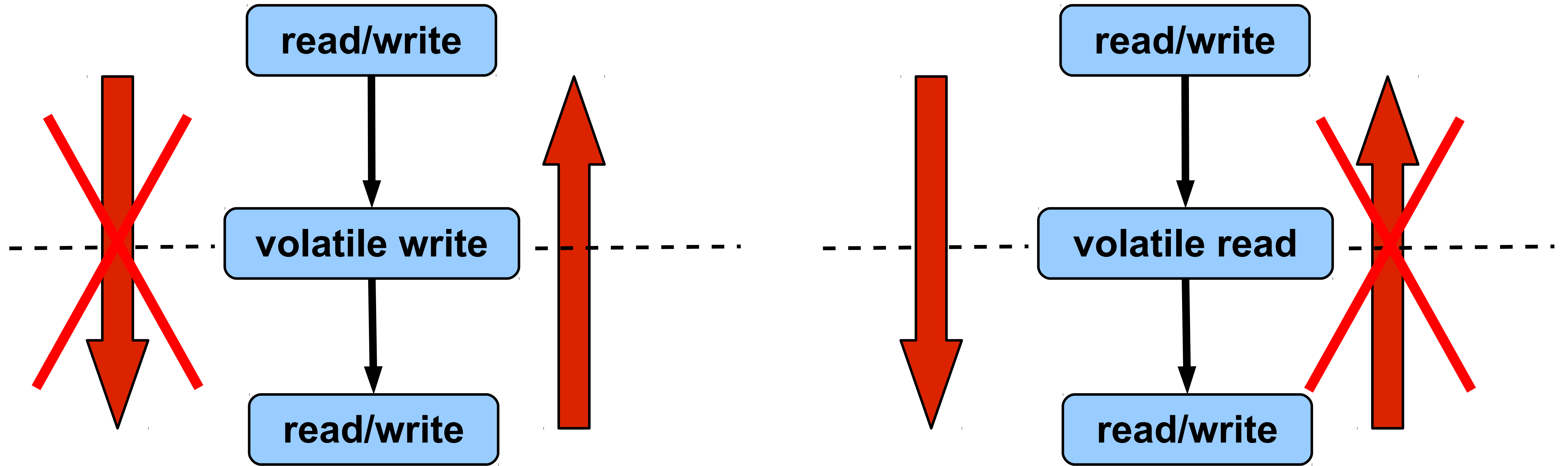
- Допустимые перестановки внутри одного треда:
  - Обычные `read/write` с точностью до зависимости по данным.
- `Volatile read/write` и `lock/unlock` НЕ могут переупорядочиваться.

# Reordering



Roach motel

# Reordering





# Пример 3

Начальные значения:

```
int A = 0;
```

```
boolean ready = false;
```

Thread 1	Thread 2
<pre>A = 41;</pre>	<pre>while(!ready);</pre>
<pre>A = 42;</pre>	<pre>System.out.println(A);</pre>
<pre>ready = true;</pre>	
<pre>A = 43;</pre>	

## Что будет напечатано?

- НИЧЕГО
- 0
- 41
- 42
- 43

# Пример 3

Начальные значения:

```
int A = 0;
```

```
boolean ready = false;
```

Thread 1	Thread 2
<pre>A = 41;</pre>	<pre>while(!ready);</pre>
<pre>A = 42;</pre>	<pre>System.out.println(A);</pre>
<pre>ready = true;</pre>	
<pre>A = 43;</pre>	

## Что будет напечатано?

- ничего +
- 0 +
- 41 +
- 42 +
- 43 +

# Пример 4

Начальные значения:

```
int A = 0;
```

```
volatile boolean ready = false;
```

Thread 1	Thread 2
<pre>A = 41;</pre>	<pre>while(!ready);</pre>
<pre>A = 42;</pre>	<pre>System.out.println(A);</pre>
<pre>ready = true;</pre>	
<pre>A = 43;</pre>	

## Что будет напечатано?

- НИЧЕГО
- 0
- 41
- 42
- 43

# Пример 4

Начальные значения:

```
int A = 0;
```

```
volatile boolean ready = false;
```

Thread 1	Thread 2
A = 41;	<i>while(!ready);</i>
A = 42;	<i>System.out.println(A);</i>
ready = true;	
A = 43;	

## Что будет напечатано?

~~-ничего-~~

~~-0-~~

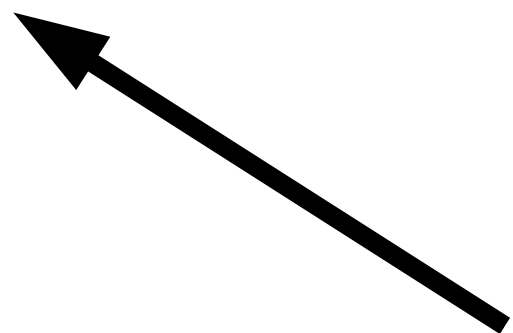
- 41

- 42 +

- 43 +

# Специальная семантика `final` полей

```
class A {  
    final B ref;  
    public A(...){  
        this.ref=...;  
    }  
    ...  
}
```

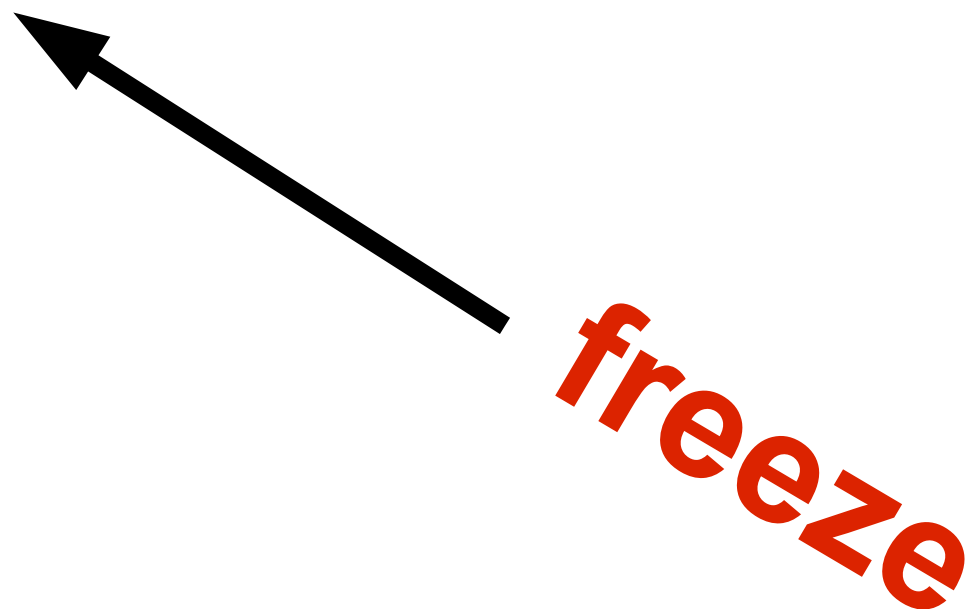


**freeze**

- После завершения конструктора любой тред видит значения записанные в *final* поле
  - а также все дерево объектов начиная с этого поля (*dereference-chain*).

# Специальная семантика `final` полей

```
class A {  
    final B ref;  
    public A(...){  
        this.ref=...;  
    }  
    ...  
}
```



- Требуется - *this* не должен убежать (escape) из конструктора.
  - e.g. регистрация *listener* в его-же конструкторе.
- Модификации в *final dereference-chain* выполненные после завершения конструктора подчиняются обычным правилам видимости.

# Agenda

- **Для чего?**
- **Из-за чего?**
- **Как**
- **Примеры**
- **Сколько стоит**



# Double-checked locking

```
class Foo {  
    private Helper helper = null;  
    public Helper getHelper() {  
        if (helper == null)  
            synchronized(this) {  
                if (helper == null)  
                    helper = new Helper();  
            }  
        return helper;  
    }  
    ...  
}
```



# Double-checked locking

```
class Foo {  
    private volatile Helper helper = null;  
    public Helper getHelper() {  
        if (helper == null)  
            synchronized(this) {  
                if (helper == null)  
                    helper = new Helper();  
            }  
        return helper;  
    }  
    ...  
}
```

# Double-checked locking (is it correct?)

```
class Foo {  
    private static volatile List<A> list = null;  
    public static List<A> getList() {  
        if (list == null)  
            synchronized(Foo.class) {  
                if (list == null)  
                    list = new ArrayList<A>();  
                list.add(...); list.add(...);  
            }  
        }  
        return list;  
    }  
}
```

# Yet another fail

```
class Foo {  
    private A something;  
    public synchronized void setSomething(B b) {  
        A a = evaluate_A_from_B();  
        this.something = a;  
    }  
    public A getSomething() {  
        return this.something;  
    }  
}
```

# Yet another fail (fix 1)

```
class Foo {  
    private A something;  
    public synchronized void setSomething(B b) {  
        A a = evaluate_A_from_B();  
        this.something = a;  
    }  
    public synchronized A getSomething() {  
        return this.something;  
    }  
}
```

# Yet another fail (fix 2)

```
class Foo {  
    private volatile A something;  
    public synchronized void setSomething(B b) {  
        A a = evaluate_A_from_B();  
        this.something = a;  
    }  
    public A getSomething() {  
        return this.something;  
    }  
}
```

# Agenda

- **Для чего?**
- **Из-за чего?**
- **Как**
- **Примеры**
- **СКОЛЬКО СТОИТ**



# Производительность

- Как измерить скорость `volatile read/write`?
  - Влияние HW memory model
  - Влияние на оптимизации компилятора
- x86:
  - `volatile read == обычный read`
  - `volatile write?`

# Тест 1

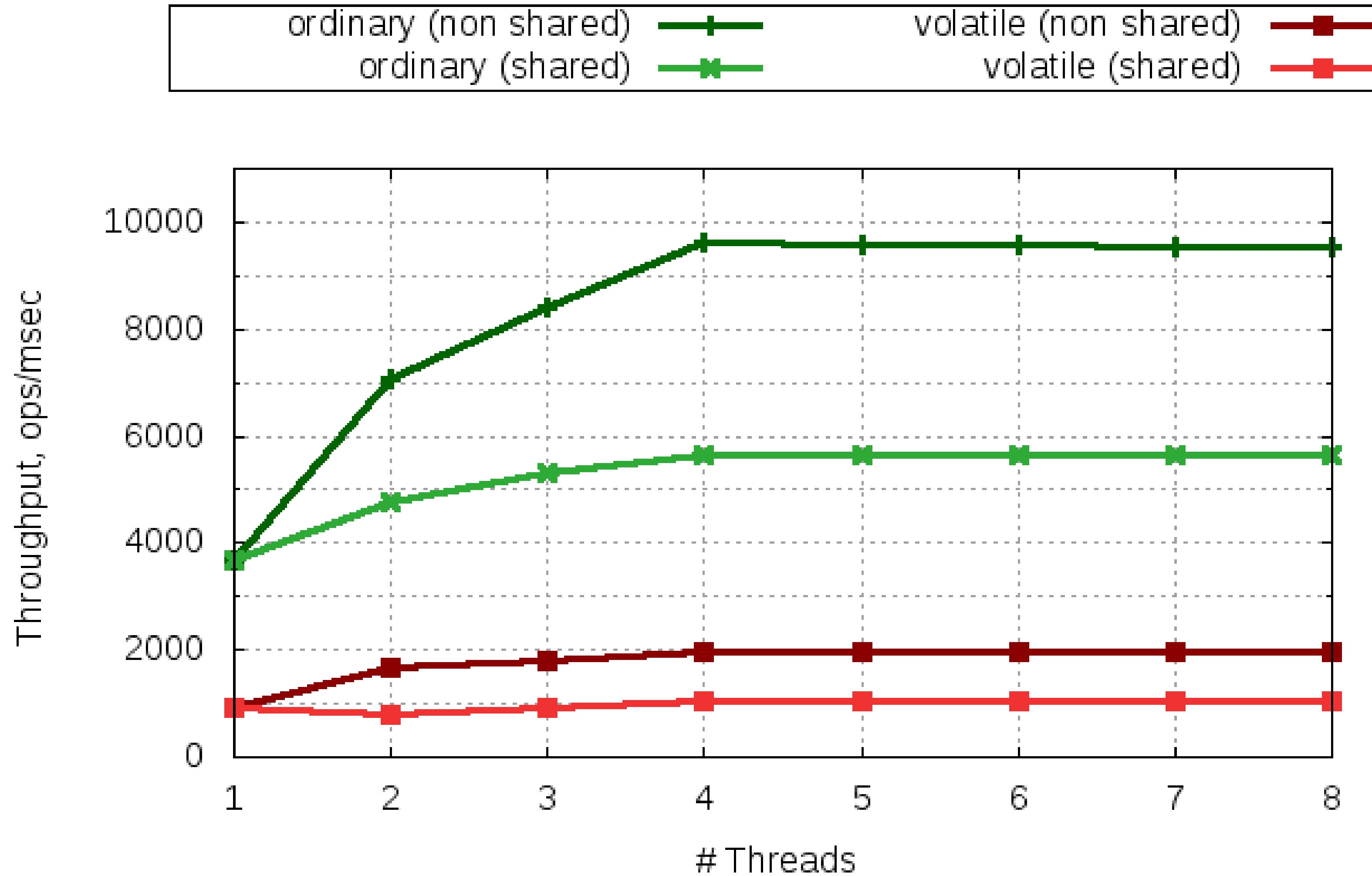
Тест не измеряет ничего реального  
1 operation :

```
for(...) { //128 times  
    v = arr[i]++;  
}
```

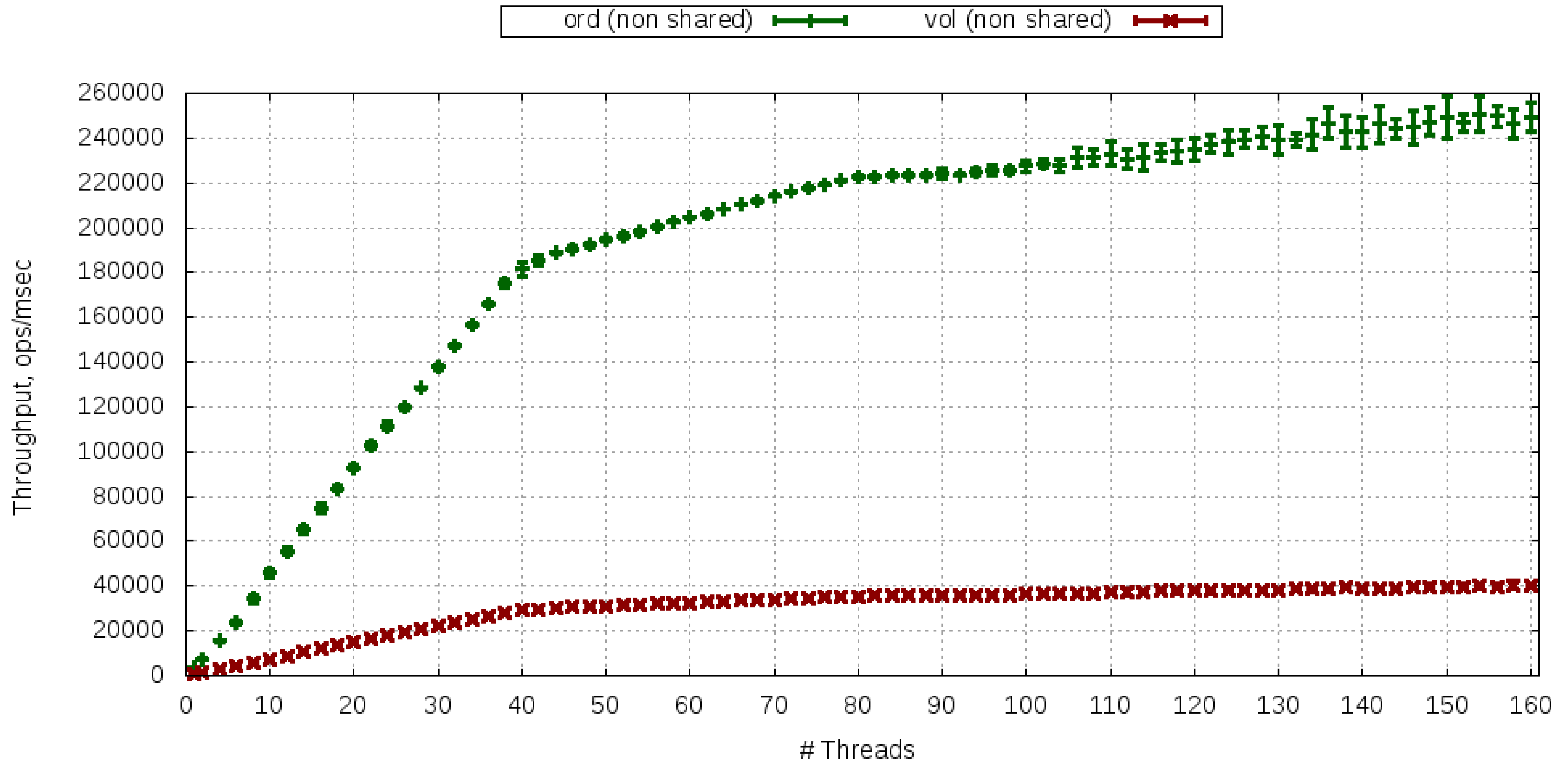
- 129 операций чтения (всегда non-shared, thread local)
- 128 операций записи (всегда non-shared, thread local)
- немного арифметики
- 128 операций записи (4 варианта теста):
  - shared / non shared
  - volatile / non volatile



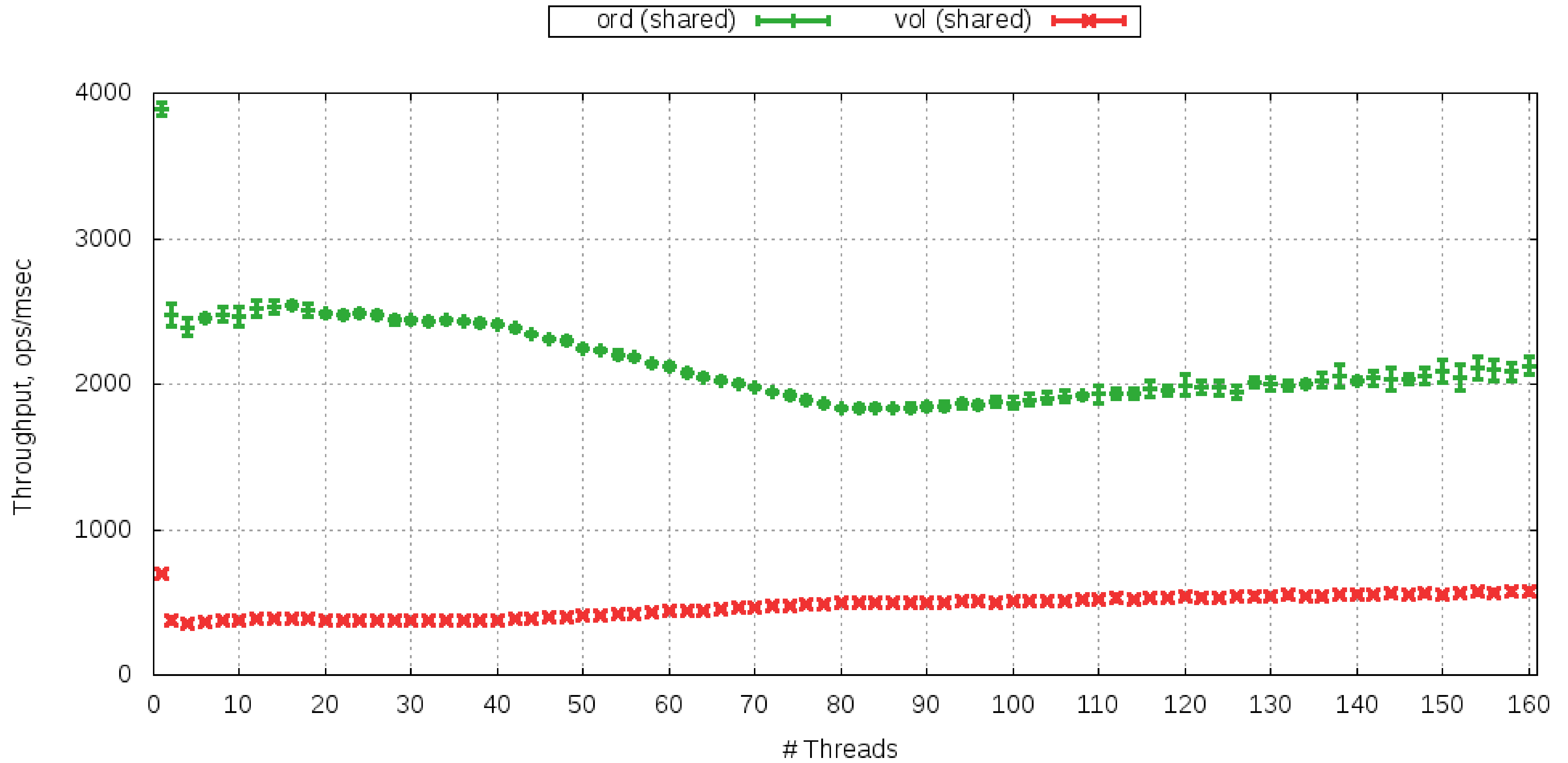
# i5 (2 core, +HT) = 4 HW threads



# Intel Westmere-EX (E7-4860) 2.27Ghz, 4x10x2 = 80 HW threads, RHEL 5.5

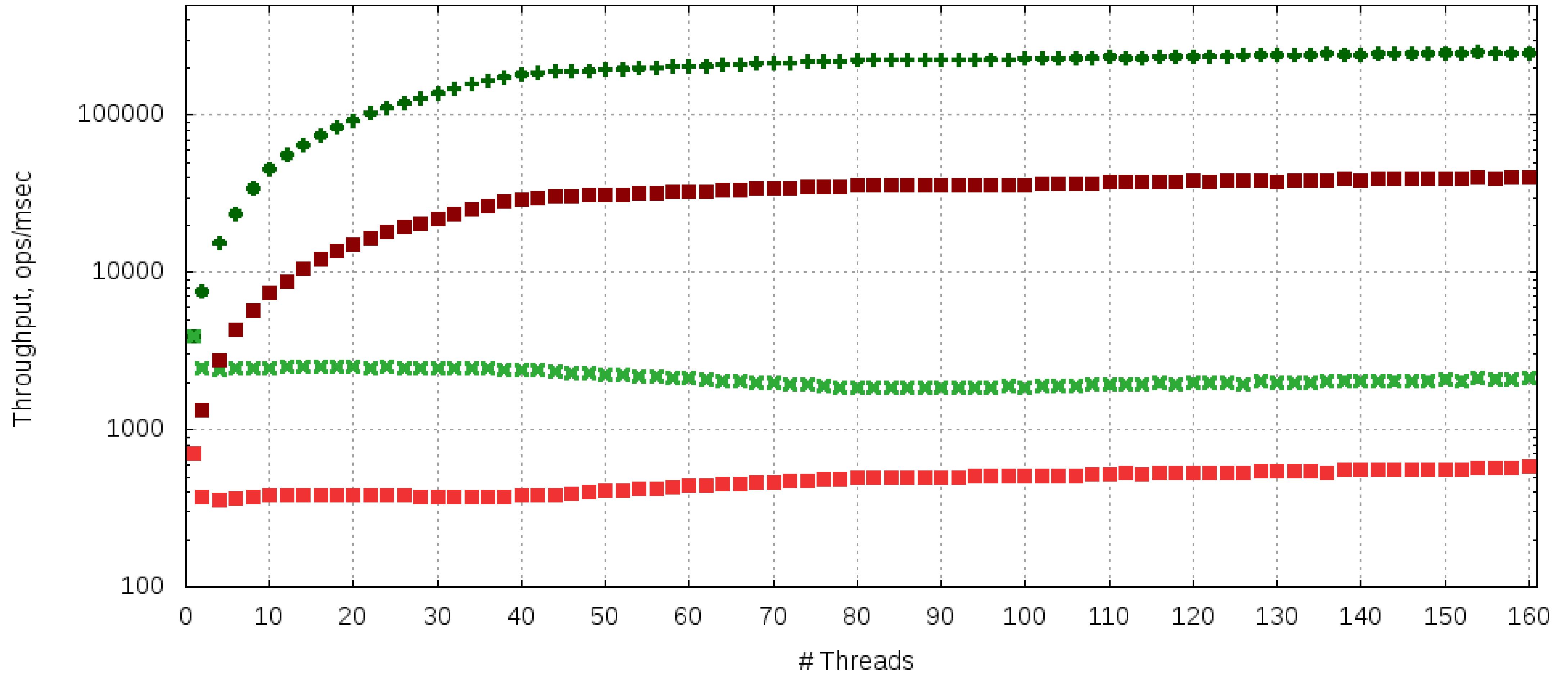


# Intel Westmere-EX (E7-4860) 2.27GHz, 4x10x2 = 80 HW threads, RHEL 5.5



# Intel Westmere-EX (E7-4860) 2.27GHz, 4x10x2 = 80 HW threads, RHEL 5.5

ord (non shared)    ord (shared)    vol (non shared)    vol (shared)



# Тест 2

```
private int[] array = new int[10000];  
...  
public int test(){  
    int s = 0;  
    for(int i=0; i<array.length; i++) {  
        s+=array[i];  
    }  
    return s;  
}
```

# Тест 2 (rev. 1)

```
private int[] array = new int[10000];
...
public int test(){
    int s = 0;
    for(int i=0; i<array.length; i++) {
        s+=array[i];
    }
    return s;
}
```

**i5 (2 core, +HT) = 4 HW threads**

**Throughput = 495 ops/msec**

## Тест 2 (rev. 2)

```
private volatile int[] array = new int[10000];
...
public int test(){
    int s = 0;
    for(int i=0; i<array.length; i++) {
        s+=array[i];
    }
    return s;
}
```

**i5 (2 core, +HT) = 4 HW threads**

**Throughput = 62 ops/msec**

## Тест 2 (rev. 3)

```
private volatile int[] array = new int[10000];  
...  
public int test(){  
    int s = 0;  
    int[] a = array;  
    for(int i=0; i<a.length; i++) {  
        s+=a[i];  
    }  
    return s;  
}
```

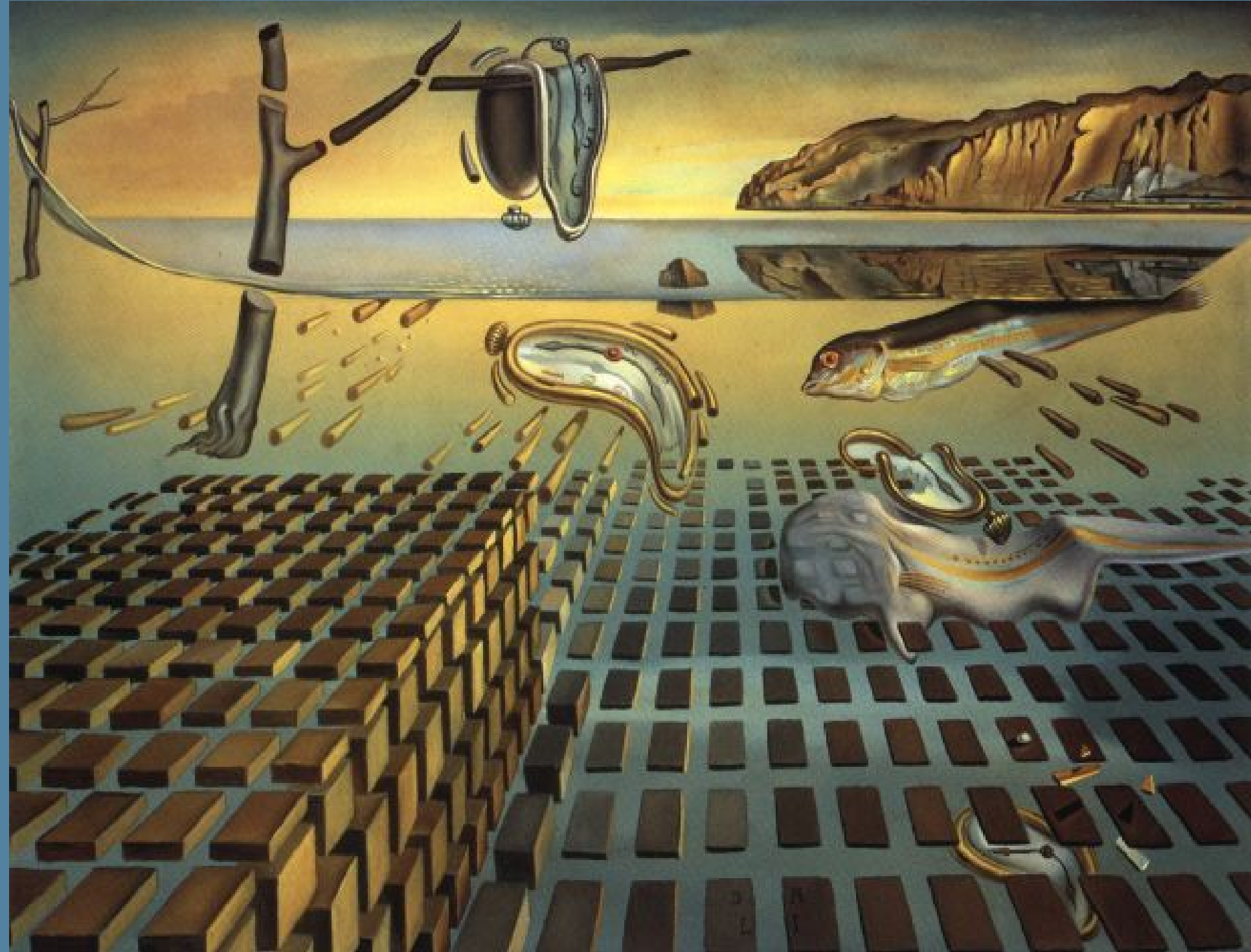
**i5 (2 core, +HT) = 4 HW threads**

**Throughput = 495 ops/msec**



# Читаем

- "Java Concurrency in Practice", Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea
- "The Art of Multiprocessor Programming", Maurice Herlihy, Nir Shavit
- "JSR 133 Cookbook"
  - <http://g.oswego.edu/dl/jmm/cookbook.html>
- "Memory Barriers: a Hardware View for Software Hackers", Paul E. McKenney
- "What Every Programmer Should Know About Memory", Ulrich Drepper
  - <http://www.akkadia.org/drepper/cpumemory.pdf>



Q&A ?