

Многопоточное Программирование: Определения и Формализм

Роман Елизаров, JetBrains, elizarov@gmail.com

Никита Коваль, JetBrains, ndkoval@ya.ru

ИТМО 2019



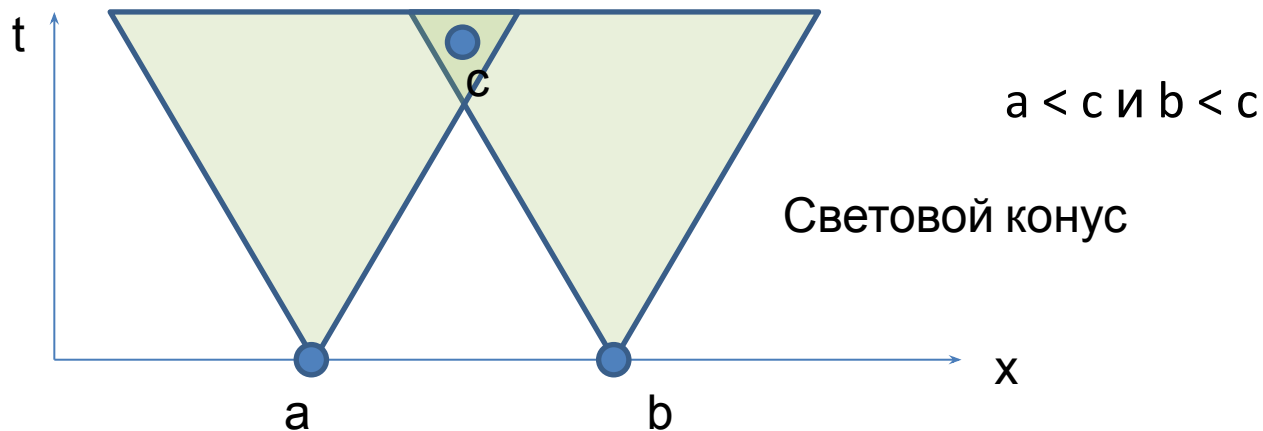
ITMO UNIVERSITY

Физическая реальность (1)

- Свет (электромагнитные волны) в вакууме распространяется со скоростью $\sim 3 \cdot 10^8$ 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 физически *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 » ($a < b$), 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» (happens before)

- Впервые введена Л. Лампортом в 1978 году
- **Исполнение** системы – это пара (H, \rightarrow_H)
 - H – это множество **операций** e, f, g, \dots (чтение и запись ячеек памяти и т.п.), произошедших во время исполнения
- **Произошло до**
 - На **операциях** определен *частичный* порядок $e \rightarrow_H f$
 - \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 $e \not\rightarrow f \wedge f \not\rightarrow e$

Модель «произошло до» (happens before)

- **Операция** (сложная) состоит из двух **событий** (простых):
 - $inv(e)$ – вызов операции
 - $res(e)$ – ответ на операцию (результат)
 - Все события **полностью** упорядочены отношением $<_H$
- Порядок **операций** определяется порядком **событий**

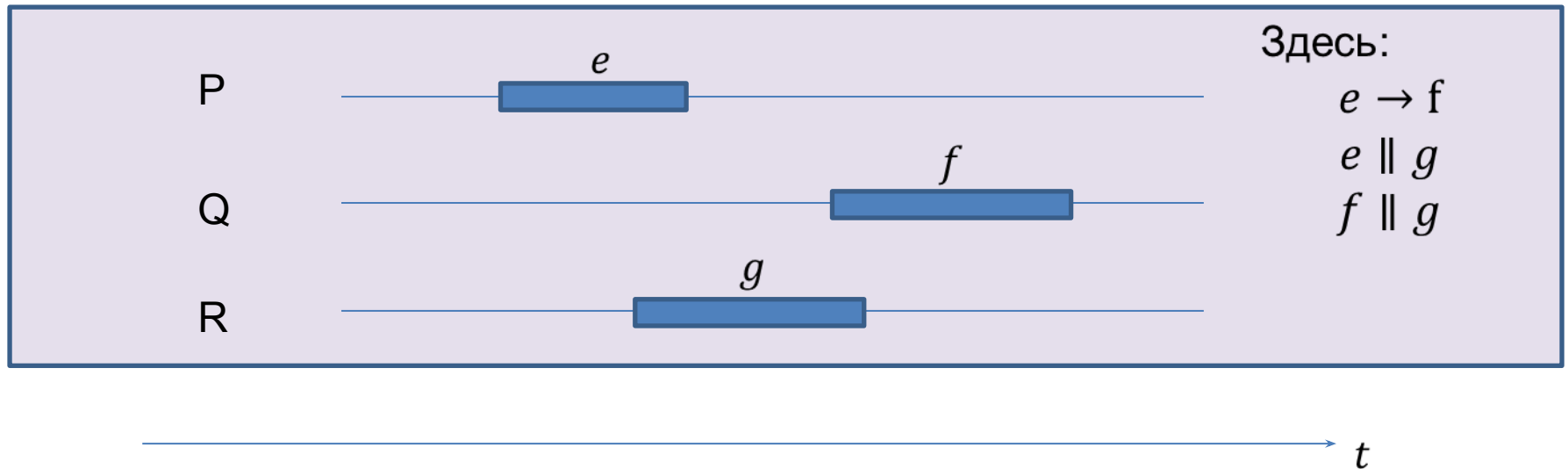
$e \rightarrow_H f$ по определению тогда и только тогда когда $res(e) <_H inv(f)$

- **Замечание 1:** отдельные **события** нам нужны редко
- **Замечание 2:** полная упорядоченность отдельных событий играет важную роль в теории многопоточного программирования с общей памятью

Глобальное время

- Располагаем $inv(e)$ и $res(e)$ на числовой оси
- Каждая **операция** это числовой интервал

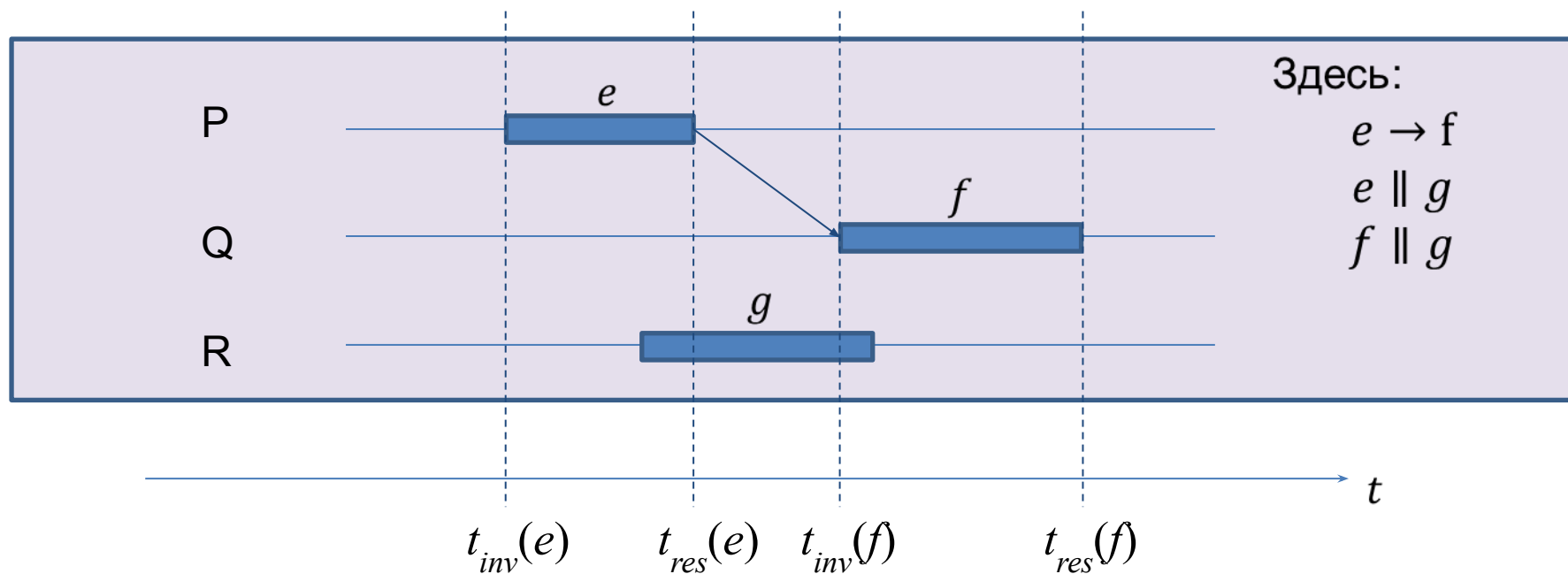
$$e \rightarrow f \stackrel{\text{def}}{=} t_{res}(e) < t_{inv}(f)$$



Глобальное время

- Располагаем $inv(e)$ и $res(e)$ на числовой оси
- Каждая **операция** это числовой интервал

$$e \rightarrow f \stackrel{\text{def}}{=} t_{res}(e) < t_{inv}(f)$$



Система

- **Система** – это набор всех возможных исполнений системы
- Говорим, что «система имеет свойство P », если каждое исполнение системы имеет свойство P

«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

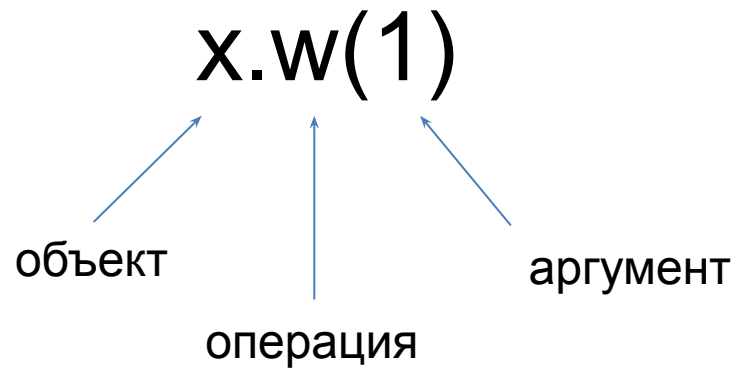
Подробности в отдельной лекции

Свойства исполнений над общими объектами

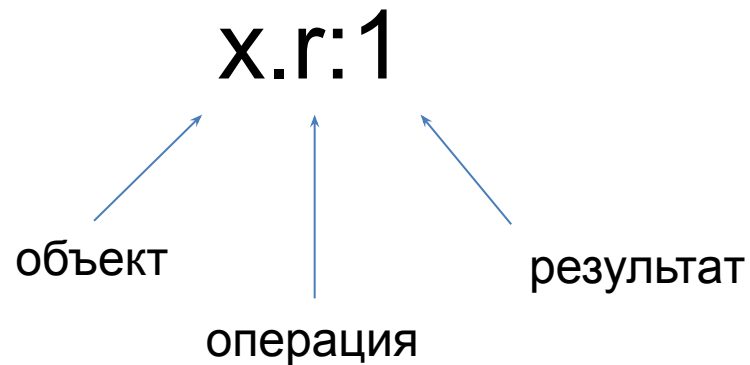


Операции над общими объектами

Запись (write)
общей переменной



Чтение (read)
общей переменной

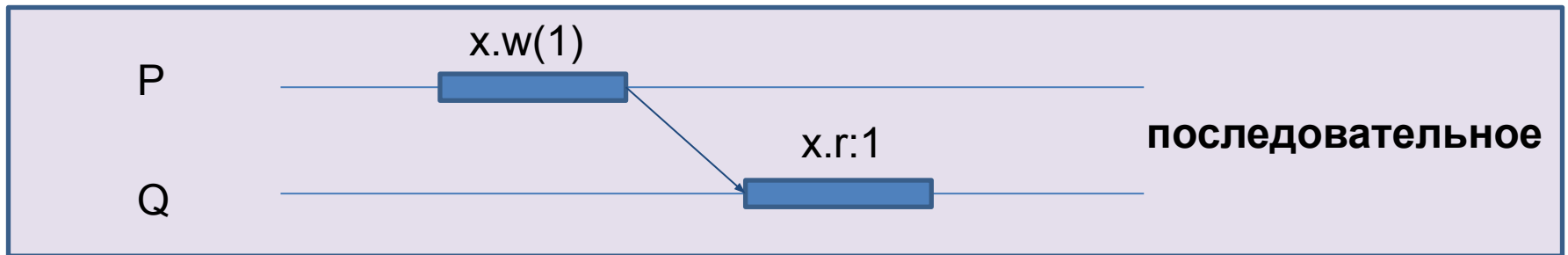


Последовательное исполнение

Исполнение системы называется **последовательным**, если все **операции** линейно-упорядочены отношением “произошло до”

$$\forall e, f \in H : (e = f) \vee (e \rightarrow f) \vee (f \rightarrow e)$$

Значит все события упорядочены $inv(e_1) < res(e_1) < inv(e_2) < \dots$

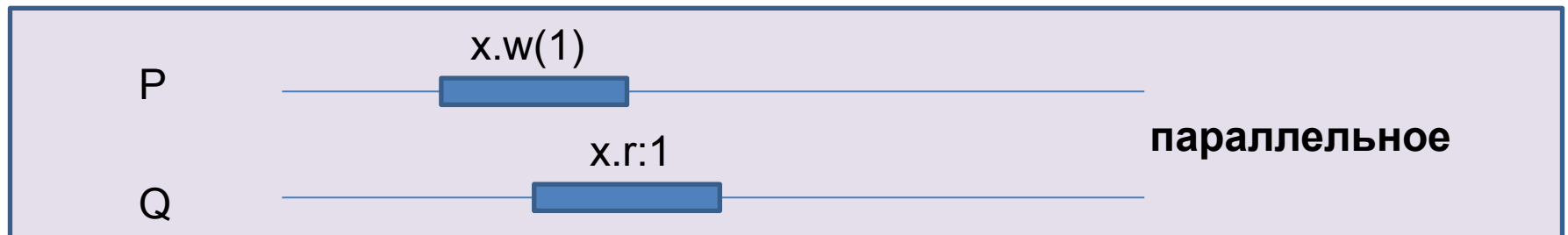
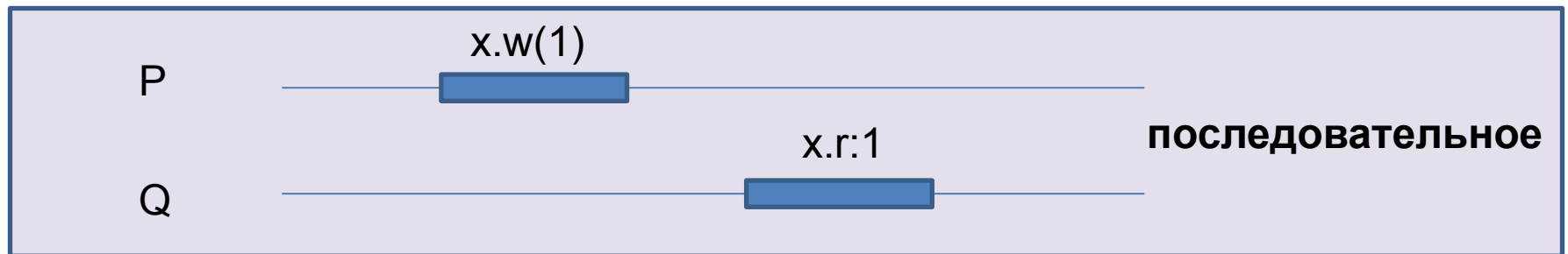


Последовательное исполнение

Исполнение системы называется **последовательным**, если все **операции** линейно-упорядочены отношением “произошло до”

$$\forall e, f \in H : (e = f) \vee (e \rightarrow f) \vee (f \rightarrow e)$$

Значит все события упорядочены $inv(e_1) < res(e_1) < inv(e_2) < \dots$



Конфликты и гонки данных (data race)

- Две операции над одной переменной одна из которых это **запись** называются **конфликтующими**
 - read-write или write-write

Конфликты и гонки данных (data race)

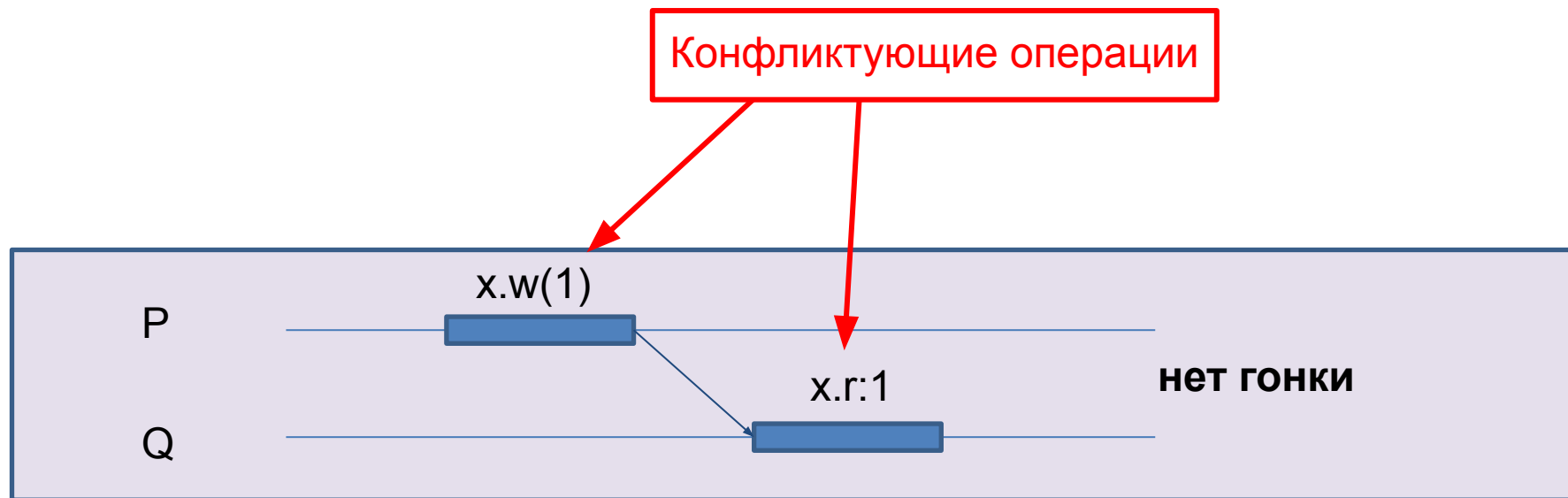
- Две операции над одной переменной одна из которых это **запись** называются **конфликтующими**
 - read-write или write-write

Конфликтующие операции не коммутируют
в модели чередования

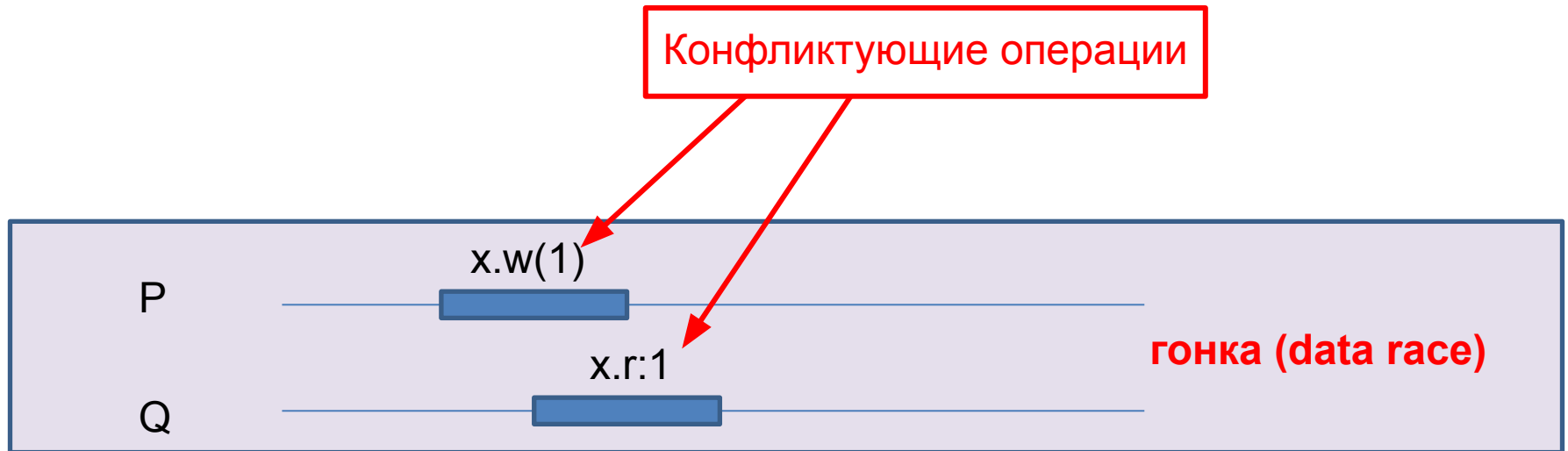
Конфликты и гонки данных (data race)

- Две операции над одной переменной одна из которых это **запись** называются **конфликтующими**
 - read-write или write-write
- Если две конфликтующие операции произошли **параллельно** (нет отношения произошло до) то такая ситуация называется **гонка данных (data race)**
 - Это свойство конкретного **исполнения**

Пример



Пример



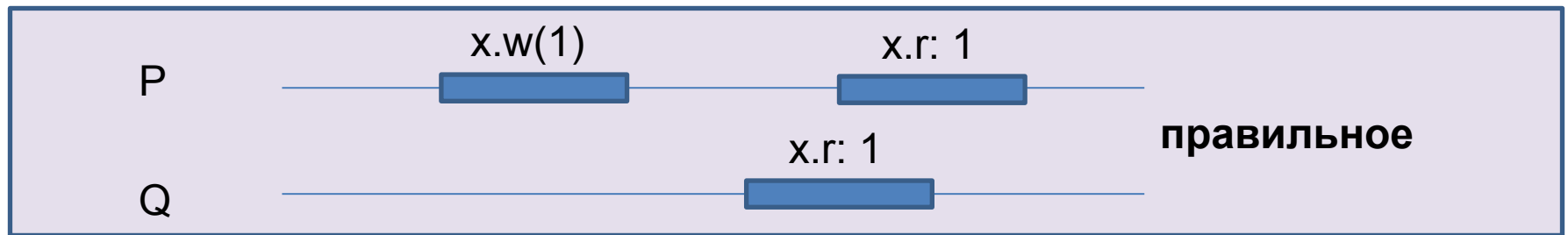
Конфликты и гонки данных (data race)

- Две операции над одной переменной одна из которых это **запись** называются **конфликтующими**
 - read-write или write-write
- Если две конфликтующие операции произошли **параллельно** (нет отношения произошло до) то такая ситуация называется **гонка данных (data race)**
 - Это свойство конкретного **исполнения**

Программа, в любом **допустимом** исполнении которой (с точки зрения модели памяти) нет **гонок данных**, называется **корректно синхронизированной**

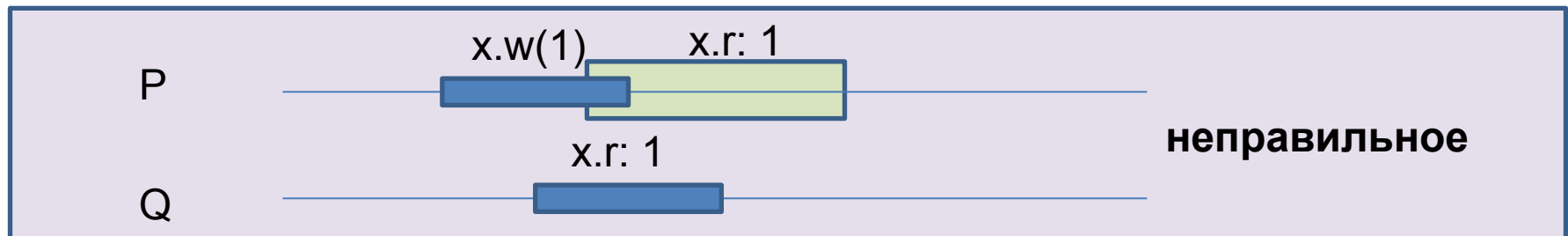
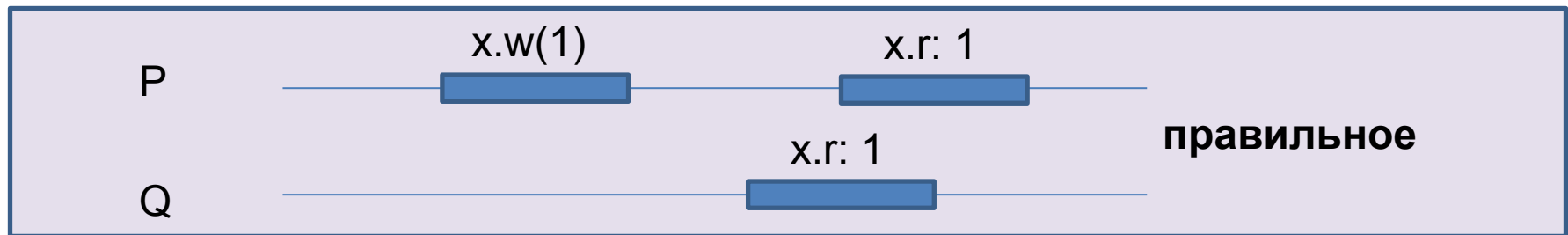
Правильное исполнение

- $H|_P$ — 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**



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**



Правильное исполнение и нотация

- $H|_P$ – сужение исполнения на поток P – это множество всех **операций** $e \in H$, таких что $\text{proc}(e) = P$
 - Исполнение называется **правильным (well-formed)**, если его сужение на любой поток P является **последовательным**
 - Задается программой которую выполняет поток
 - Объединение всех сужений на потоки называется **программным порядком** (po = program order)

Нас интересуют только правильные исполнения
Дальше работает только с такими

Правильное исполнение и нотация

- $H|_P$ – сужение исполнения на поток P – это множество всех **операций** $e \in H$, таких что $\text{proc}(e) = P$
 - Исполнение называется **правильным (well-formed)**, если его сужение на любой поток P является **последовательным**
 - Задается программой которую выполняет поток
 - Объединение всех сужений на потоки называется **программным порядком** (po = program order)
- $H|_x$ – сужение истории на объект x – это множество всех **операций** $e \in H$, таких что $\text{obj}(e) = x$

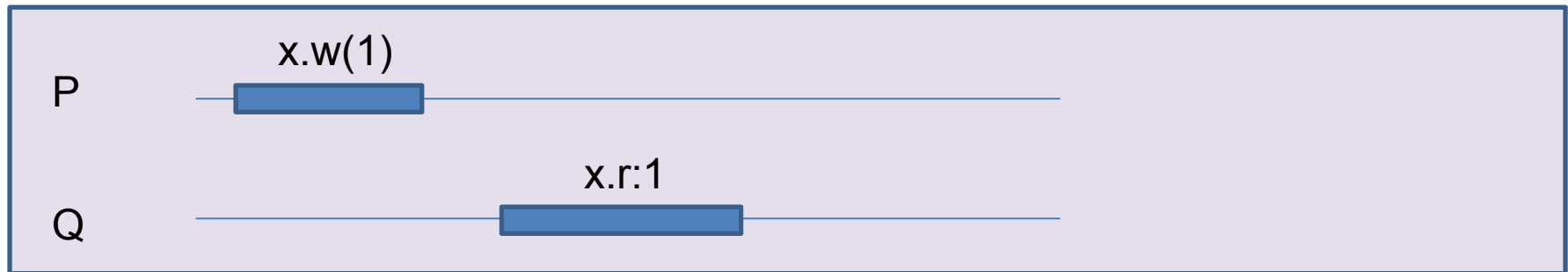
В правильном исполнении сужение на
объекты не обязательно является
последовательным(!)

Последовательная спецификация объекта

- Если сужение исполнения на объект $N|_x$ является **последовательным**, то можно проверить его на соответствие **последовательной спецификации объекта**

Последовательная спецификация объекта

- Если сужение исполнения на объект $N|_x$ является **последовательным**, то можно проверить его на соответствие **последовательной спецификации объекта**



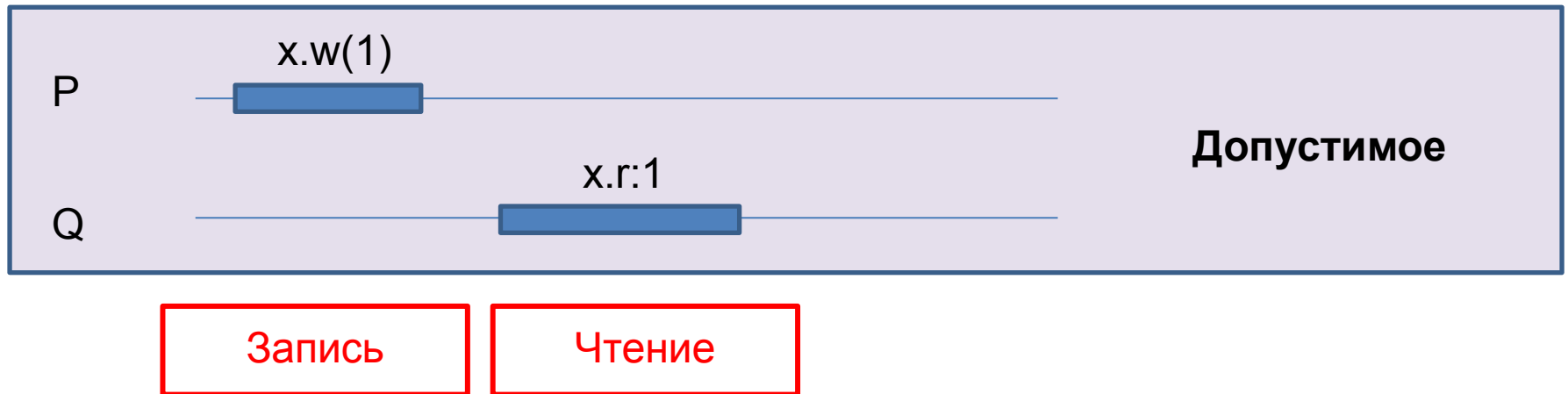
Запись

Чтение

Что мы привыкли видеть в обычном мире
последовательного программирования?

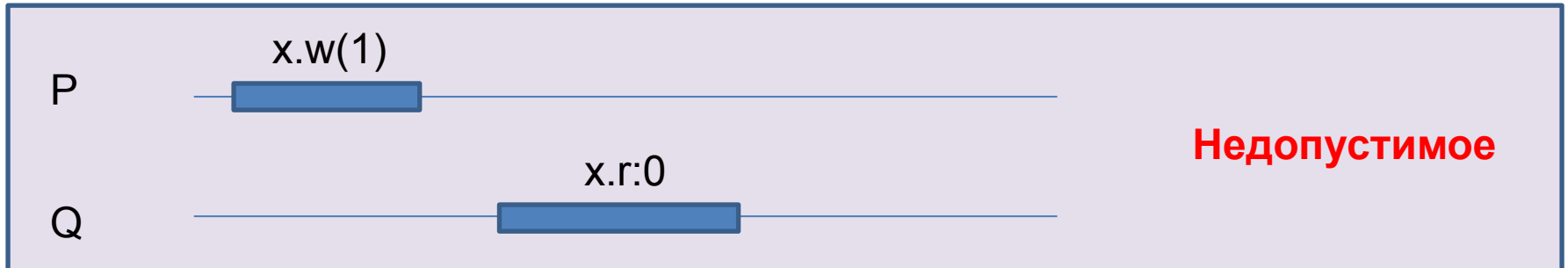
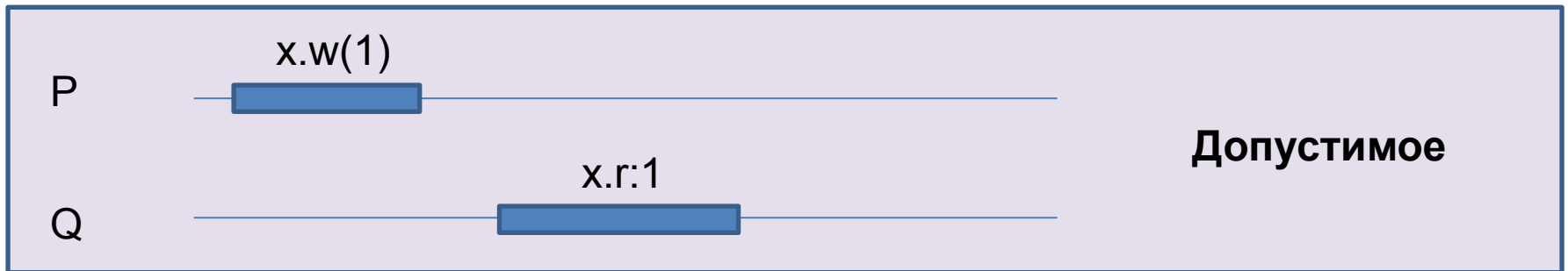
Допустимое последовательное исполнение

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



Допустимое последовательное исполнение

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

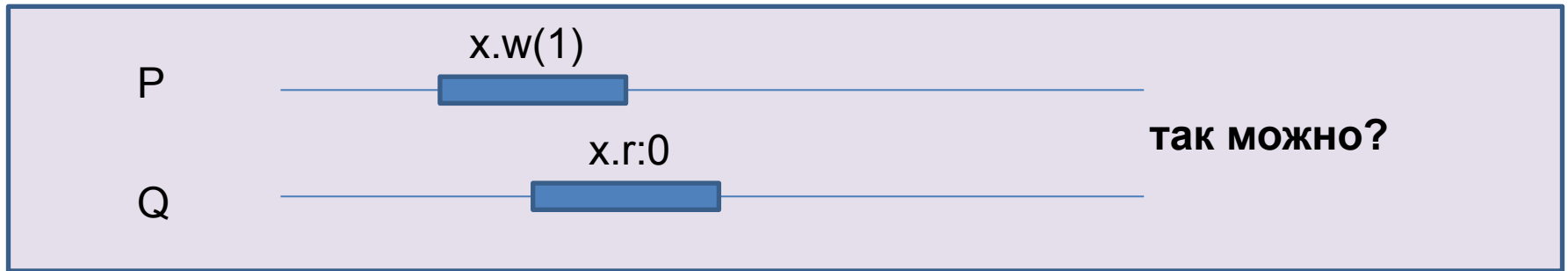


Условия согласованности (корректности)

- Как определить допустимость **параллельного** исполнения?

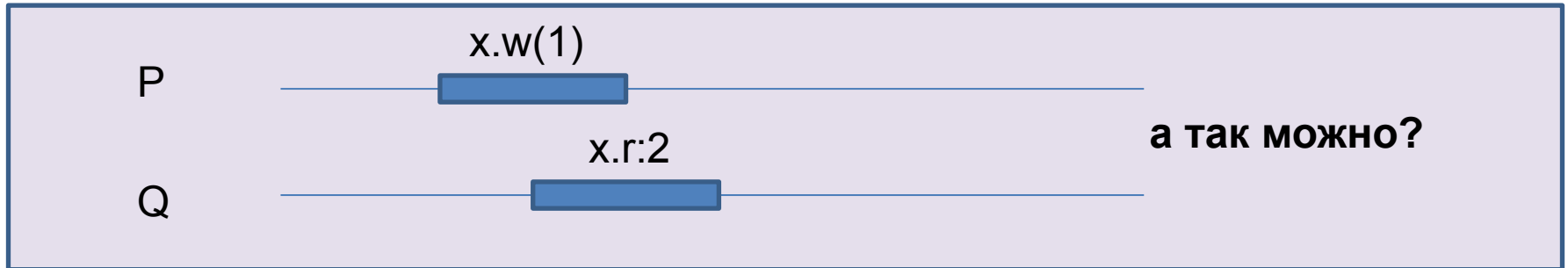
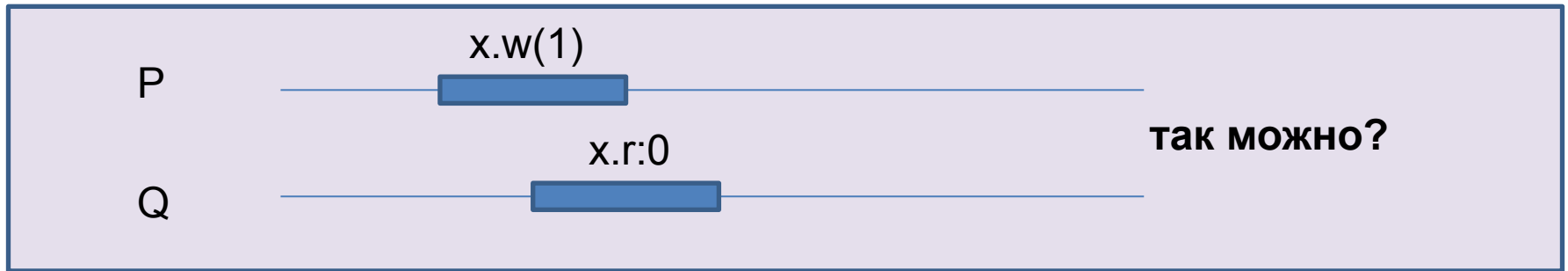
Условия согласованности (корректности)

- Как определить допустимость **параллельного** исполнения?



Условия согласованности (корректности)

- Как определить допустимость **параллельного** исполнения?



Условия согласованности (корректности)

- Как определить допустимость **параллельного** исполнения?
 - Сопоставим ему **эквивалентное** (состоящее из тех же событий и операций) **допустимое последовательное исполнение**
 - Как именно – тут есть варианты: **условия согласованности**

Условия согласованности (корректности)

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

Условия согласованности (корректности)

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

Базовое требование:
корректные последовательные
программы должны считаться
согласованными при любом их
исполнении в одном потоке

Условия согласованности (корректности)

- Как определить допустимость **параллельного** исполнения?
 - Сопоставим ему **эквивалентное** (состоящее из тех же событий и операций) **допустимое последовательное исполнение**
 - Как именно – тут есть варианты: **условия согласованности**
- Согласованность это аналог «корректности» в мире многопоточного программирования
- Условий согласованности много:
 - Согласованность при покое (quiescent consistency)
 - **Последовательная согласованность (sequential consistency)**
 - **Линеаризуемость (linearizability)**
 - и другие

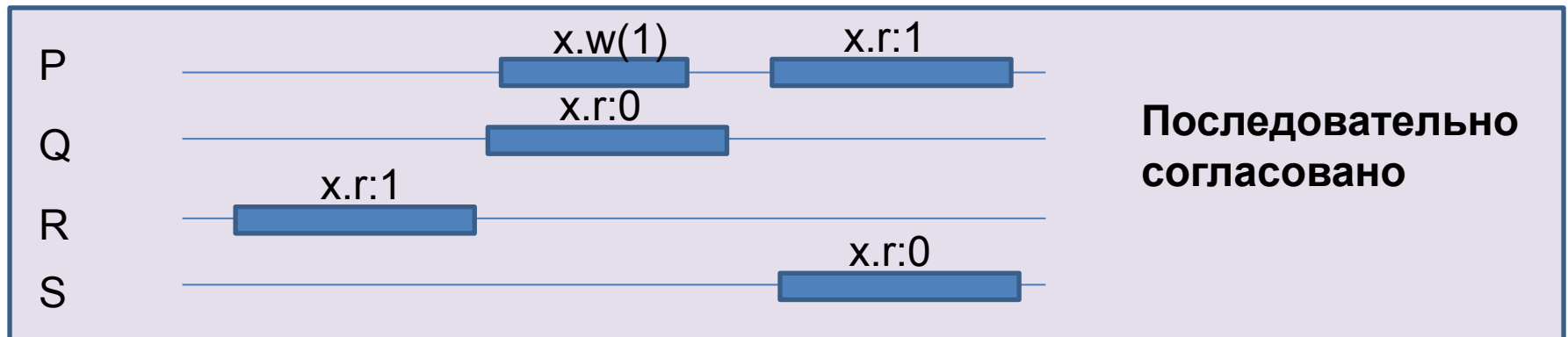
Условия согласованности (корректности)

Нельзя просто сказать:
*“Вот корректная реализация
многопоточной очереди”*.
Корректная в каком смысле?

- Условий согласованности много:
 - Согласованность при покое (quiescent consistency)
 - **Последовательная согласованность (sequential consistency)**
 - **Линеаризуемость (linearizability)**
 - и другие

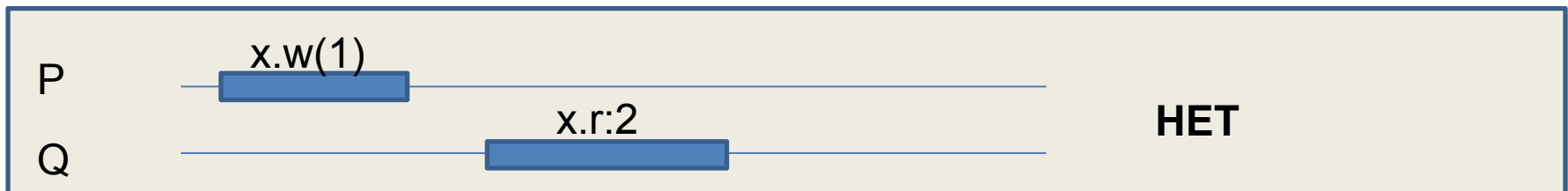
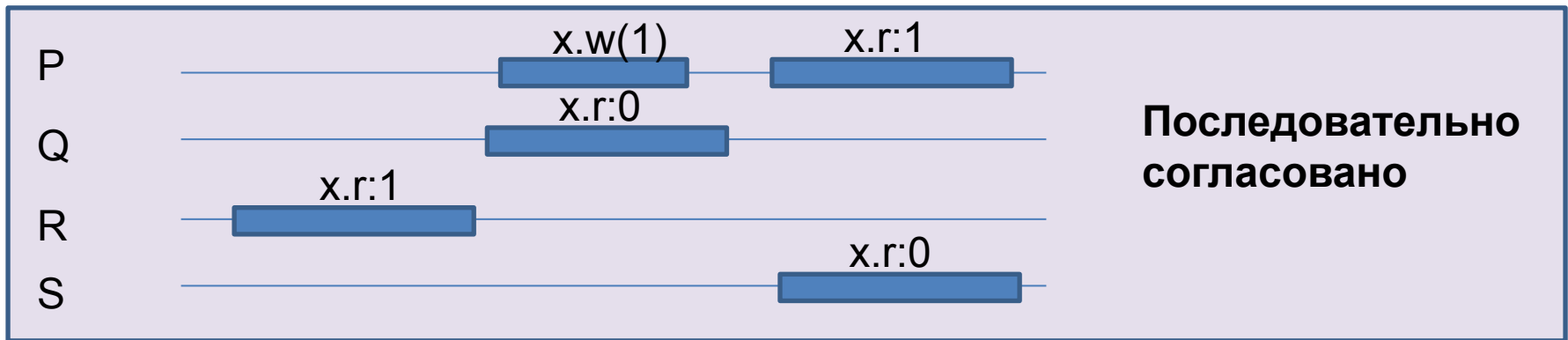
Последовательная согласованность

- Исполнение **последовательно согласованно**, если можно сопоставить эквивалентное ему допустимое последовательное исполнение, которое сохраняет **программный порядок** – порядок операций на каждом потоке



Последовательная согласованность

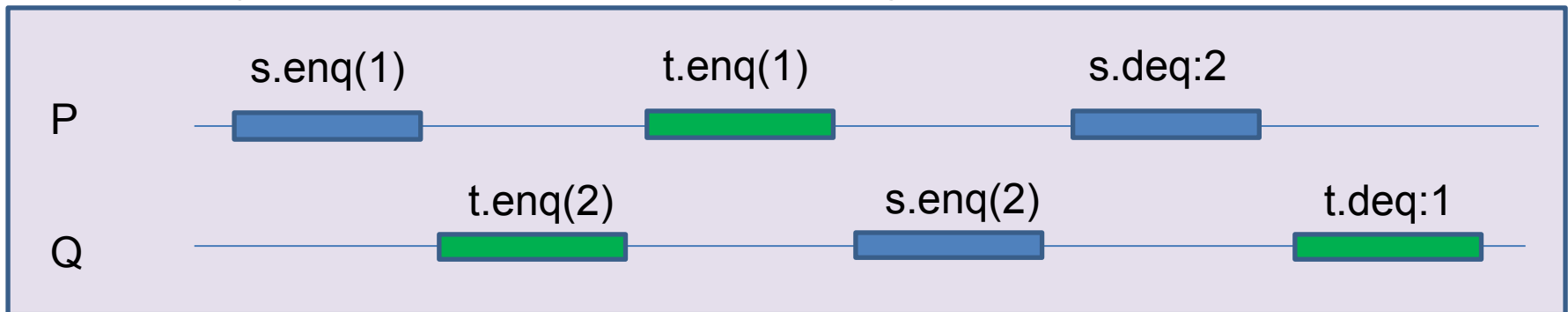
- Исполнение **последовательно согласованно**, если можно сопоставить эквивалентное ему допустимое последовательное исполнение, которое сохраняет **программный порядок** – порядок операций на каждом потоке



Последовательная согласованность

- Последовательная согласованность на каждом объекте по отдельности не влечет последовательную согласованность всего исполнения

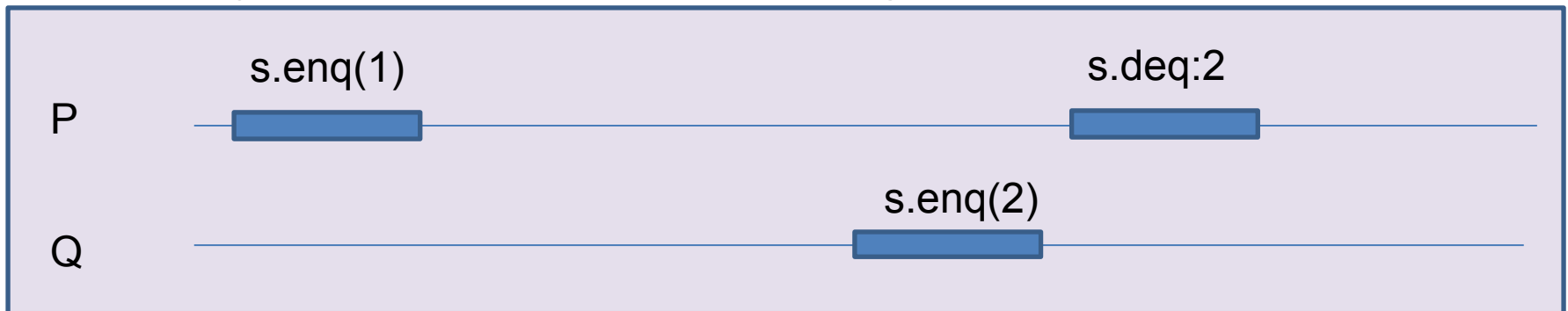
ПРИМЕР: (здесь s и t это две FIFO очереди)



Последовательная согласованность

- Последовательная согласованность на каждом объекте по отдельности не влечет последовательную согласованность всего исполнения

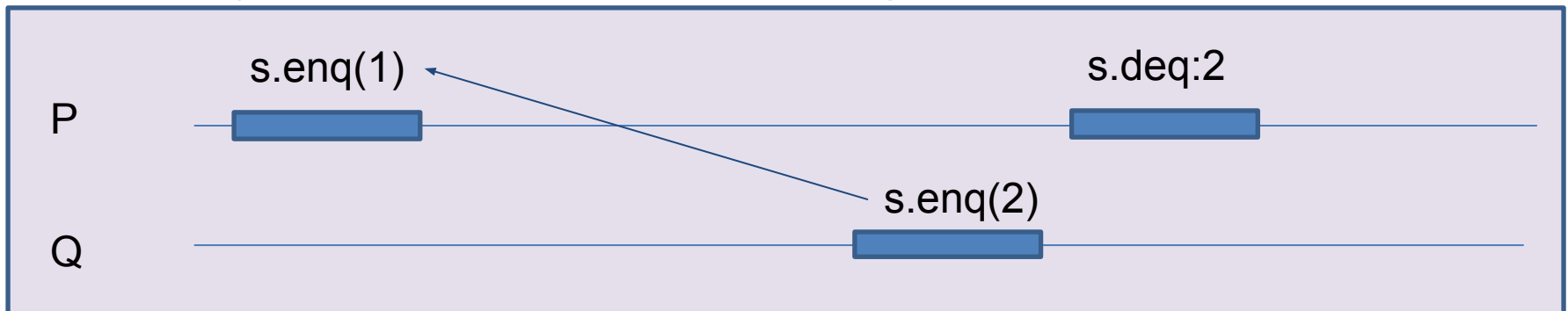
ПРИМЕР: (здесь s и t это две FIFO очереди)



Последовательная согласованность

- Последовательная согласованность на каждом объекте по отдельности не влечет последовательную согласованность всего исполнения

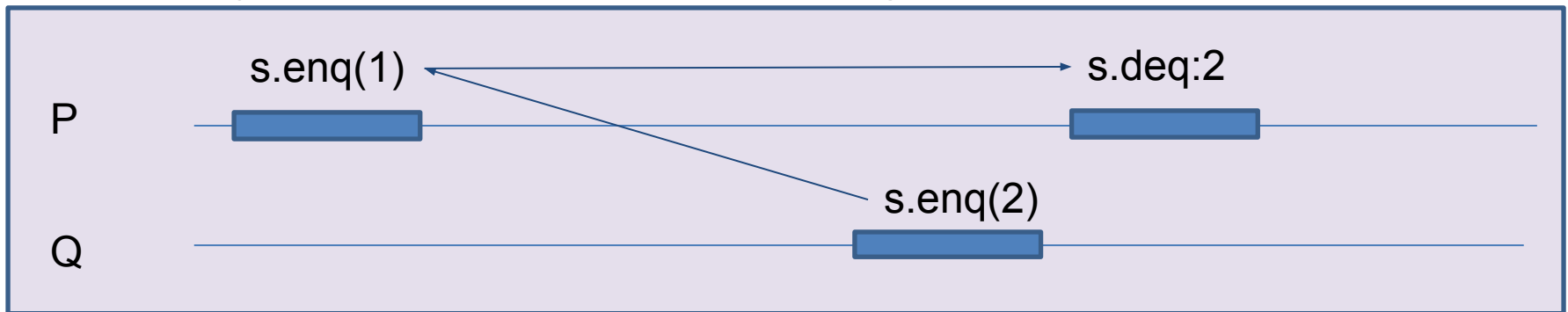
ПРИМЕР: (здесь s и t это две FIFO очереди)



Последовательная согласованность

- Последовательная согласованность на каждом объекте по отдельности не влечет последовательную согласованность всего исполнения

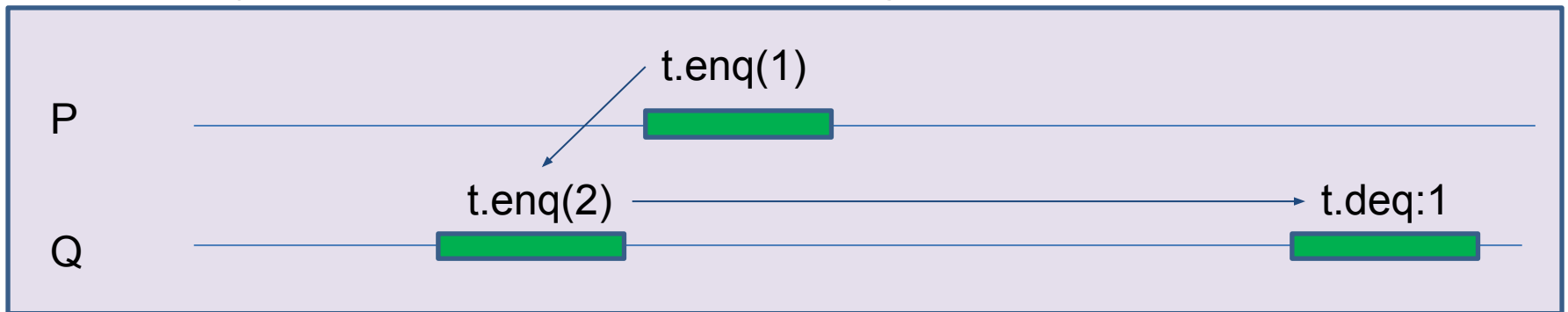
ПРИМЕР: (здесь s и t это две FIFO очереди)



Последовательная согласованность

- Последовательная согласованность на каждом объекте по отдельности не влечет последовательную согласованность всего исполнения

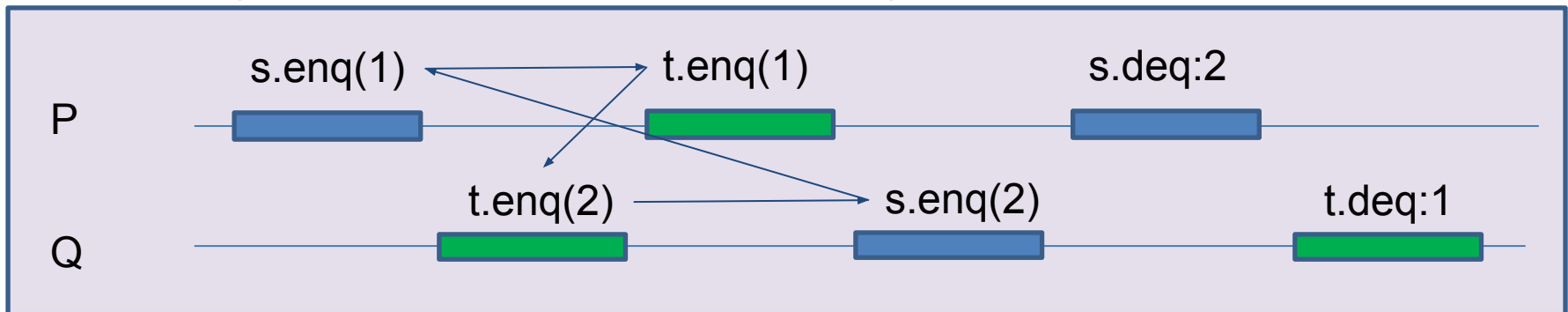
ПРИМЕР: (здесь s и t это две FIFO очереди)



Последовательная согласованность

- Последовательная согласованность на каждом объекте по отдельности не влечет последовательную согласованность всего исполнения

ПРИМЕР: (здесь s и t это две FIFO очереди)

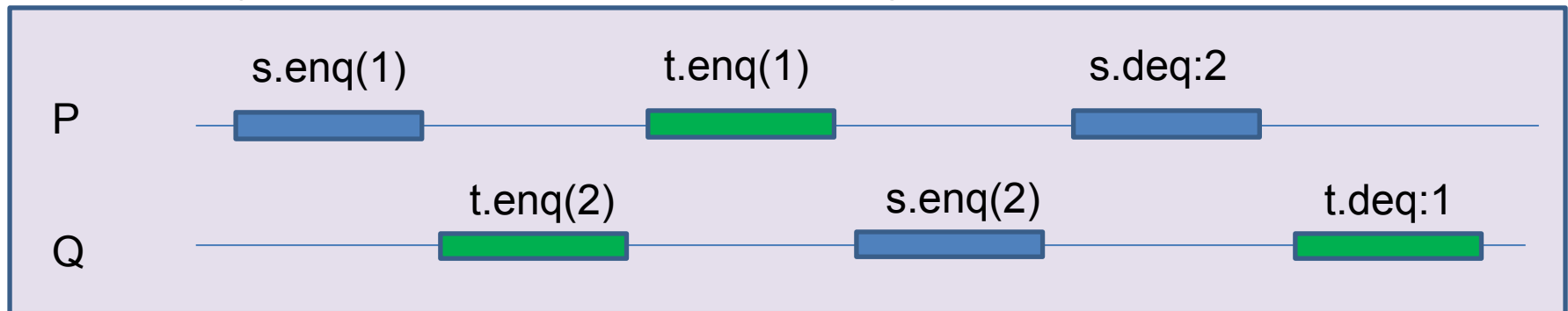


Ой! Цикл -- не упорядочить линейно

Последовательная согласованность

- Последовательная согласованность на каждом объекте по отдельности не влечет последовательную согласованность всего исполнения

ПРИМЕР: (здесь s и t это две FIFO очереди)



Последовательную согласованность нет смысла использовать для спецификации поведения отдельных объектов в системе

Последовательная согласованность

- Используется как условие корректности исполнения **всей системы в целом**
 - когда нет никакого «внешнего» наблюдателя, который может «увидеть» фактический порядок между операциями на разных процессах

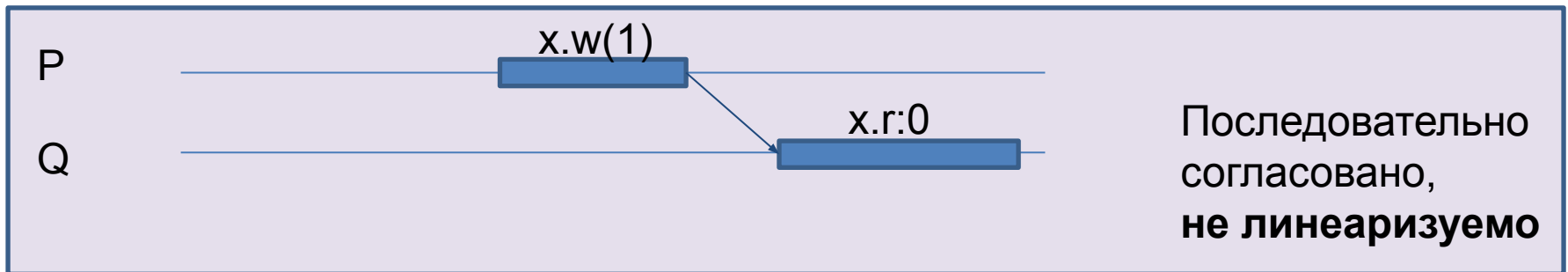
Последовательная согласованность не говорит о том, когда операция физически на самом деле была выполнена

Последовательная согласованность

- Используется как условие корректности исполнения **всей системы в целом**
 - когда нет никакого «внешнего» наблюдателя, который может «увидеть» фактический порядок между операциями на разных процессах
- **Модель памяти** языка программирования и системы исполнения кода используют последовательную согласованность для своих формулировок
 - В том числе, **C++11** и **Java 5** (JMM = JLS Chapter 17)

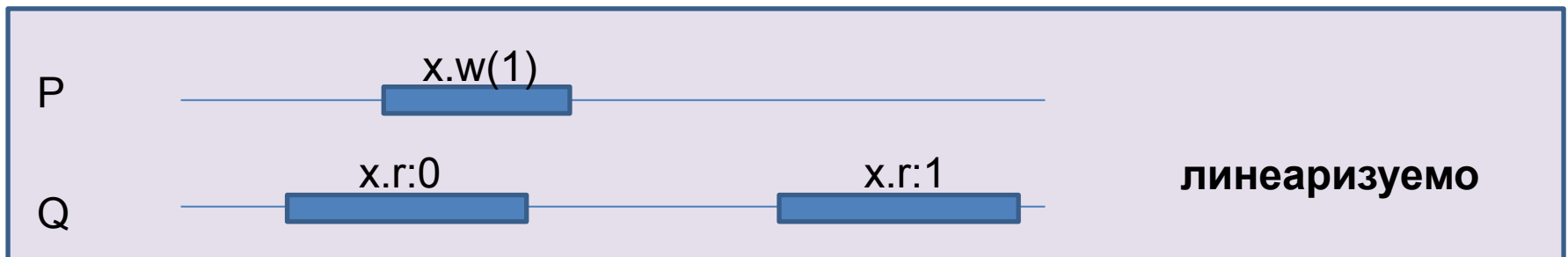
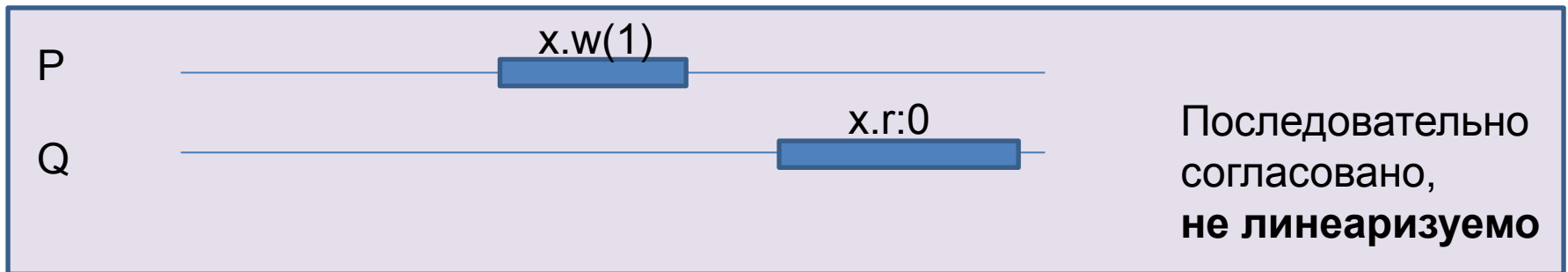
Линеаризуемость

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



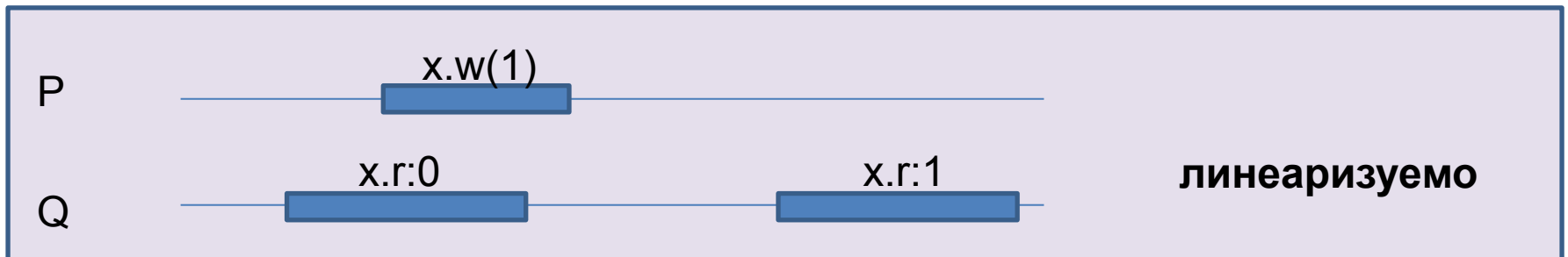
Линеаризуемость

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



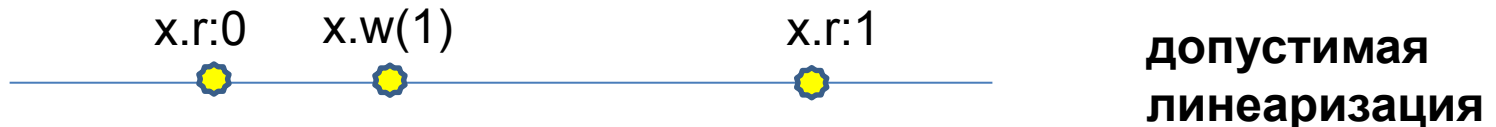
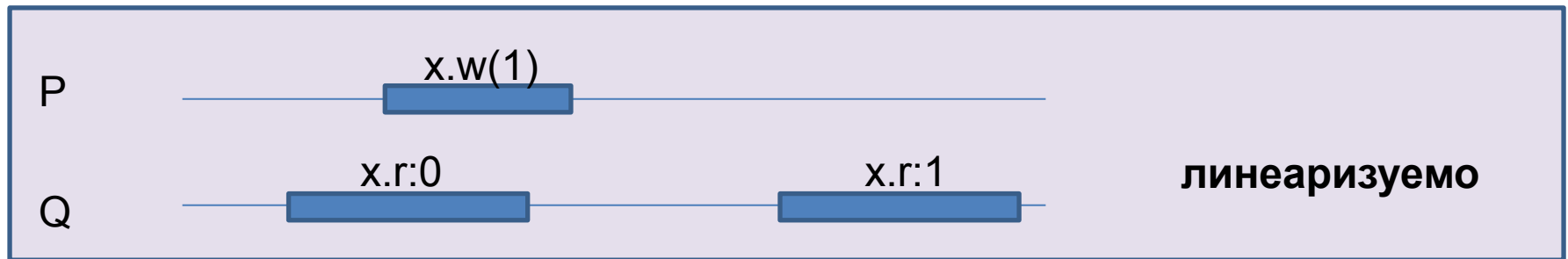
Линеаризуемость

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



Линеаризуемость

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



Свойства линеаризуемости: локальность

- Исполнение линеаризуемо *тогда и только тогда*, когда линеаризуемо исполнение на каждом объекте по отдельности
- **ДОКАЗАТЕЛЬСТВО**
 - Тогда: очевидно
 - Только тогда:
 - Пусть H линеаризуемо на каждом объекте x
 - Возьмем объединение:
 - линеаризации отношения “произошло до” на каждом объекте \rightarrow_x
 - исходное отношение “произошло до” \rightarrow_H
 - Транзитивно замкнем (сохранился исходный порядок и на объектах)
 - Надо доказать что получилось *ацикличное* отношение
 - **Докажем от противного**

Свойства линеаризуемости: локальность

- Если получился цикл в транзитивном замыкании?
 - Любые два последовательность ребра \rightarrow_x или \rightarrow_H можно объединить (так как каждое из отношений транзитивно)
 - Двух последовательных ребер на разных объектах \rightarrow_x и \rightarrow_y быть не может (каждая операция происходит ровно над одним объектом)
 - Значит имеем ситуацию $e \rightarrow_H f \rightarrow_x g \rightarrow_H h$
 - Но это значит по определению отношения “произошло до”:
 - $e \rightarrow_H f \Rightarrow res(e) < inv(f)$
 - $f \rightarrow_x g \Rightarrow inv(f) < res(g)$
 - Потому что $<_H$ это полный порядок и если бы было наоборот, то есть $res(g) < inv(f)$, что значит $g \rightarrow_H f$ что противоречит $f \rightarrow_x g$
 - $g \rightarrow_H h \Rightarrow res(g) < inv(h)$
 - Из этого получаем по транзитивности $<_H$ и по определению \rightarrow_H :
 - $res(e) < inv(h) \Rightarrow e \rightarrow_H h$
 - Таким образом любой цикл можно свести к циклу в исходном

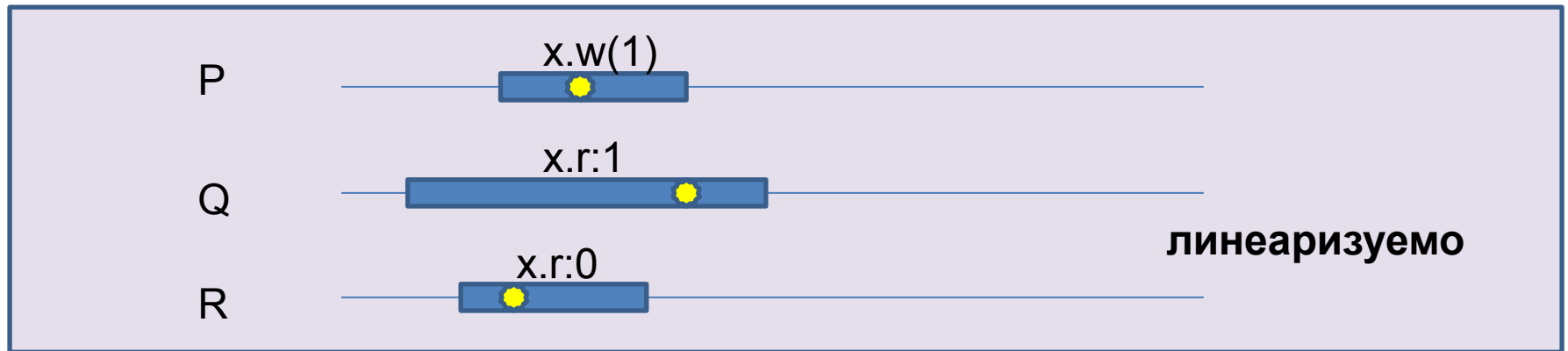
Линеаризуемость и атомарность

- Операции над линеаризуемыми объектами называют **атомарными**
- **Замечание 1:** При доказательстве локальности линеаризуемости мы воспользовались **полным порядком** над индивидуальными событиями (“вызов операции” и “результат операции”).
 - В системах с общей памятью есть возможность “упорядочить” все операции используя общую память как механизм синхронизации
- **Замечание 2:** Понятие “линеаризуемости” отсутствует в распределенных системах, где нет общей памяти.

Линеаризуемость в глобальном времени


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

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



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

Подразумеваемая линейризуемость

~~Нельзя просто сказать:~~
*“Вот корректная реализация
многопоточной очереди”.*
Корректная в каком смысле?

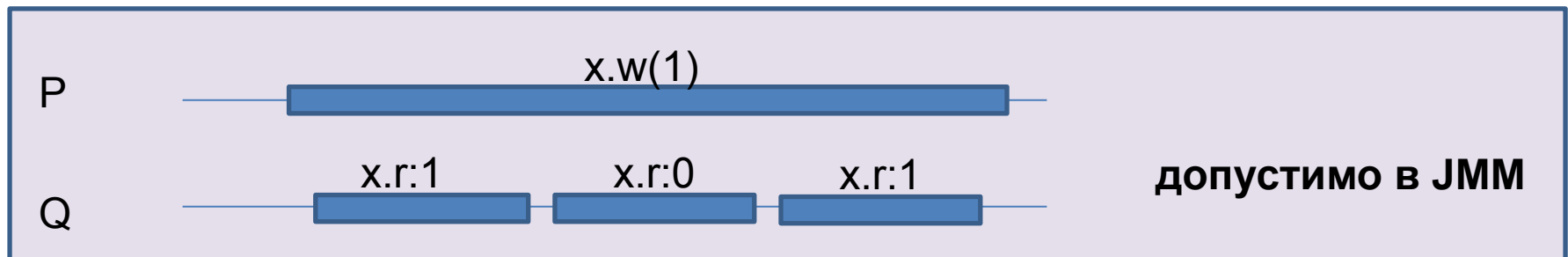
Подразумеваемая линейризуемость

~~Нельзя просто сказать:~~
*“Вот корректная реализация
многопоточной очереди”.*
Корректная в каком смысле?

- Когда говорят что 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)



Применительно к Java

- Если программа **корректно синхронизирована** (в ней нет **гонок**), то JMM дает гарантию **последовательно согласованного** исполнения для всего кода
 - Даже для кода работающего с **не volatile** переменными
 - Если доступ к ним **синхронизирован** (через операции синхронизации) и гонок не возникает
- В C++11 есть примитивы (**std::atomic**) для получения аналогичных гарантий

Попробуем на практике (2)

```
@JCStressTest
@State
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE)
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;
    }
}
```

Попробуем на практике (2)

```
@JCStressTest
@State
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE)
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;
    }
}
```

Попробуем на практике (2)

```
@JCStressTest
@State
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE)
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;
    }
}
```

Запустим!

Попробуем на практике (2)

```
@JCStressTest
@State
@Outcome(id = "0, 1", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE)
public class SimpleTest2 {
    volatile int x;
    volatile int y;

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

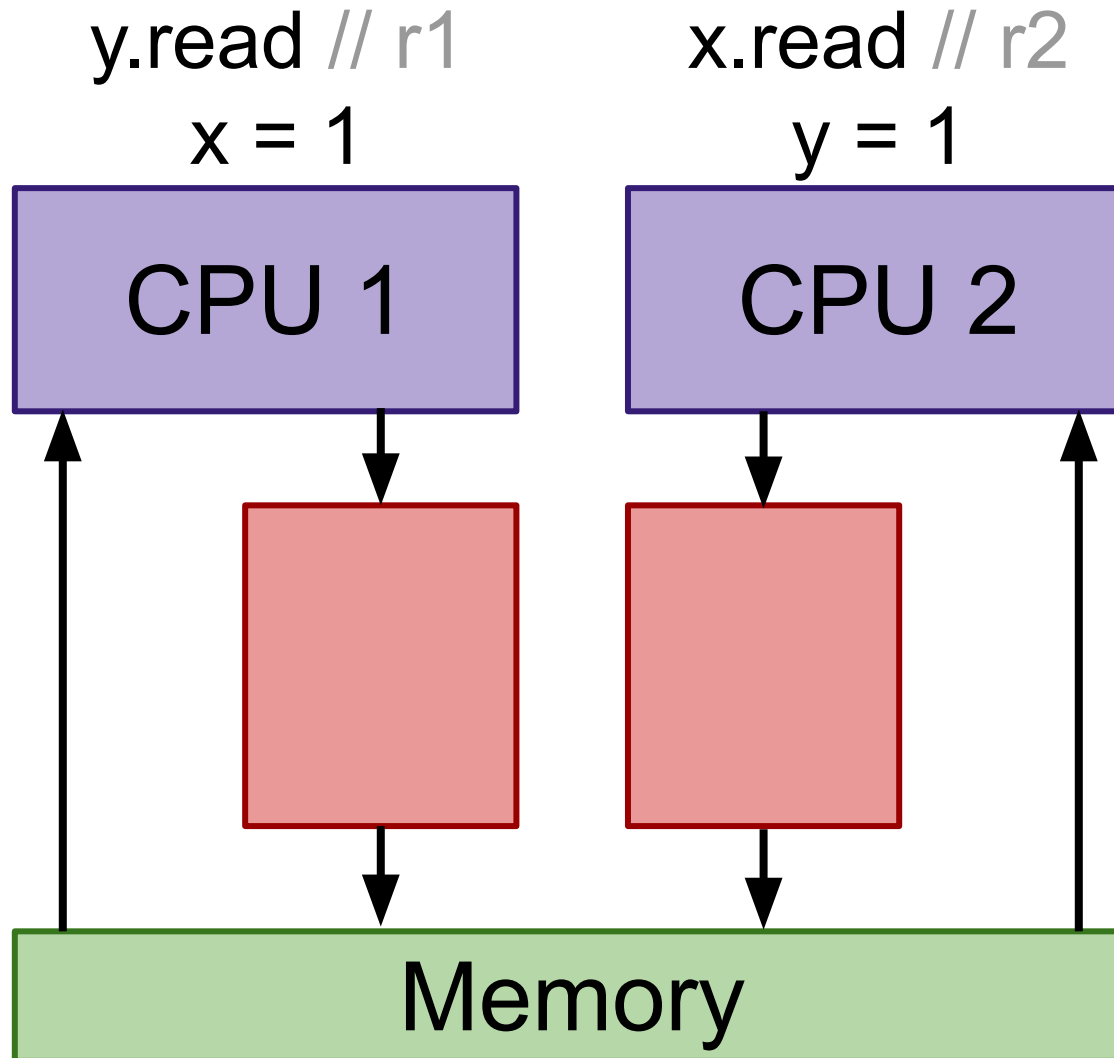
    @Actor
    public void threadQ(IntResult2 r) {
        y = 1;
        r.r2 = x;
    }
}
```

Запустим!

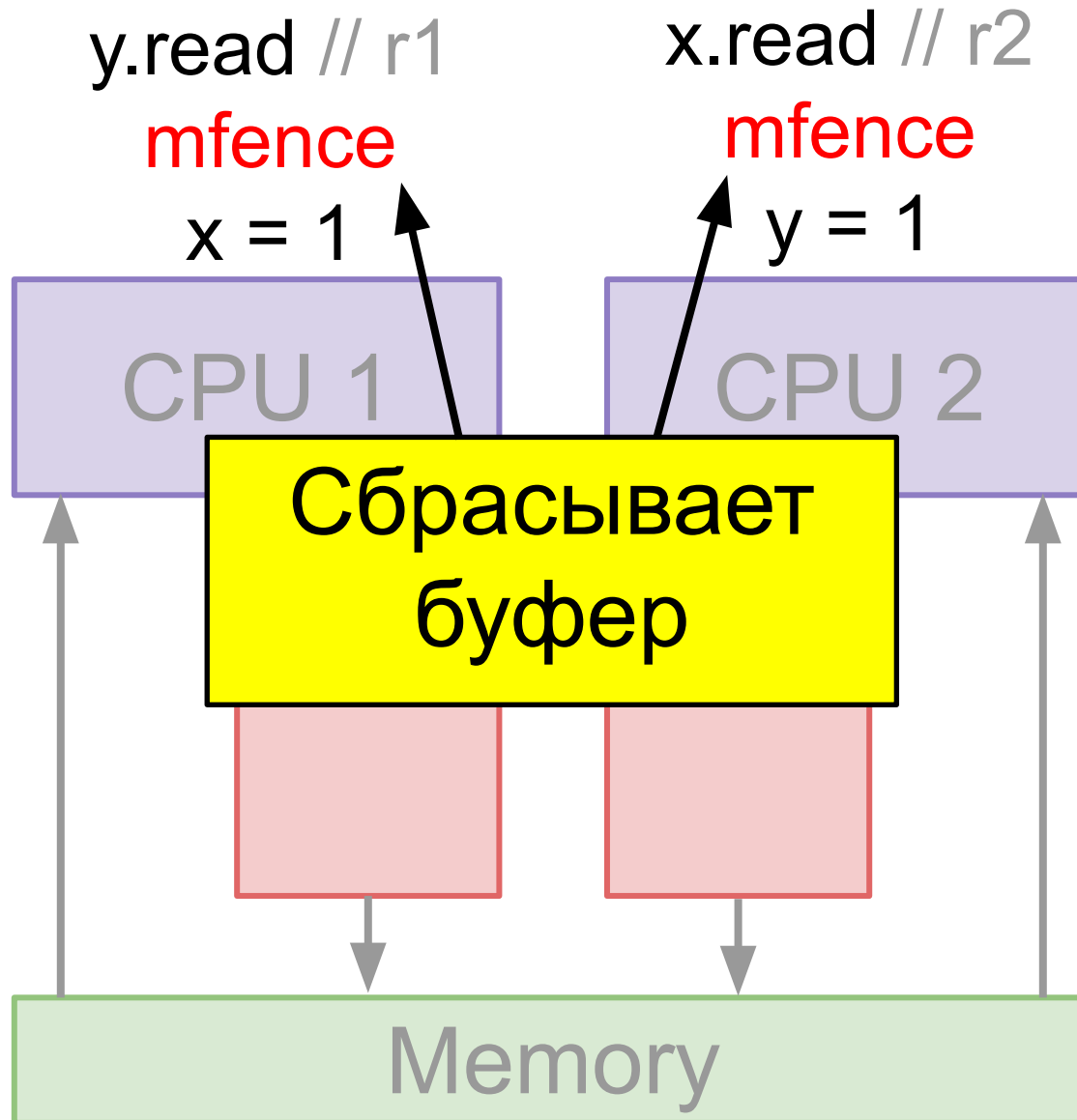
[PASSED]	SimpleTest2
State	Occurrences
0, 1	46,917,076
1, 0	37,773,885
1, 1	21,989

Что мы и ожидали
исходя из модели
чередования
операций!

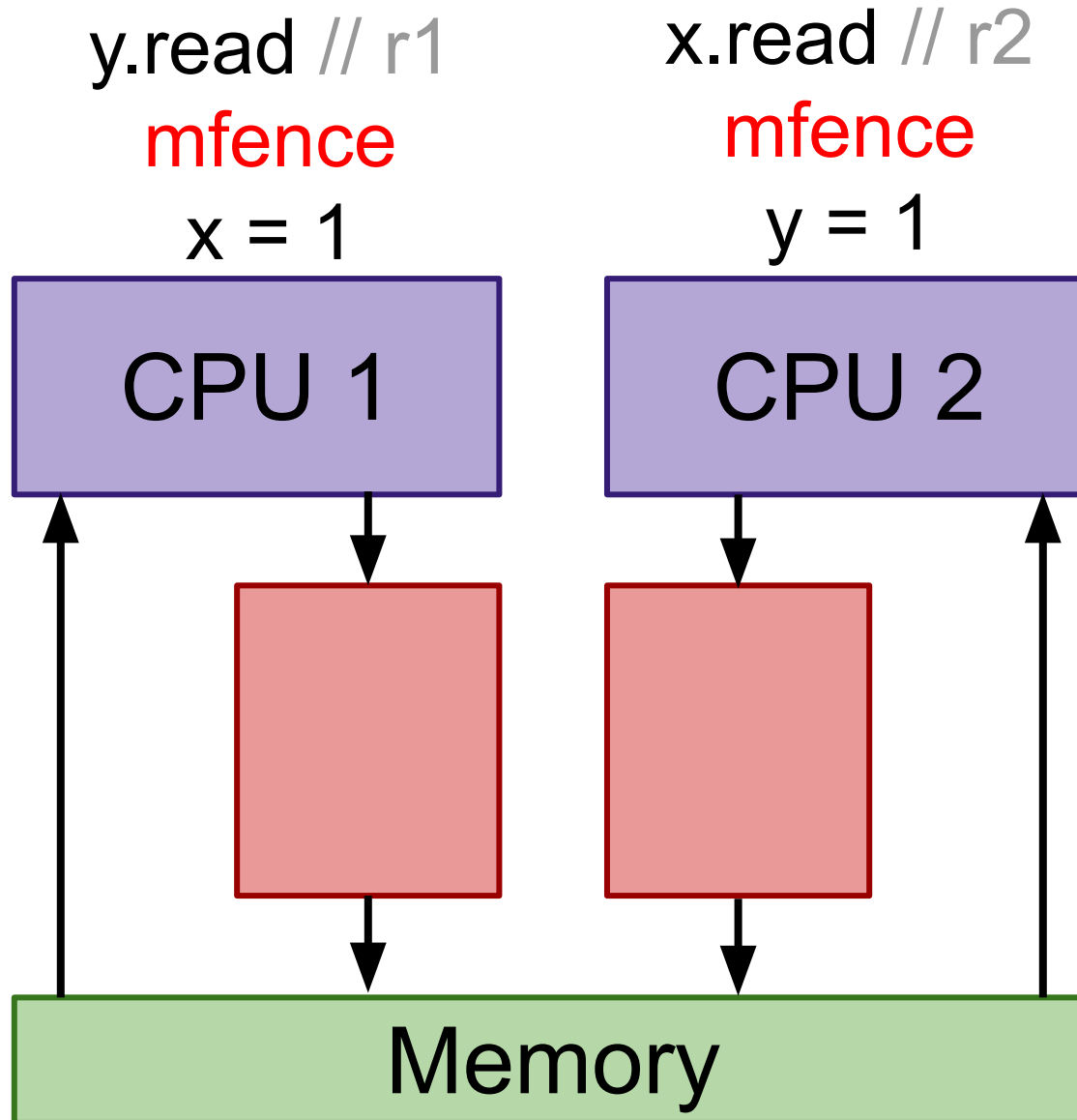
Как исправить на x86 (TSO)



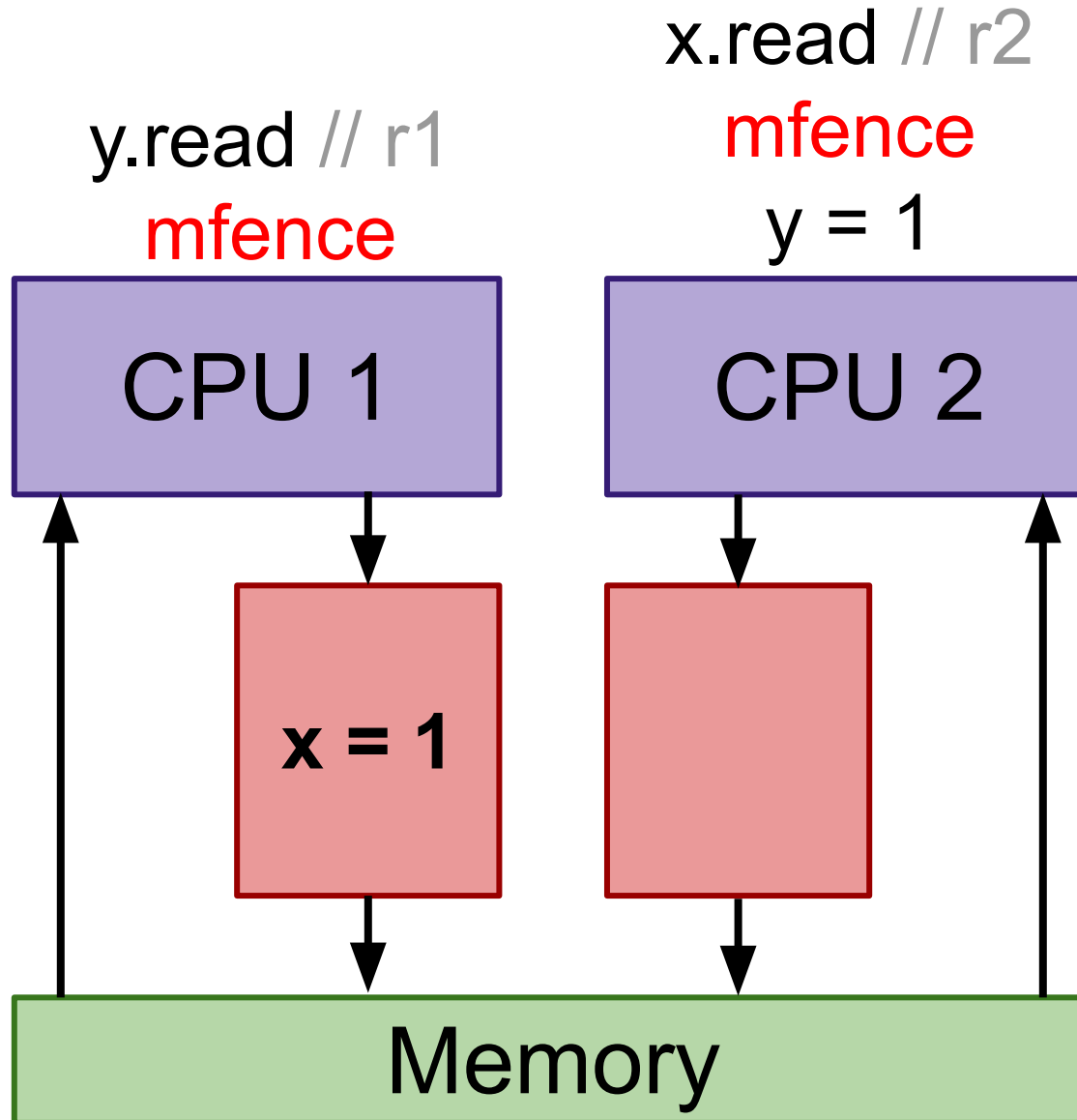
Как исправить на x86 (TSO)



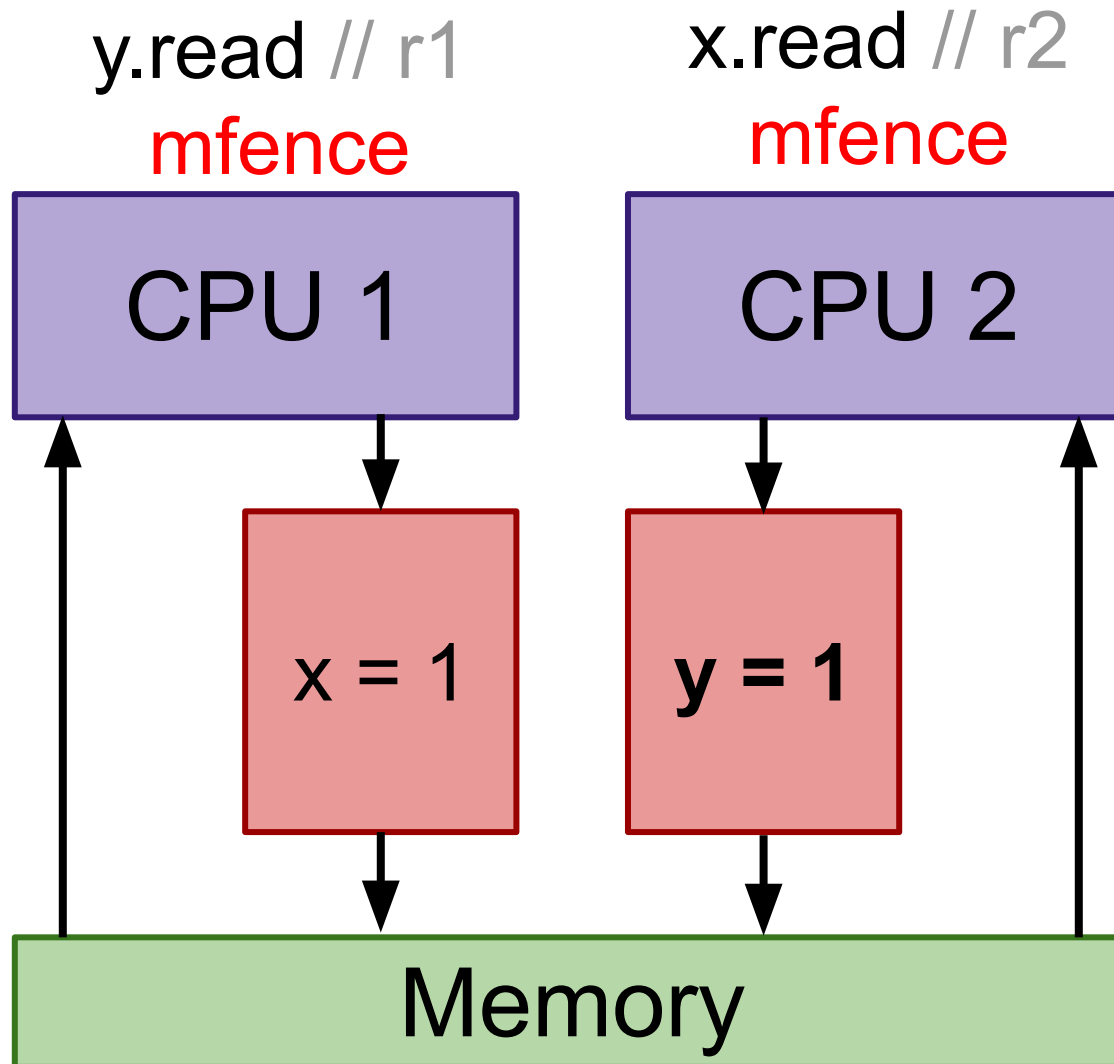
Как исправить на x86 (TSO)



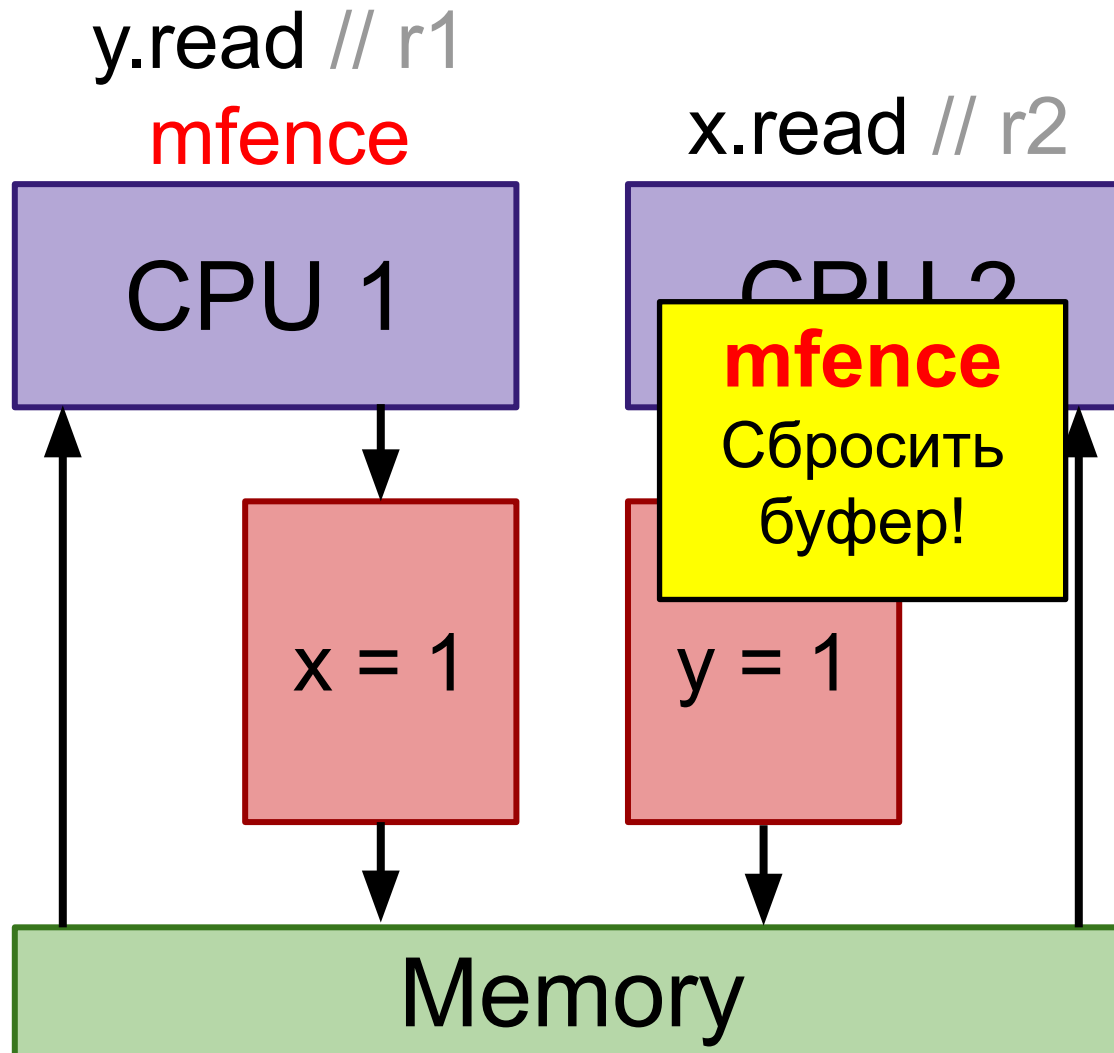
Как исправить на x86 (TSO)



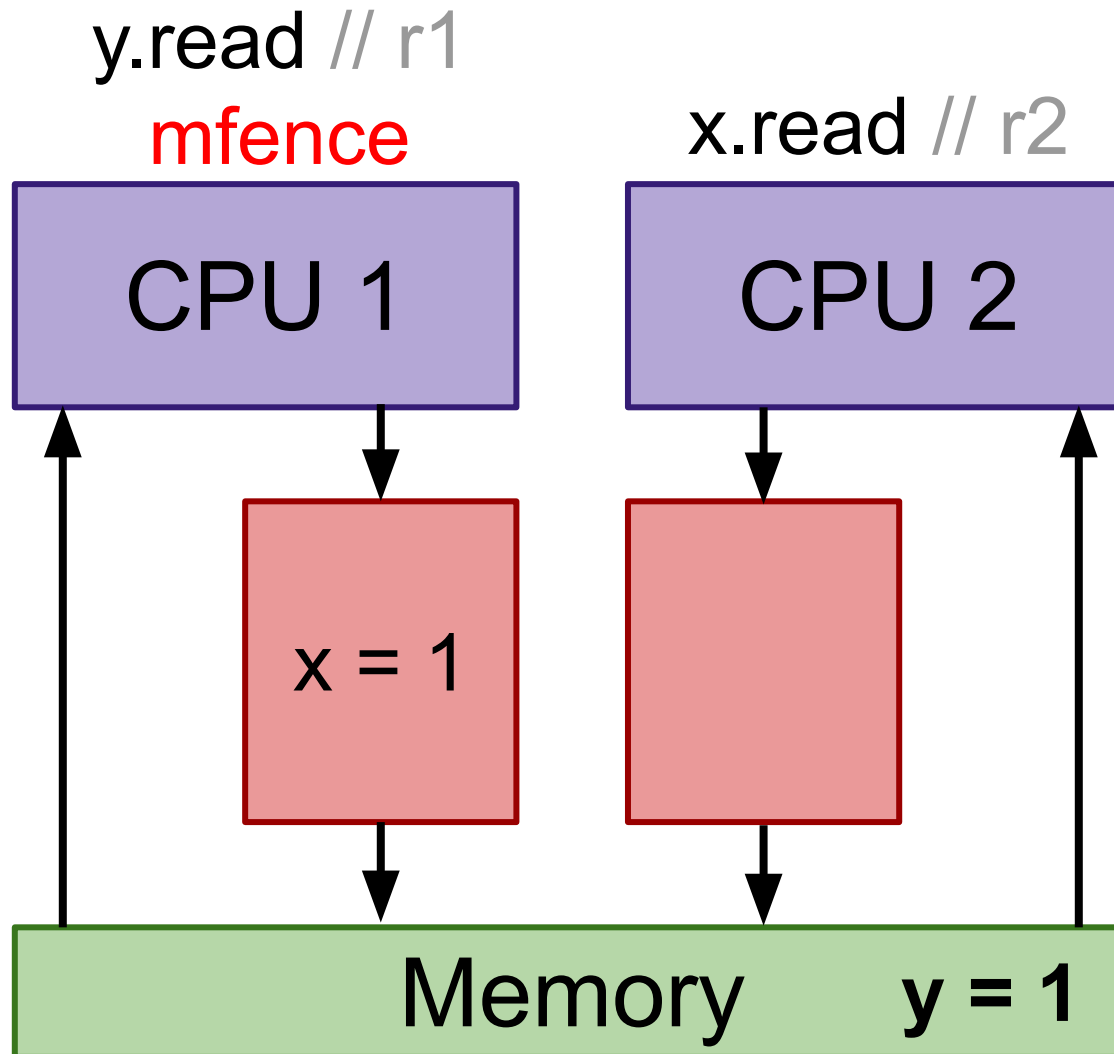
Как исправить на x86 (TSO)



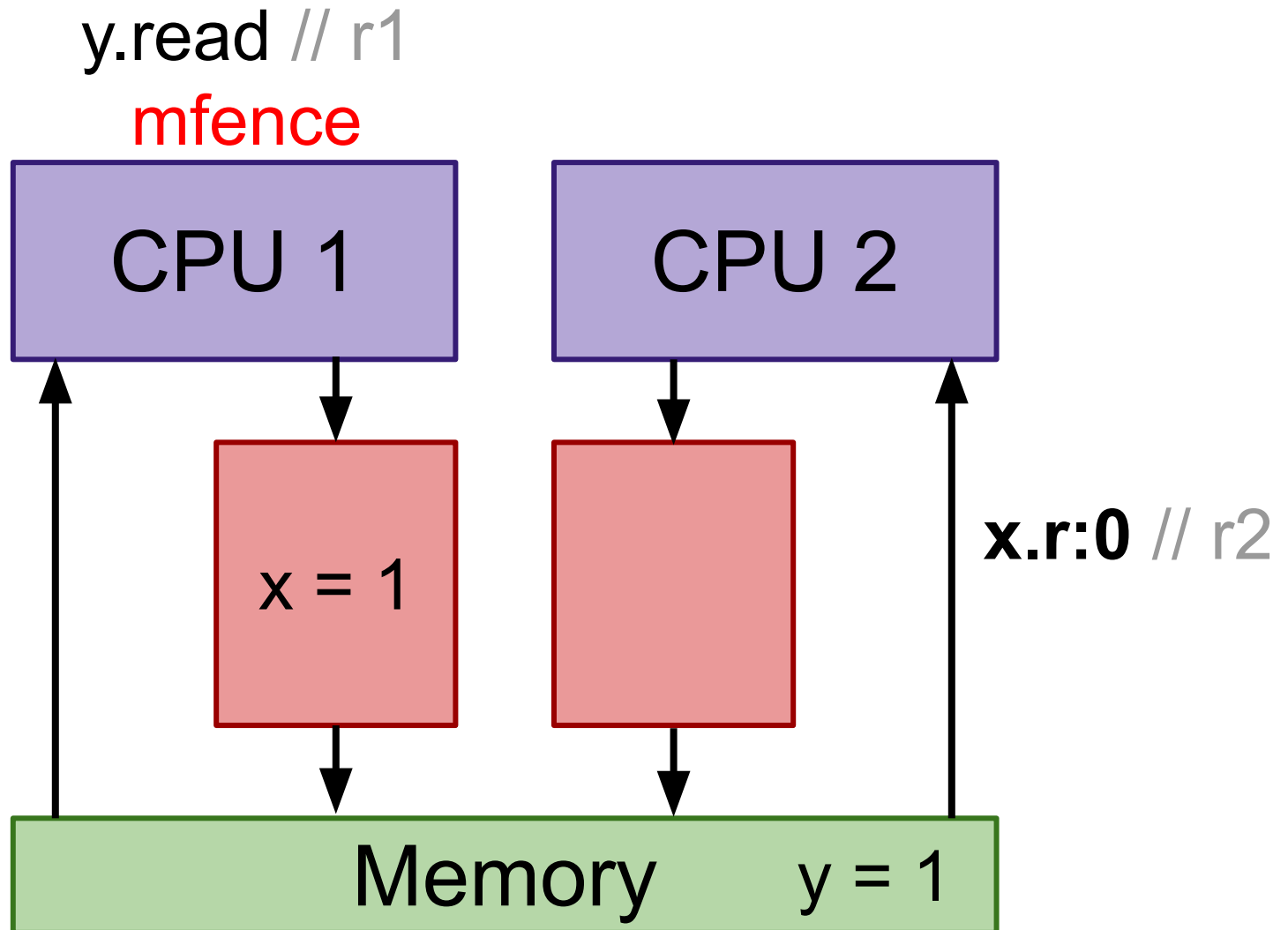
Как исправить на x86 (TSO)



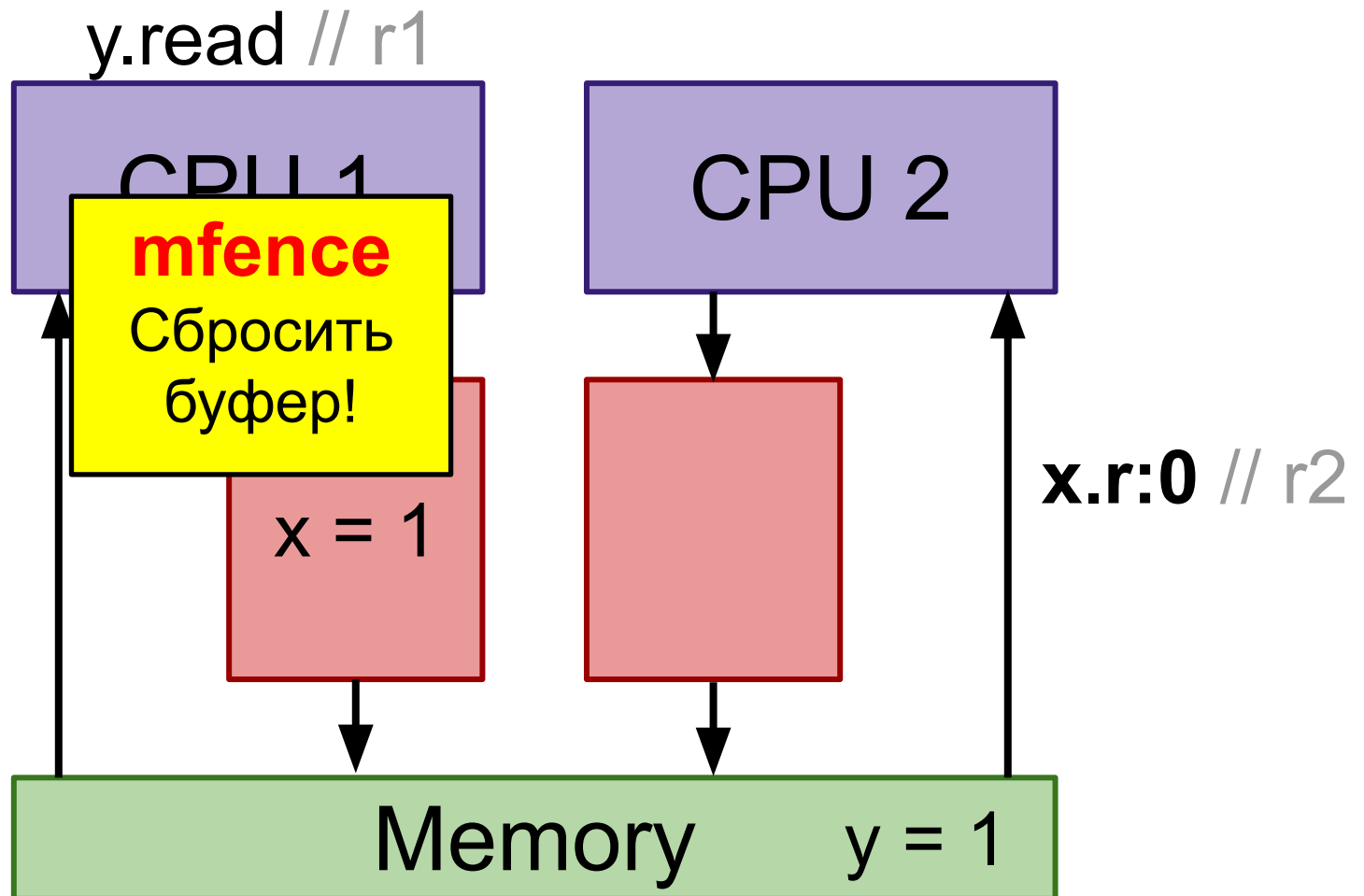
Как исправить на x86 (TSO)



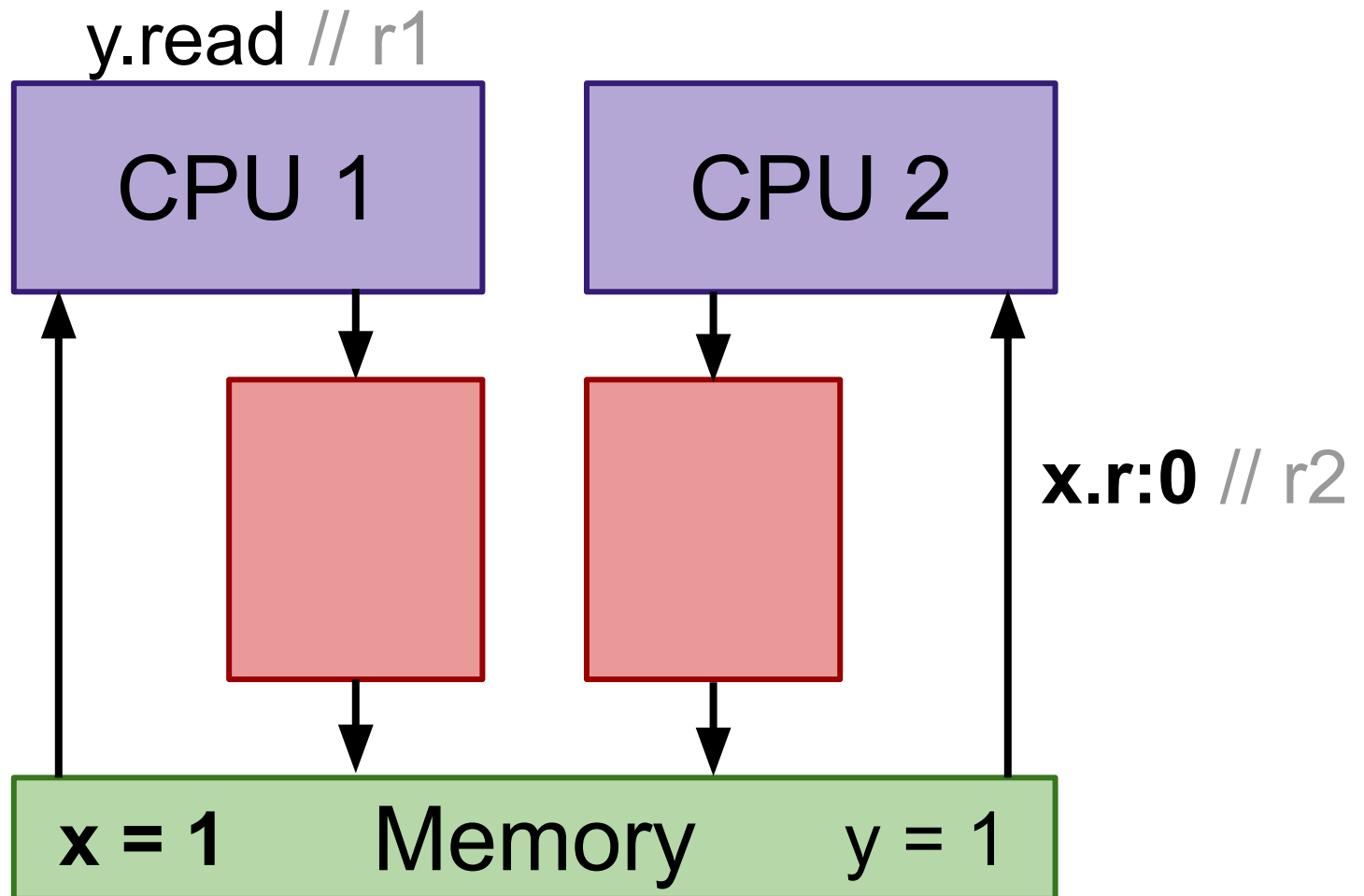
Как исправить на x86 (TSO)



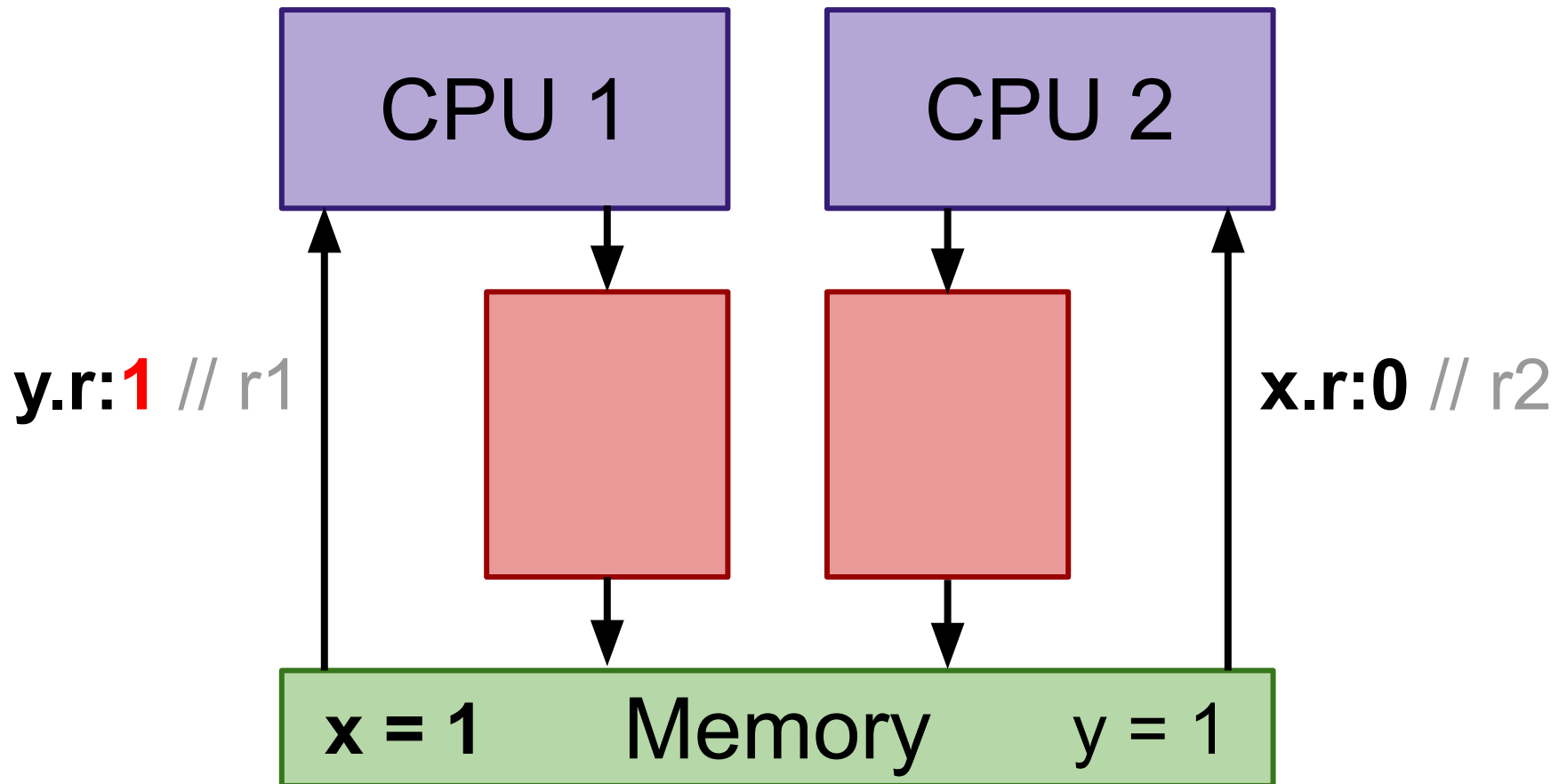
Как исправить на x86 (TSO)



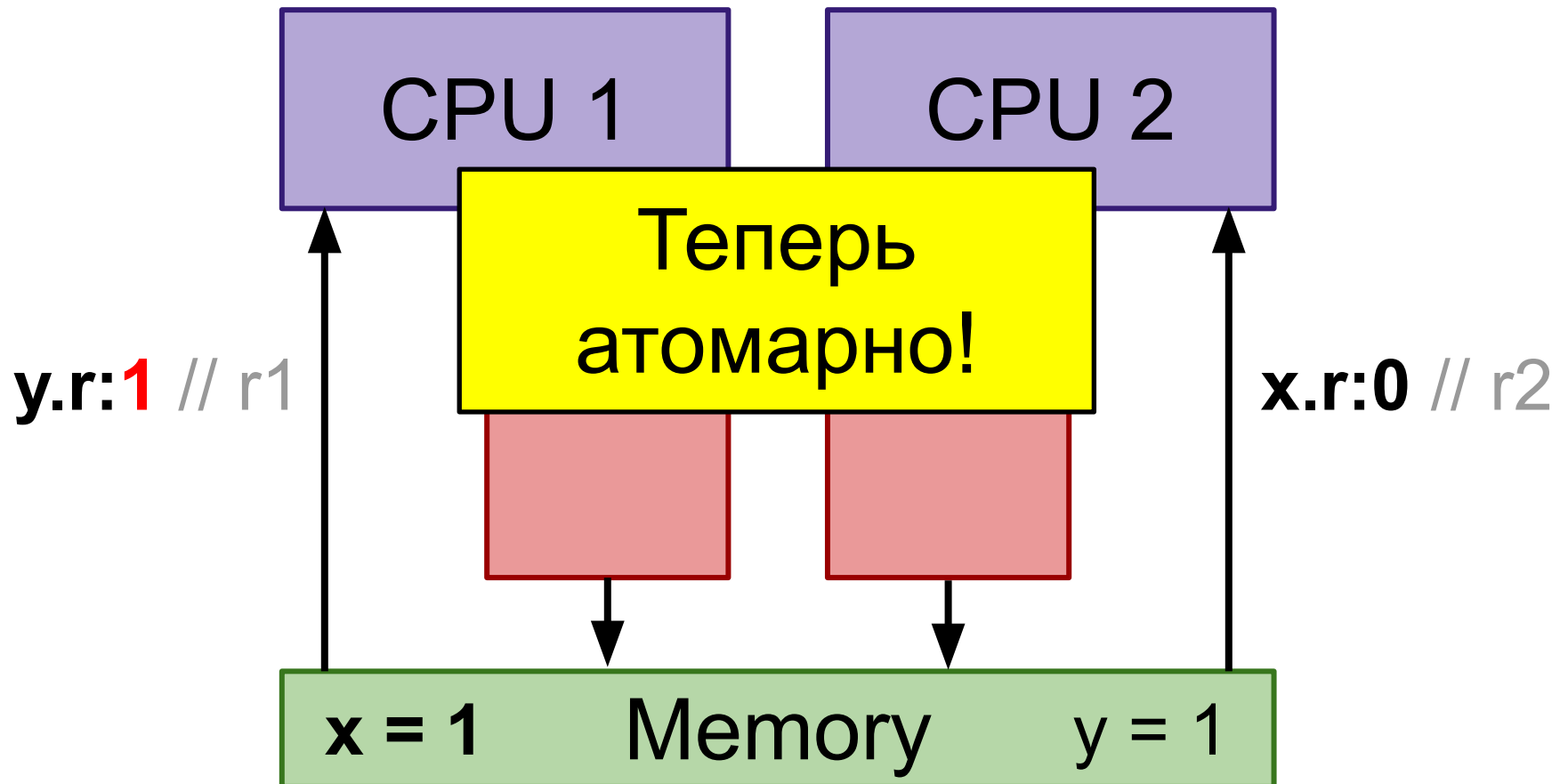
Как исправить на x86 (TSO)



Как исправить на x86 (TSO)



Как исправить на x86 (TSO)



Во что тогда компилируется наш код?

- **SimpleTest1** (non-volatile)

thread P:

```
mov dword ptr [rdi+0xc], 0x1    ;x = 1  
mov r10d, dword ptr [rdi+0x10];y.read
```

thread Q:

```
mov dword ptr [rsi+0x10], 0x1  ;y = 1  
mov r10d, dword ptr [rsi+0xc]  ;x.read
```

Во что тогда компилируется наш код?

- **SimpleTest1** (non-volatile)

thread P:

```
mov dword ptr [rdi+0xc], 0x1    ;x = 1  
mov r10d, dword ptr [rdi+0x10];y.read
```

thread Q:

```
mov dword ptr [rsi+0x10], 0x1  ;y = 1  
mov r10d, dword ptr [rsi+0xc]  ;x.read
```

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

thread P:

```
mov dword ptr [r11+0xc], 0x1    ;x = 1  
lock add dword ptr [rsp],0x0    ;mfence  
mov r10d, dword ptr [r11+0x10];y.read
```

thread Q:

```
mov dword ptr [r11+0x10], 0x1  ;y = 1  
lock add dword ptr [rsp],0x0    ;mfence  
mov r10d, dword ptr [r11+0xc]  ;x.read
```

Во что тогда компилируется наш код?

- SimpleTest1 (non-volatile)

thread P:

```
mov dword ptr [rdi+0xc], 0x1 ;x = 1
```

```
mov r10d, dword ptr [rdi+0x10] ;y.read
```

thread Q:

```
mov dword ptr [rsi+0x10], 0x1 ;y = 1
```

```
mov r10d, dword ptr [rsi+0xc] ;x.read
```

Появился тот
самый барьер!

- SimpleTest2 (volatile)

thread P:

```
mov dword ptr [r11+0xc], 0x1 ;x = 1
```

```
lock add dword ptr [rsp], 0x0 ;mfence
```

```
mov r10d, dword ptr [r11+0x10] ;y.read
```

thread Q:

```
mov dword ptr [r11+0x10], 0x1 ;v = 1
```

```
lock add dword ptr [rsp], 0x0 ;mfence
```

```
mov r10d, dword ptr [r11+0xc] ;x.read
```

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;
    }
}
```


Эффект на производительность?

```
@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;
    }
}
```

Эффект на производительность?

Benchmark	Score	Error	Units
SimpleTest1.group	97 ,150,871.246 ± 28,985,900.030		ops/s
SimpleTest2.group	44 ,168,485.894 ± 2,244,677.631		ops/s

Синхронизация
далеко не бесплатна