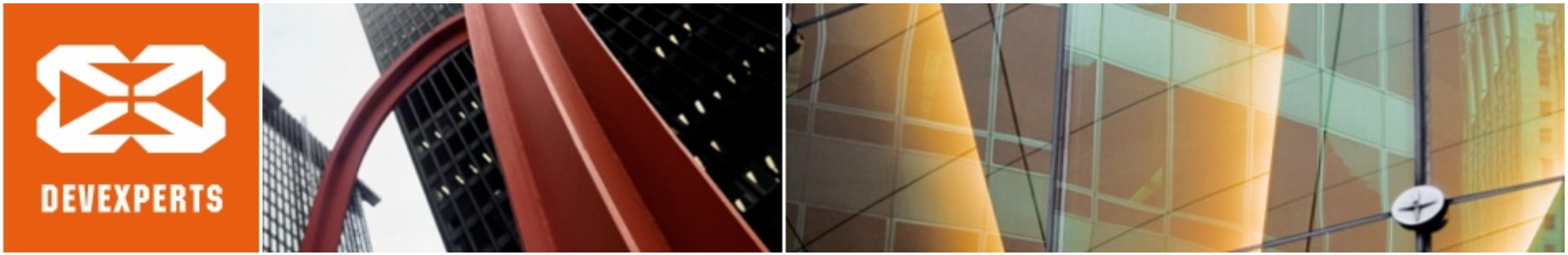
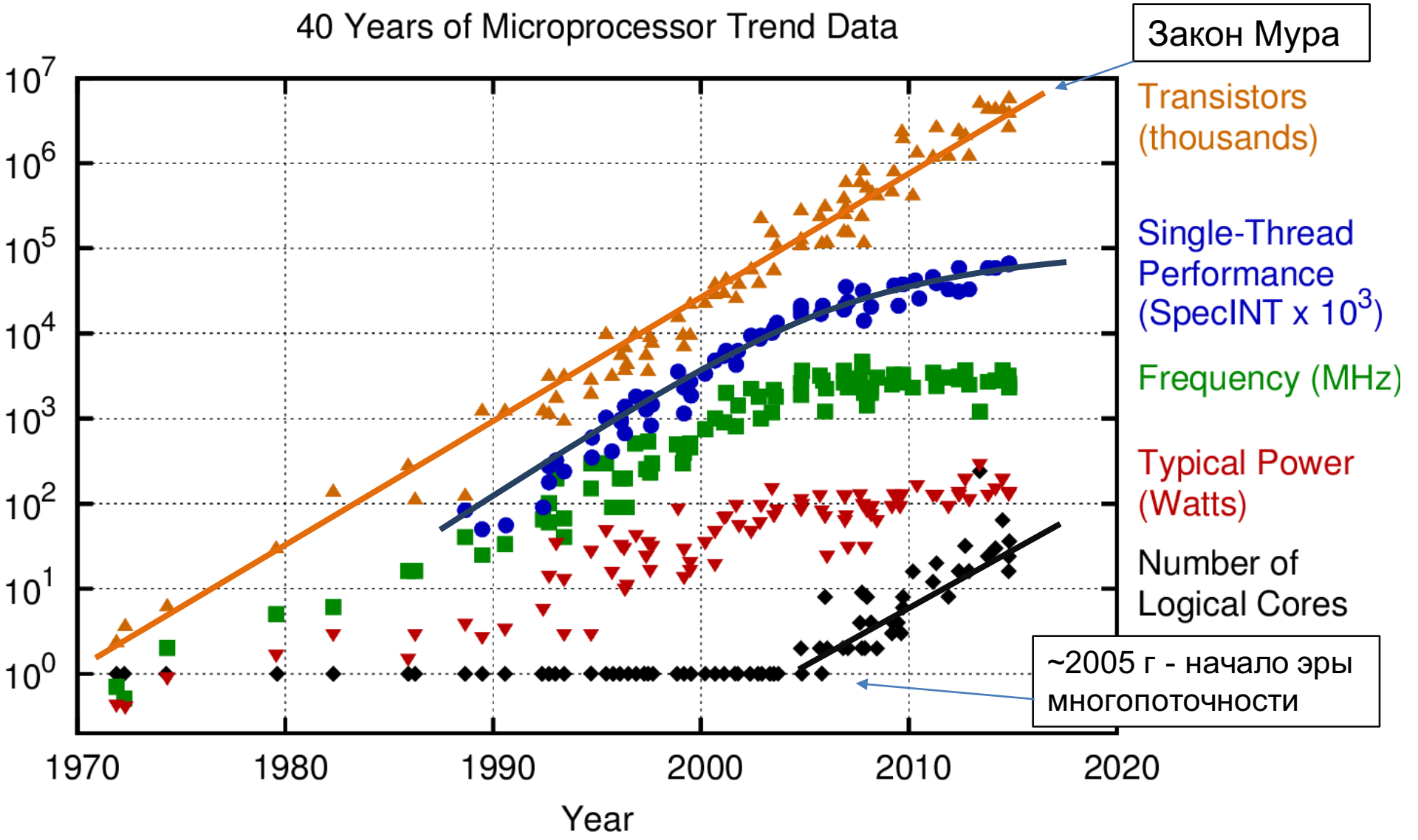



# Многопоточное программирование Теория и Практика

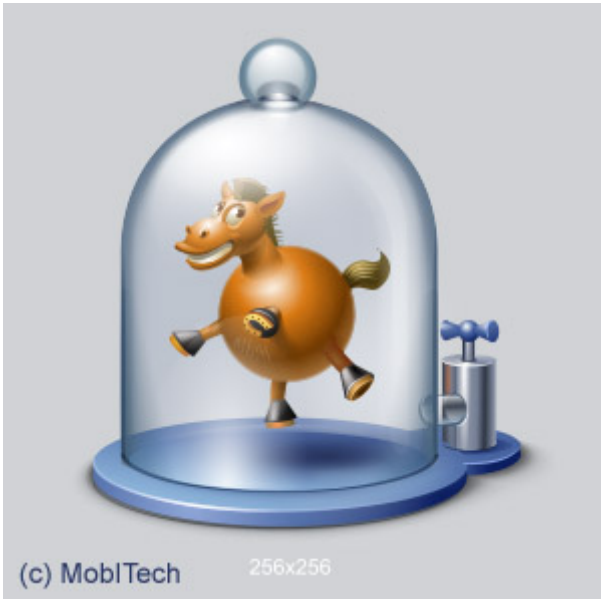




Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

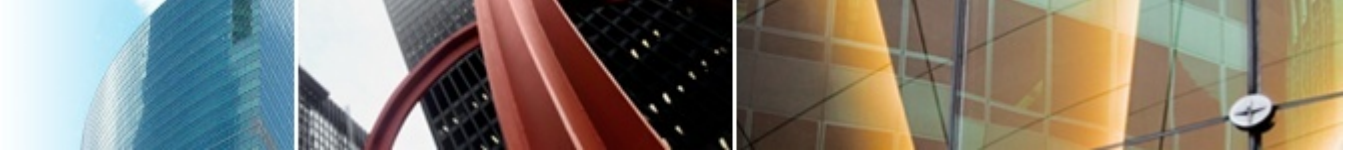
DEVEXPERTS

# Нужна формальная модель параллельных вычислений

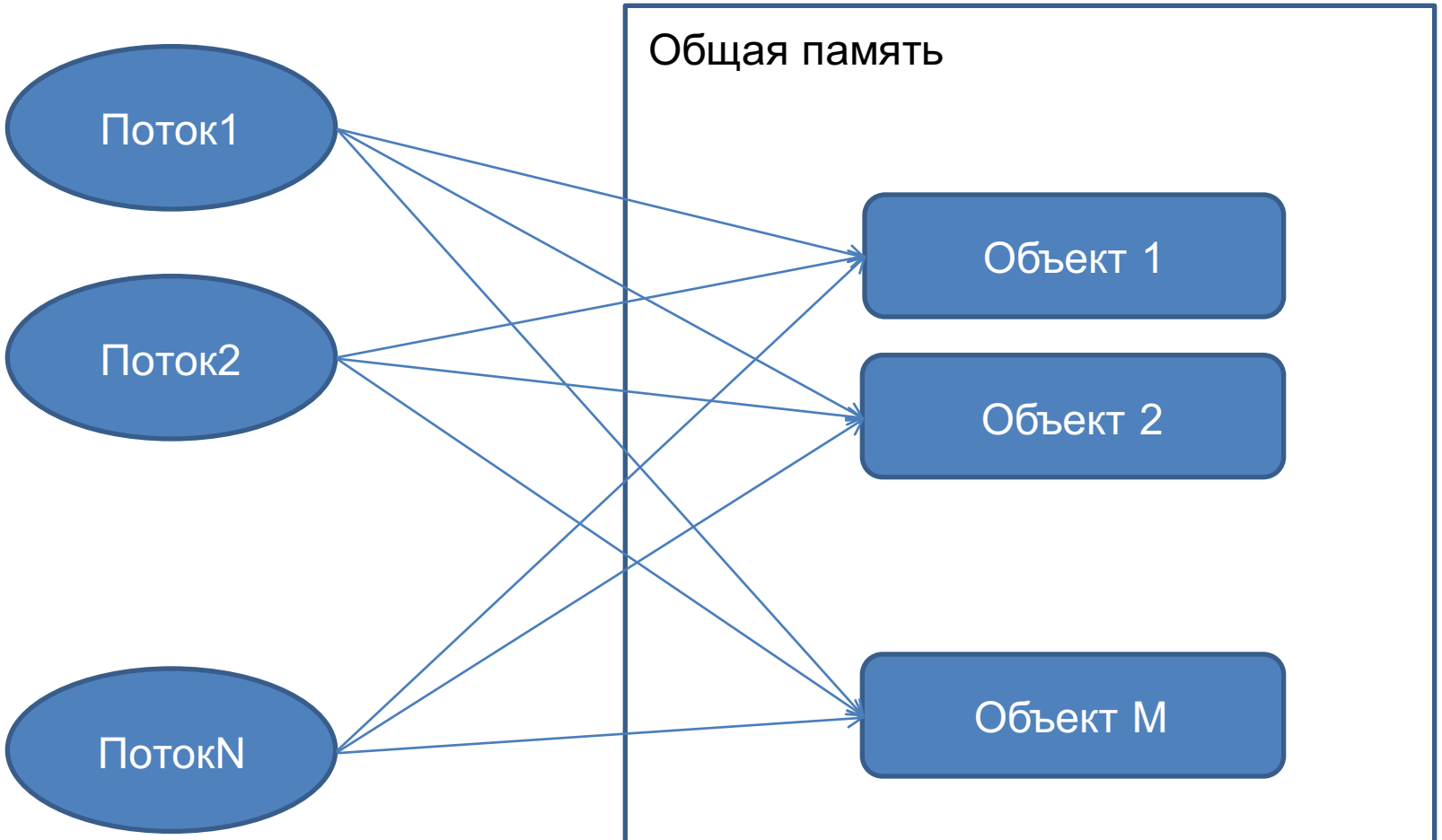
- Чтобы доказывать:
  - корректность алгоритмов
  - невозможность построения тех или иных алгоритмов
  - минимально-необходимые требования для тех или иных алгоритмов
- **Для формализации отношений между прикладным программистом и разработчиком компилятора и системы исполнения кода**



DEVEXPERTS



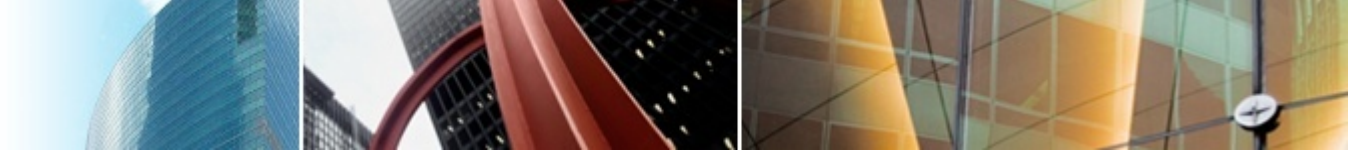
# Модель с общими объектами (общей памятью)




*Потоки и их внутренние состояния*

*Операции*

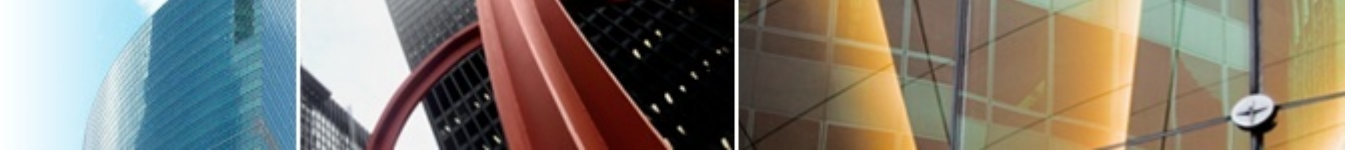
*Общие объекты*



# Общие переменные

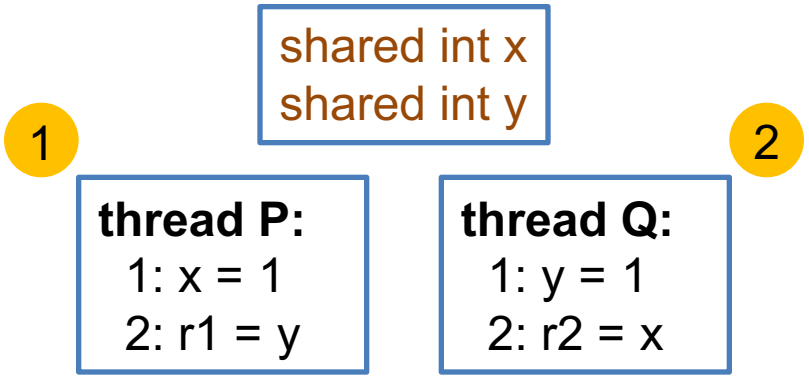
- Общие переменные – это просто простейший тип общего объекта:
  - У него есть значение определенного типа
  - Есть операция чтения (**read**) и записи (**write**).
- Общие переменные – это базовые строительные блоки для параллельных алгоритмов
- Модель с общими переменными – это хорошая абстракция современных многопроцессорных систем и многопоточных ОС
  - **На практике, это область памяти *процесса*, которая одновременно доступна для чтения и записи всем *потокам*, исполняемым в данном процессе**

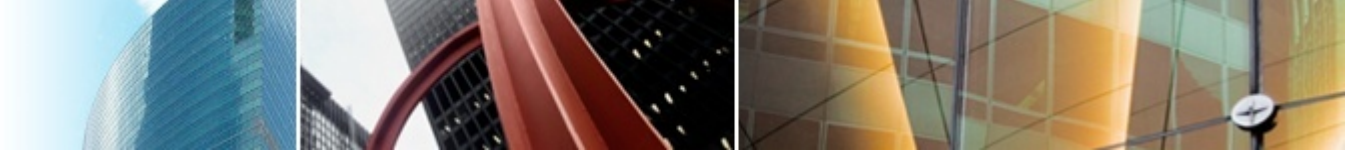
В теоретических трудах общие переменные называют *регистрами*



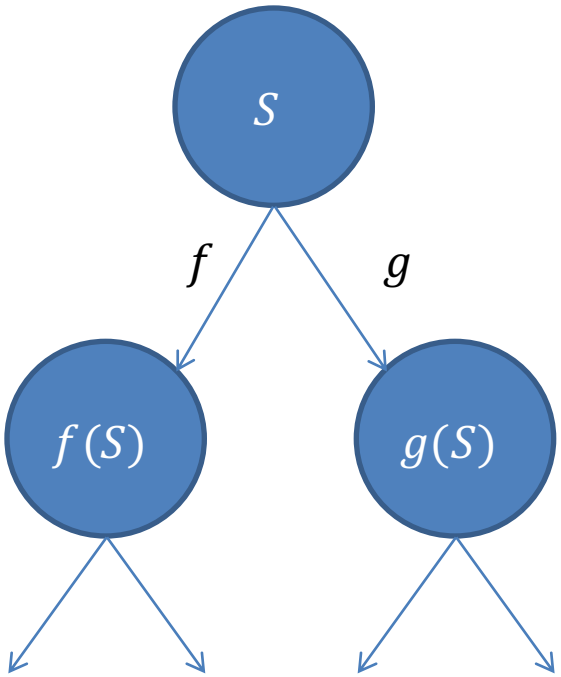
# Как моделировать многопоточное выполнение?

- **ПРИМЕР:** Очень простая программа. Всего два потока (P и Q), каждый из которых последовательно выполняет 2 действия и останавливается
  - У них есть общие переменные x и y (в начале равные 0)
  - Какие есть варианты r1 и r2 после исполнения?





# Моделирование исполнений через чередование операций



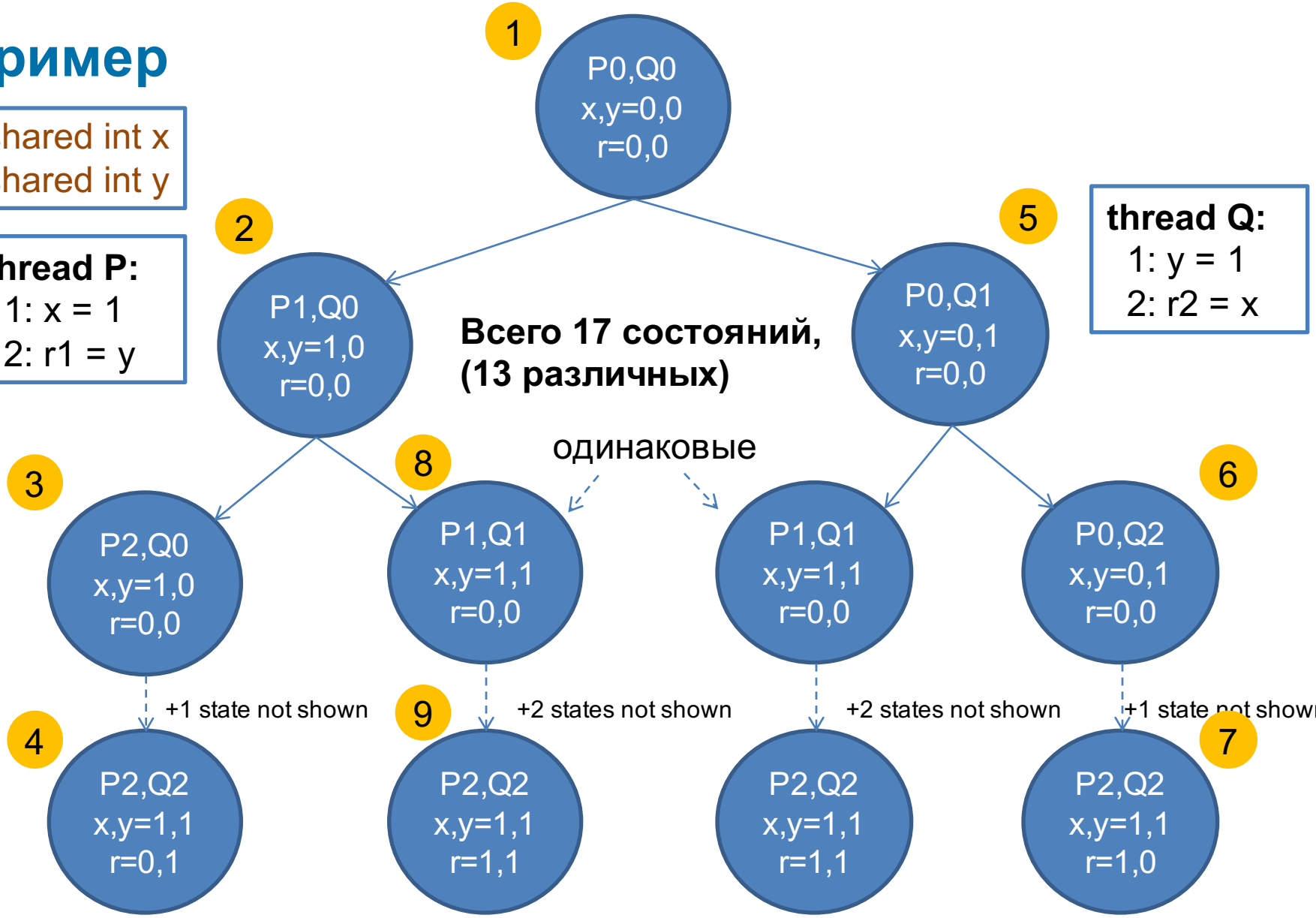

- $S$ это **общее состояние**:
  - Состояние всех потоков
  - И состояние всех общих объектов
- $f$ и $g$ это **операции** над объектами
  - Для переменных это операции чтения (вместе с результатом) и записи (вместе с значением)
  - Количество активных операций равно количеству потоков
- $f(S)$ это новое состояние после применения операции $f$ в состоянии $S$

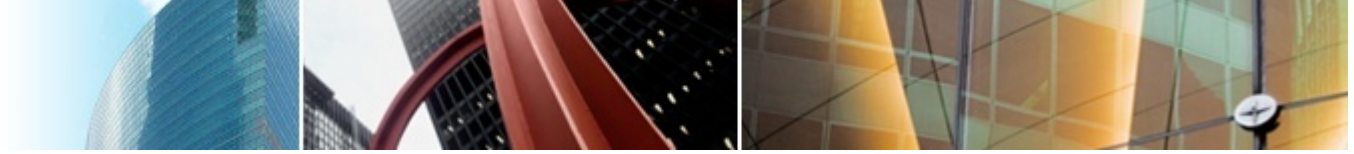
# Пример

shared int x
shared int y

**thread P:**
1: x = 1
2: r1 = y

**thread Q:**
1: y = 1
2: r2 = x





# Попробуем на практике (1)

```
@JCStressTest
@State
public class SimpleTest1 {
    int x;
    int y;

    @Actor
    public void threadP(IntResult2 r) {
        x = 1;
        r.r1 = y;
    }

    @Actor
    public void threadQ(IntResult2 r) {
        y = 1;
        r.r2 = x;
    }
}
```

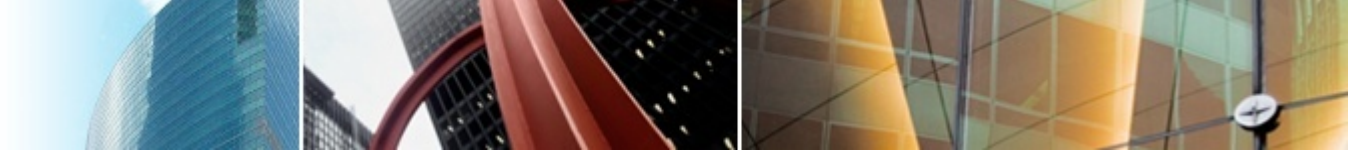
- Какие пары [r1, r2] будут наблюдаться?

```
Observed state     Occurrences
        [0, 0] (    1a059a702)
        [1, 1] (           29)
        [0, 1] (    7a473a762)
        [1, 0] (   10a625a627)
```

- Как же так? Ведь пары [0, 0] вообще не должно было быть!


"Very simple was my explanation,
and plausible enough –
as most wrong theories are!"

H.G. Wells, The Time Machine

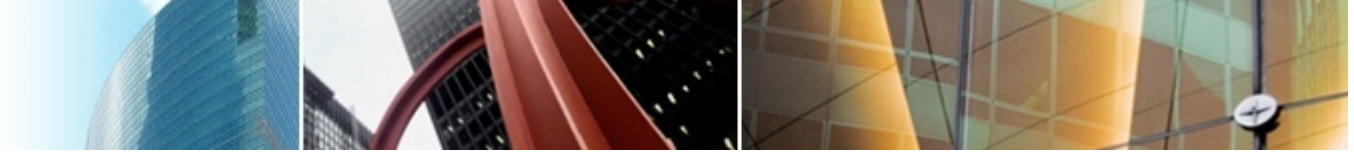


# Практическое объяснение

- Запись в память (x = 1 и y = 1) на современных процессорах (включая x86) не сразу идет в память, а попадает в очередь на запись
  - Другой поток не увидит эту запись сразу, ибо запись реально в память произойдет много позже.
  - Это как будто в программе произошла перестановка записи после чтения:
  - Такую перестановку мог сделать и компилятор (но здесь – не делал)

| **thread P:** | **thread Q:** |
|---|---|
| 1: r1 = y | 1: r2 = x |
| 2: x = 1 | 2: y = 1 |

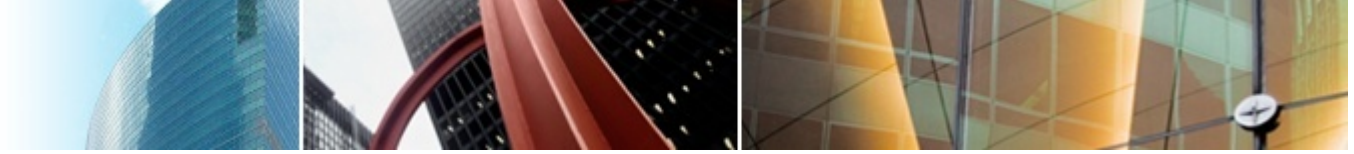
- Значит ли это, что нужно помнить о такой возможности? А что насчет других оптимизаций, которые делают процессоры и компиляторы? Как их учесть? Значит ли это что невозможно написать код одинаково работающий на разных архитектурах?



# Философия модели чередования

- Модель чередования не «параллельна»
  - Все операции в ней происходят последовательно (только порядок заранее не задан)
- А на самом деле на реальных процессорах операции чтения и записи памяти **не мгновенные**. Они происходят **параллельно** как в разных ядрах так и внутри одного ядра
  - И вообще процессор обменивается с памятью сообщениями чтения/записи и таких сообщений находящихся в обработке одновременно может быть очень много (!)

**Модель чередования не отражает фактическую реальность.** Когда же её можно использовать?

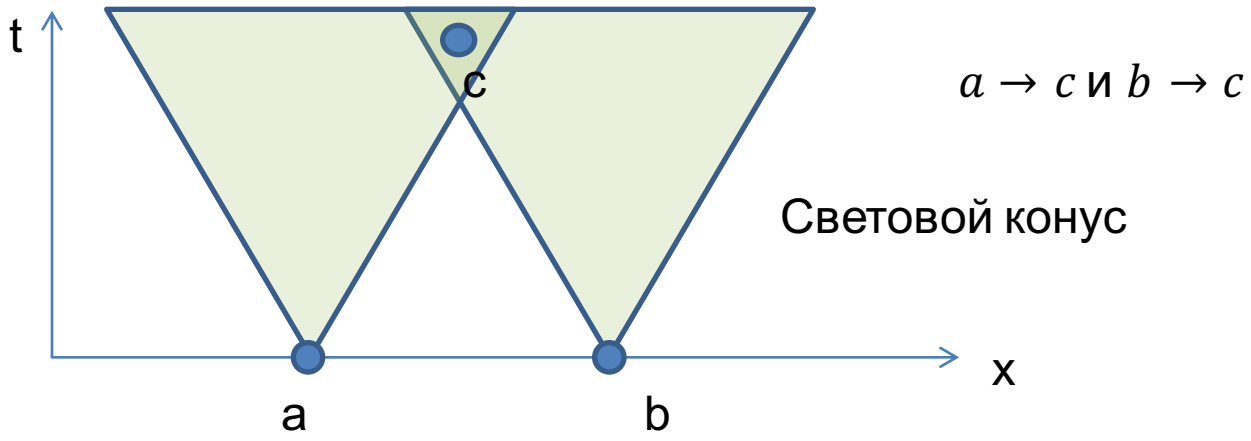


# Физическая реальность (1)

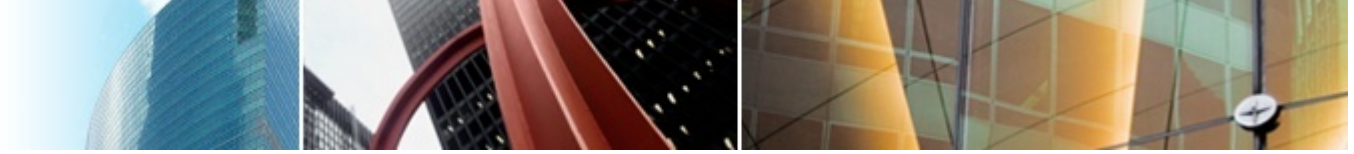
- Свет (электромагнитные волны) в вакууме распространяется со скоростью ~ $3 \cdot 10^8$ м/с
  - Это максимальный физический предел скорости распространения света. В реальных материалах – медленней.
- За один такт процессора с частотой 3 ГГц ($3 \cdot 10^9$ Гц) свет в вакууме проходит всего **10 см**.
  - **Соседние процессоры на плате физически не могут синхронизировать свою работу и физически не могут определить порядок происходящих в них событиях.**
  - Они работают действительно *параллельно*.

# Физическая реальность (2)

- Пусть $a, b, c \in E$ – это физически атомарные (неделимые) события, происходящие в пространстве-времени
  - Говорим «$a$ предшествует $b$» или «$a$ произошло до $b$» (и записываем $a \to b$), если свет от точки пространства-времени $a$ успевает дойти до точки пространства-времени $b$
  - Это отношение частичного порядка на событиях




Между $a$ и $b$ нет предшествования. Они происходят **параллельно**



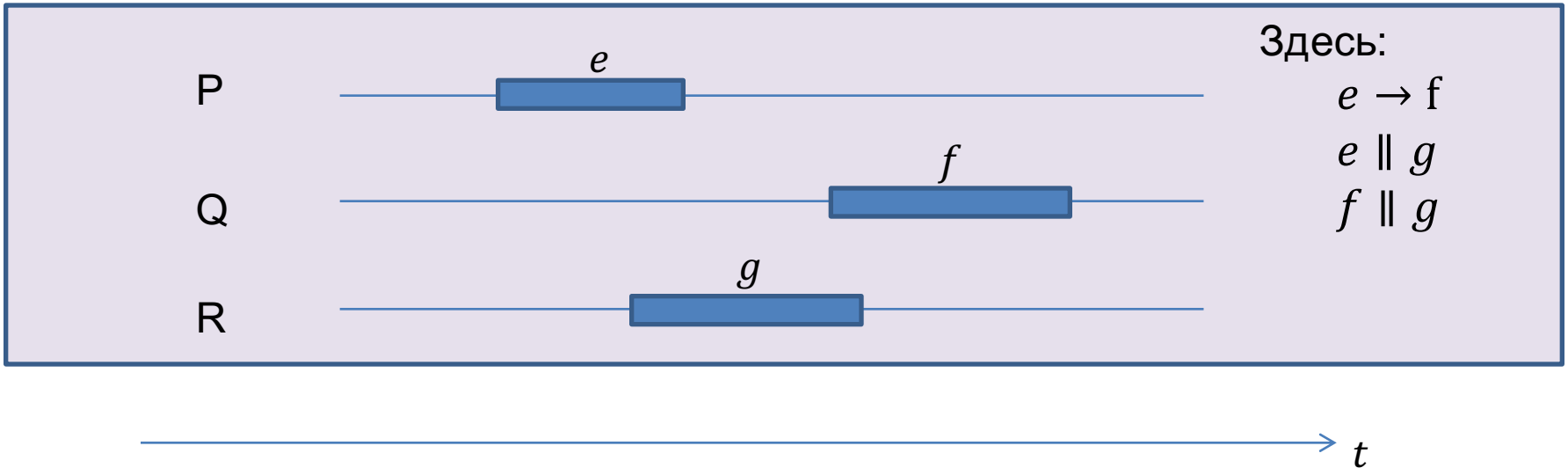
# Модель «произошло до»

- Впервые введена Л. Лампортом в 1978 году
- **Исполнение** системы – это пара $(H, \rightarrow_H)$
  - $H$ – это множество **операций** $e, f, g, \ldots$ (чтение и запись ячеек памяти), произошедших во время исполнения
  - «$\rightarrow_H$» – это транзитивное, антирефлексивное, асимметричное отношение (частичный строгий порядок) на множестве операций
  - $e \rightarrow_H f$ означает что "$e$ **произошло до** $f$ в исполнении $H$"
    - Чаще всего исполнение $H$ понятно из контекста и опускается
- Две операции $e$ и $f$ **параллельны** ($e \parallel f$) если $e \nrightarrow f \wedge f \nrightarrow e$
- **Система** – это набор всех возможных исполнений системы
  - Говорим, что «система имеет свойство $P$», если каждое исполнение системы имеет свойство $P$

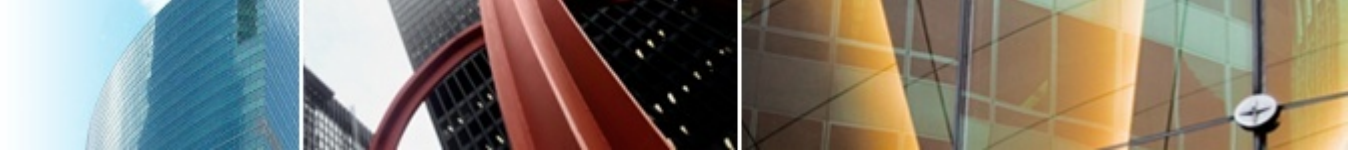
# Модель глобального времени

- В этой модели каждая операция – это временной интервал $e = [t_{inv}(e), t_{resp}(e)]$ где $t_{inv}(e), t_{resp}(e) \in \mathbb{R}$ и

$$e \to f \stackrel{\text{def}}{=} t_{resp}(e) < t_{inv}(f)$$



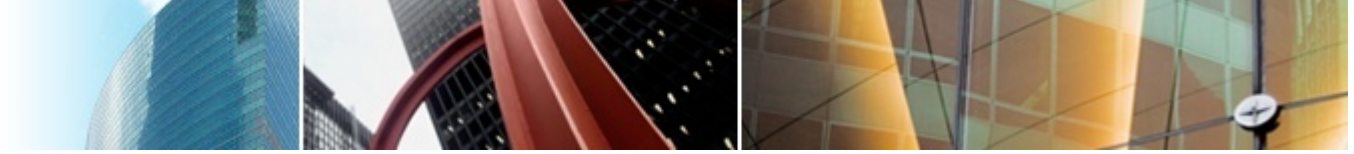

Будем использовать эту модель во всех иллюстрациях



# Обсуждение глобального времени

**На самом деле, никакого глобального времени нет и не может быть из-за физических ограничений**

- Это всего лишь механизм, позволяющий визуализировать факт существования **параллельных** операций
- При доказательстве различных фактов и анализе свойств [исполнений] системы время не используется
  - Анализируются только операции и отношения «произошло до»



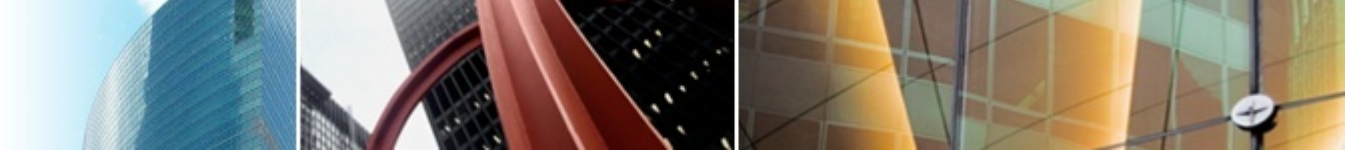
# «произошло до» на практике

- Современные языки программирования предоставляют программисту **операции синхронизации**:
  - Специальные механизмы чтения и записи переменных (**std::atomic** в C++11 и **volatile** в Java 5).
  - Создание потоков и ожидание их завершения
  - Различные другие библиотечные примитивы для **синхронизации**
- **Модель памяти** языка программирования определяет то, каким образом исполнение операций синхронизации создает отношение «произошло до»
  - Без них, разные потоки выполняются параллельно
  - Можно доказать те или иные свойства многопоточного кода, используя гарантии на возможные исполнения, которые дает модель памяти

DEVEXPERTS

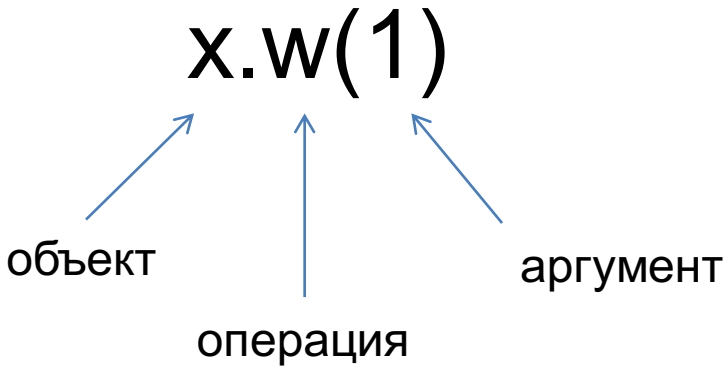
# Свойства исполнений над общими объектами

http://blog.lib.umn.edu/cspg/electionacademy/images/sharing.jpg

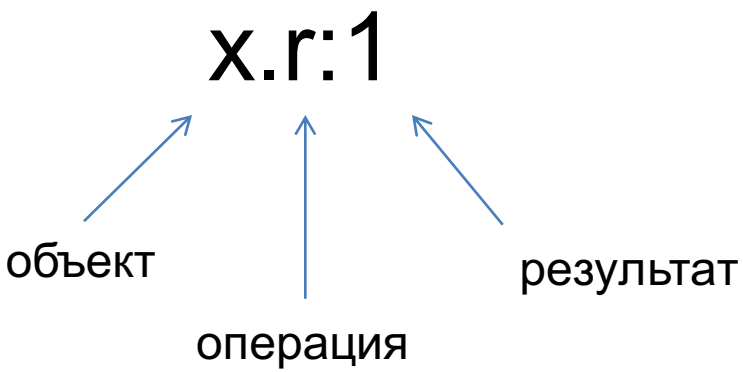


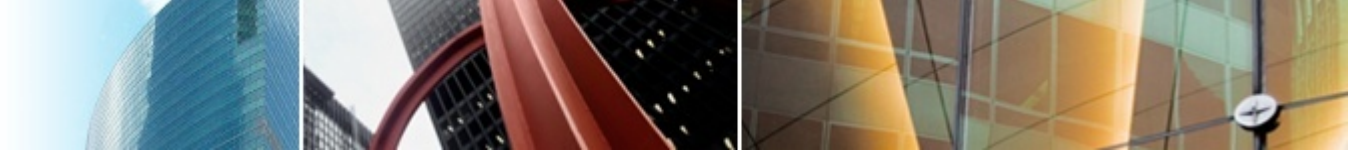
# Операции над общими объектами

**Запись (write)**
**общей переменной**



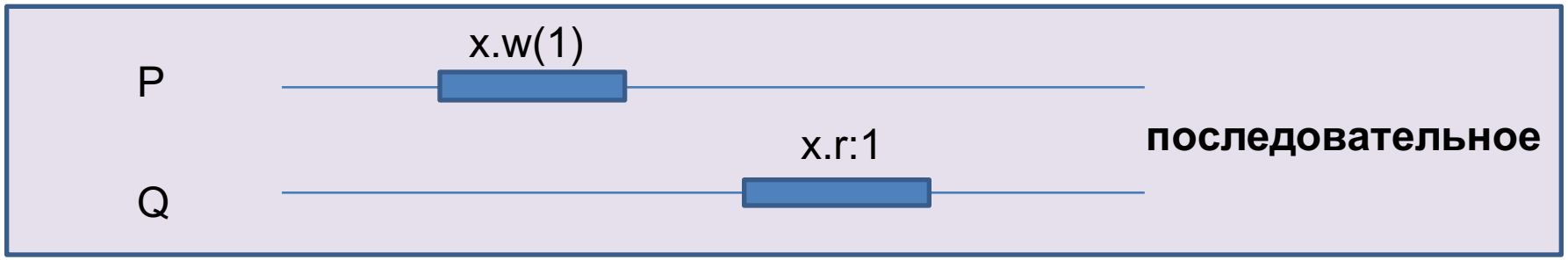

**Чтение (read)**
**общей переменной**

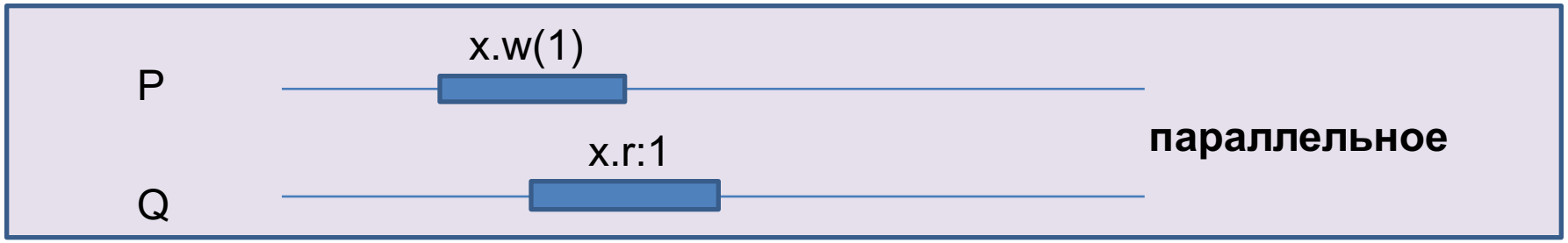




# Последовательное исполнение

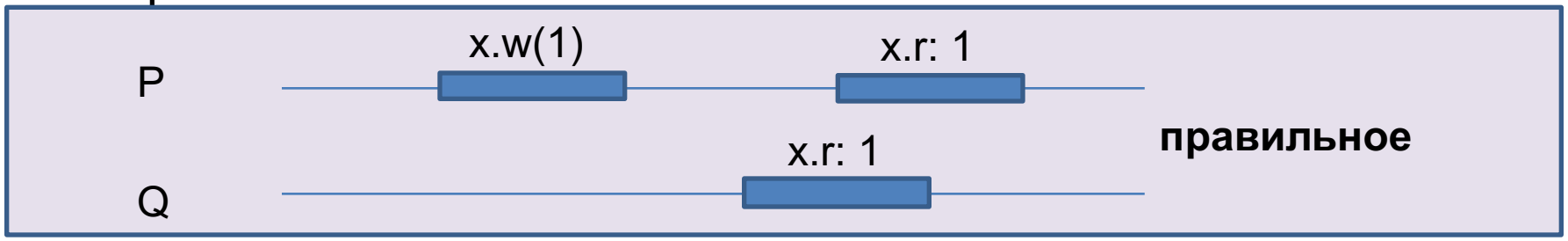
- Исполнение системы называется **последовательным**, если все **операции** линейно-упорядочены отношением «произошло до», то есть $\forall e, f \in H : e = f \vee e \to f \vee f \to e$



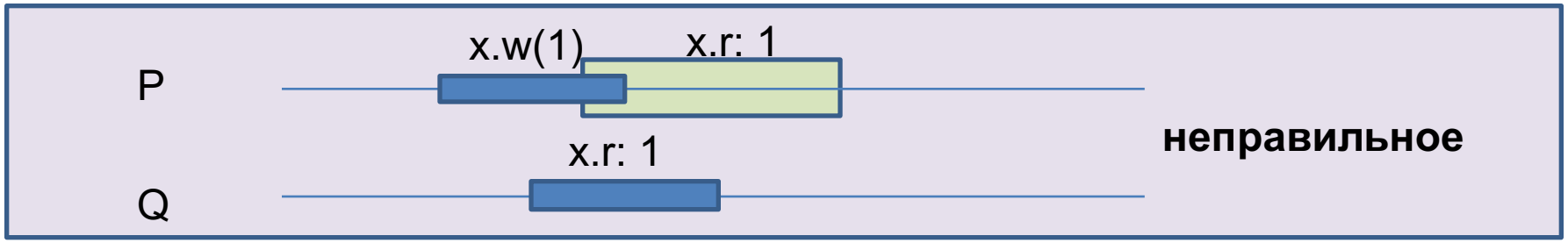



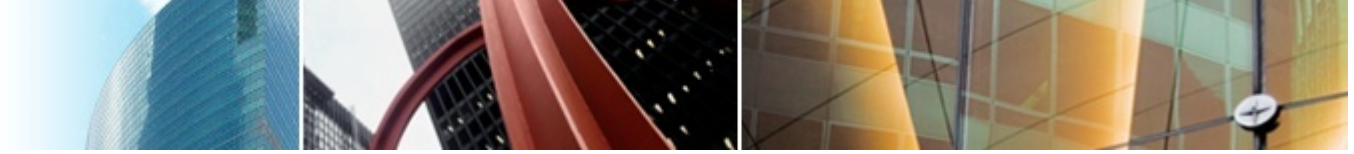
# Правильное исполнение

- $H|_P$ – сужение исполнения на поток $P$, то есть исполнение, где остались только операции происходящие в потоке $P$
- Исполнение называется **правильным (well-formed)**, если его сужение на каждый поток $P$ является последовательным историей



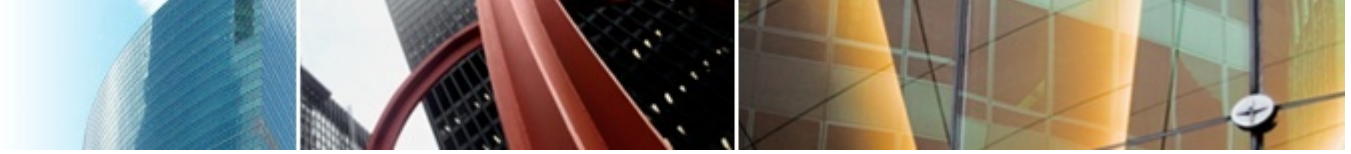

Нас интересуют только правильные исполнения





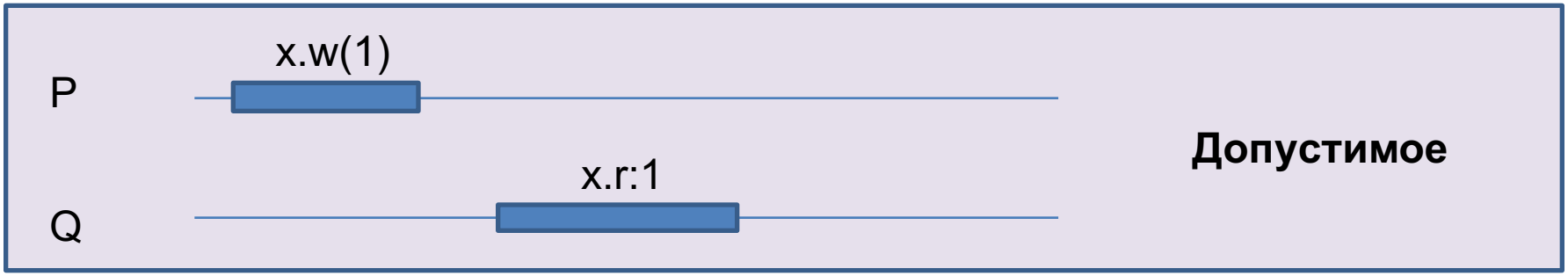
# Последовательная спецификация объекта

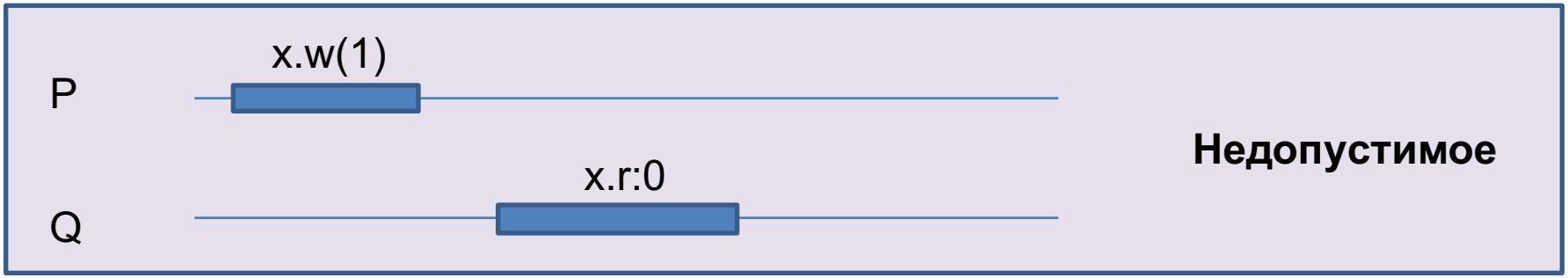
- $H|_x$ – сужение исполнения на объект $x$, то есть исполнение, где остались только операции происходящие над объектом $x$
- Если сужение исполнения на объект является последовательным, то можно проверить его на соответствие **последовательной спецификации объекта**
  - То, что мы привыкли видеть в обычном мире последовательного программирования
  - Например:
    - Чтение переменной должно вернуть последнее записанное значение.
    - Операции deque на объекте FIFO очереди должны возвращать значения в том порядке, в котором они передавались в операцию enqueue.

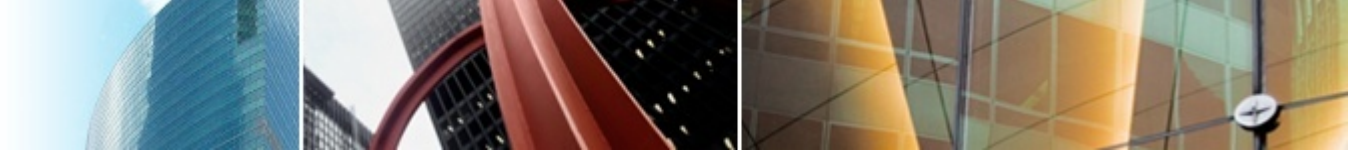


# Допустимое последовательное исполнение

- Последовательное исполнение является **допустимым (legal)**, если выполнены последовательные спецификации всех объектов





# Условия согласованности (корректности)

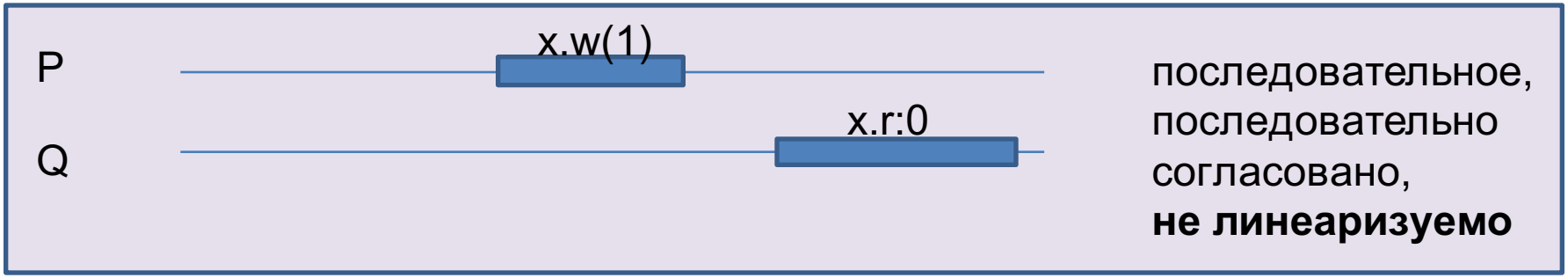
- Как определить допустимость **параллельного** исполнения?
  - Сопоставив ему **эквивалентное** (состоящее из тех же событий и операций) **допустимое последовательное исполнение**
  - Как именно – тут есть варианты: **условия согласованности**
- Аналог «корректности» в мире параллельного программирования

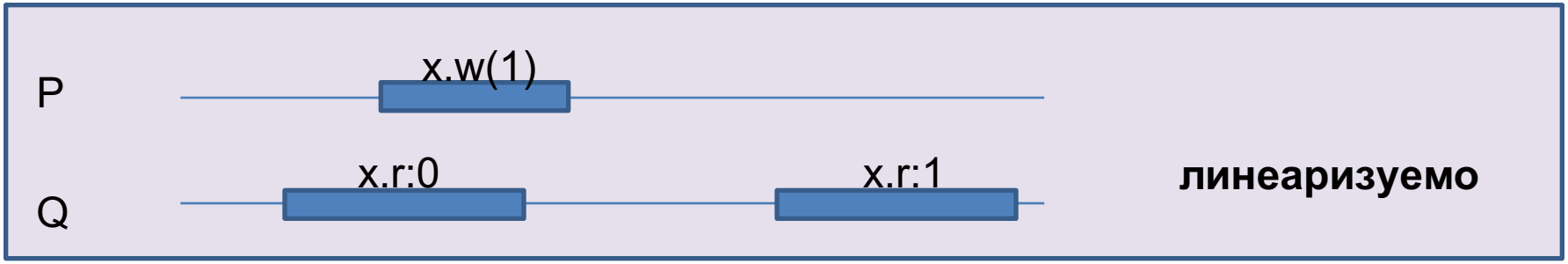
**Базовое требование: корректные последовательные программы должны работать корректно в одном потоке**

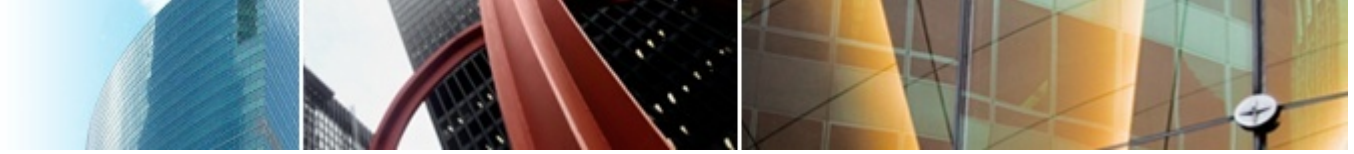
- Условий согласованности много, из них основные (удовлетворяющие базовому требованию) это:
  - **Последовательная согласованность**
  - **Линеаризуемость**

# Линеаризуемость

- Исполнение **линеаризуемо** если можно сопоставить эквивалентное ему допустимое последовательное исполнение, которое сохраняет порядок «**произошло до**»

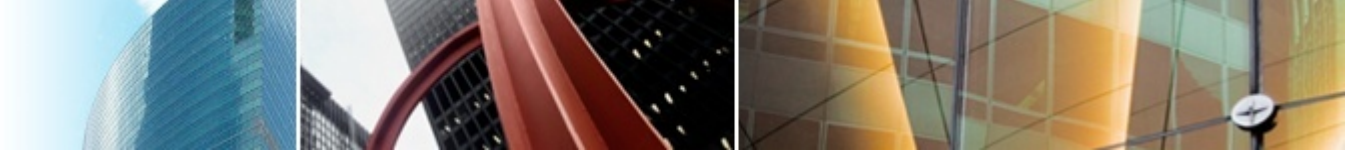




# Свойства линеаризуемости

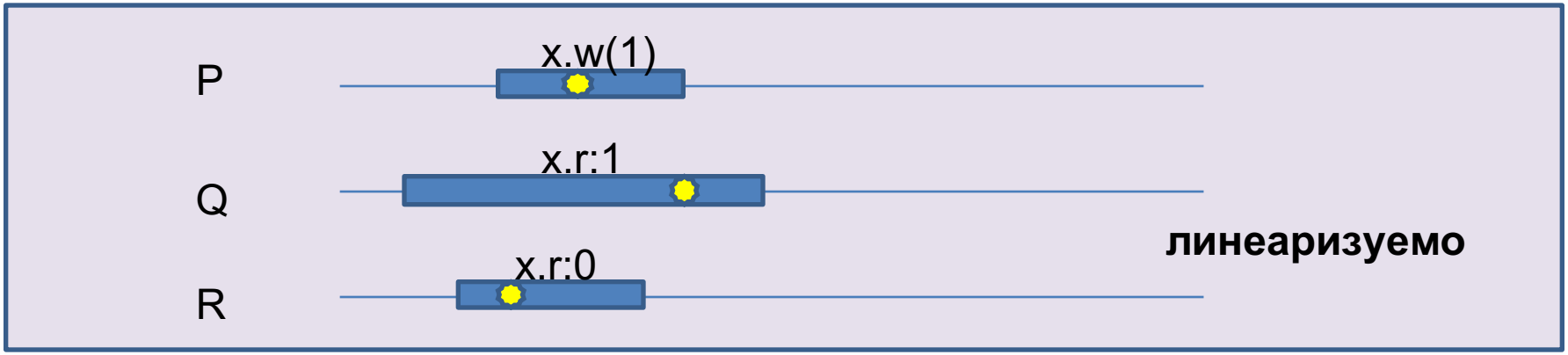
- В линеаризуемом исполнении каждой **операции** $e$ можно сопоставить некое глобальное время (**точку линеаризации**) $t(e) \in \mathbb{R}$ так, что время разных операций различно и
$$e \to f \Rightarrow t(e) < t(f)$$
  - Это эквивалентное определение линеаризуемости
- Линеаризуемость **локальна**. Линеаризуемость исполнения на каждом объекте эквивалентна линеаризуемости исполнения системы целиком
- Операции над **линеаризуемыми объектами** называют **атомарными.**
  - Они происходят *как бы* мгновенно и в определенном порядке, *как бы* в каком-то глобальном времени

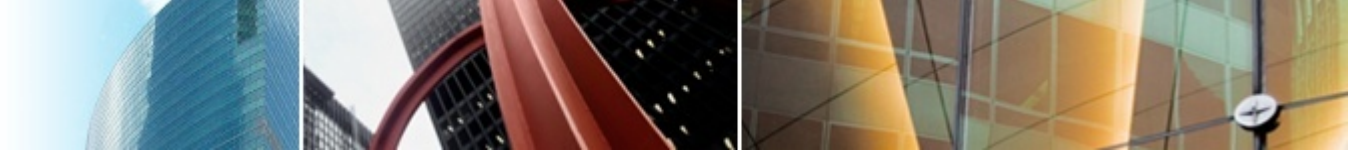


# Линеаризуемость в глобальном времени

- В глобальном времени исполнение линеаризуемо тогда и только тогда, когда точки линеаризации могут быть выбраны так, что

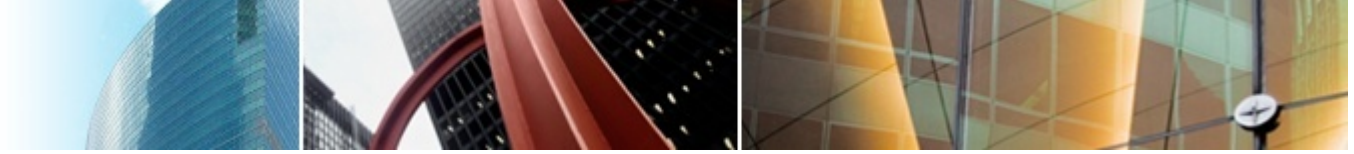
$$\forall e: t_{inv}(e) < t(e) < t_{resp}(e)$$





# Линеаризуемость и чередование

Исполнение системы, выполняющей операции над **линеаризуемыми** (**атомарными**) объектами, можно анализировать в **модели чередования**

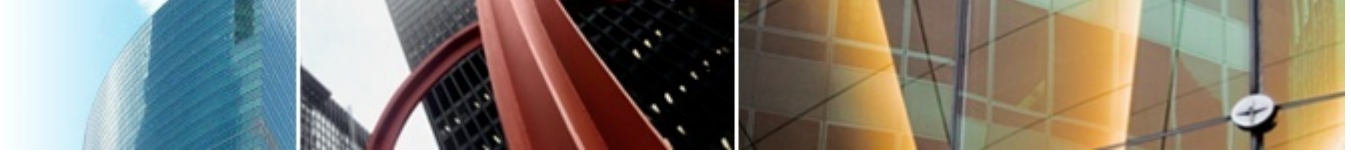


# Иерархия линеаризуемых объектов

- **Из более простых линеаризуемых объектов можно делать линеаризуемые объекты более высокого уровня**

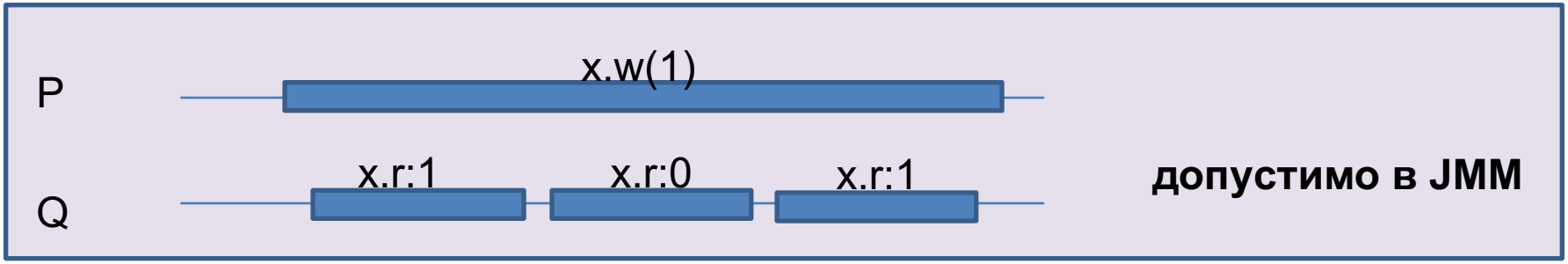
Доказав линеаризуемость сложного объекта, можно **абстрагироваться от деталей реализации** в нем, считать операции над ним атомарными и строить объекты более высокого уровня

- Когда говорят, что какой-то объект безопасен для использования из нескольких потоков **(thread-safe)**, то по умолчанию имеют в виду **линеаризуемость** операций над ним



# Применительно к Java

- Все операции над **volatile** полями в Java, согласно JMM, являются **операциями синхронизации** (17.4.2), которые всегда линейно-упорядочены в любом исполнении (17.4.4) и **согласованы** с точки зрения чтения/записи (17.4.7)
  - А значит являются **линеаризуемыми**
- Но операции над **не volatile** полями воистину могут нарушать не только линеаризуемость, но и последовательную согласованность (в отсутствии синхронизации)



# Попробуем на практике (2)

```
@JCStressTest
@State
public class SimpleTest2 {
    volatile int x;
    volatile int y;

    @Actor
    public void threadP(IntResult2 r) {
        x = 1;
        r.r1 = y;
    }

    @Actor
    public void threadQ(IntResult2 r) {
        y = 1;
        r.r2 = x;
    }
}
```

- Какие пары [r1, r2] будут наблюдаться?

```
Observed state     Occurrences
        [1, 1] (       58a197)
        [0, 1] (    3a464a591)
        [1, 0] (    3a528a262)
```

- Что мы и ожидали исходя из модели чередования операций!


# Но как это получается на x86 процессоре?

- SimpleTest1 (без **volatile**)

**thread P:**

```
mov     DWORD PTR [rdi+0xc],0x1     ;*putfield x
mov     r10d,DWORD PTR [rdi+0x10]   ;*getfield y
mov     DWORD PTR [r9+0x8c],r10d    ;*putfield r1
```

**thread Q:**

```
mov     DWORD PTR [rsi+0x10],0x1    ;*putfield y
mov     r10d,DWORD PTR [rsi+0xc]    ;*getfield x
mov     DWORD PTR [rcx+0x110],r10d  ;*putfield r2
```

- SimpleTest2 (с **volatile int** x, y)

**thread P:**

```
mov      DWORD PTR [r11+0xc],0x1    ;*putfield x
lock add DWORD PTR [rsp],0x0        ; !FENCE!
mov      r10d,DWORD PTR [r11+0x10]  ;*getfield y
mov      DWORD PTR [r9+0x8c],r10d   ;*putfield r1
```

**thread Q:**

```
mov      DWORD PTR [r11+0x10],0x1   ;*putfield y
lock add DWORD PTR [rsp],0x0        ; !FRENCE!
mov      r10d,DWORD PTR [r11+0xc]   ;*getfield x
mov      DWORD PTR [r9+0x110],r10d  ;*putfield r2
```

- Появился дополнительный барьер (memory fence)

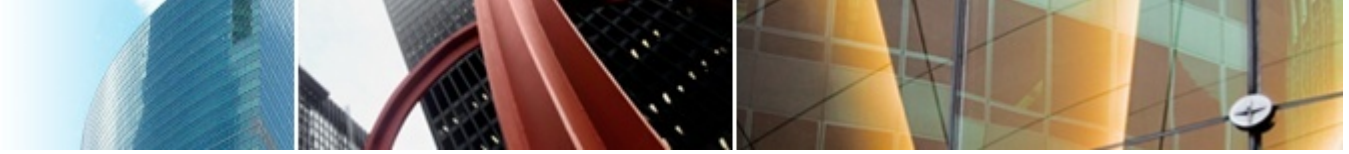

# Эффект на производительность?

```
@State(Scope.Group)
public class SimpleTest1 {
    int x;
    int y;

    @Benchmark
    @Group
    public void threadP(IntResult2 r) {
        x = 1;
        r.r1 = y;
    }

    @Benchmark
    @Group
    public void threadQ(IntResult2 r) {
        y = 1;
        r.r2 = x;
    }
}
```

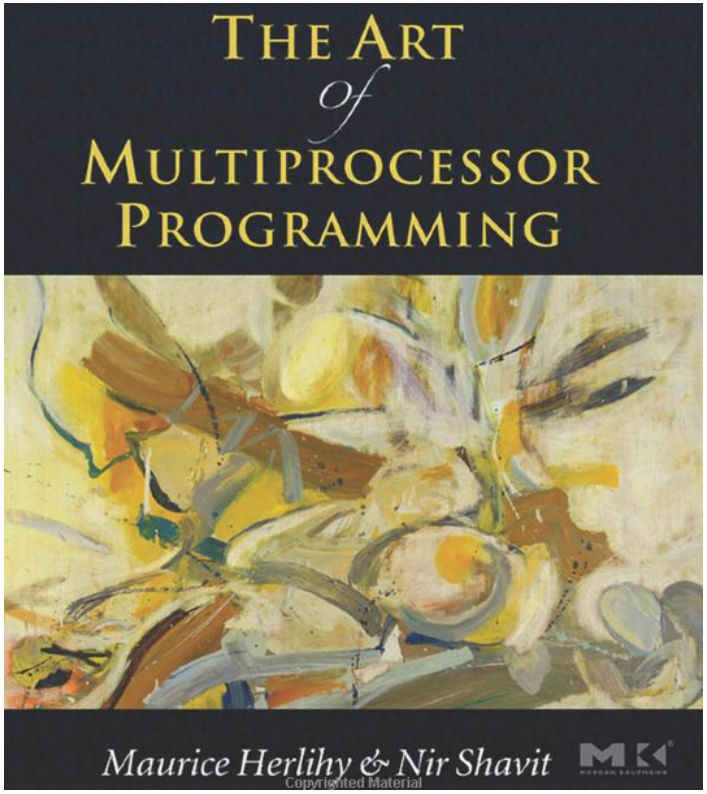
```
@State(Scope.Group)
public class SimpleTest2 {
    volatile int x;
    volatile int y;

    @Benchmark
    @Group
    public void threadP(IntResult2 r) {
        x = 1;
        r.r1 = y;
    }

    @Benchmark
    @Group
    public void threadQ(IntResult2 r) {
        y = 1;
        r.r2 = x;
    }
}
```

| | | |
|---|---|---|
| SimpleTest1.group | thrpt | 321 505 310,804 ops/s |
| SimpleTest2.group | thrpt | 54 770 289,324 ops/s |

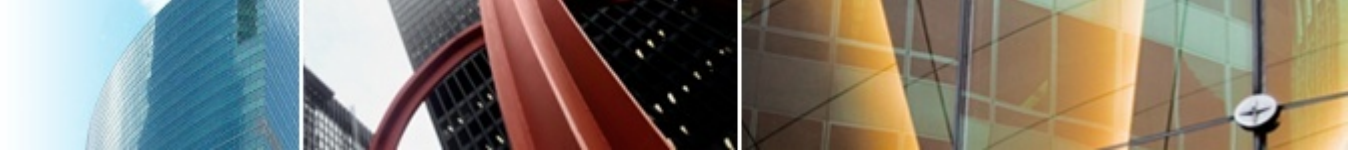
DEVEXPERTS



# Рекомендованная литература




В моем блоге **elizarov.livejournal.com** есть ссылки на научные работы для дополнительного чтения по теме



# Спасибо за внимание

Вопросы?

Слайды также выложены на **elizarov.livejournal.com**
twitter at @relizarov