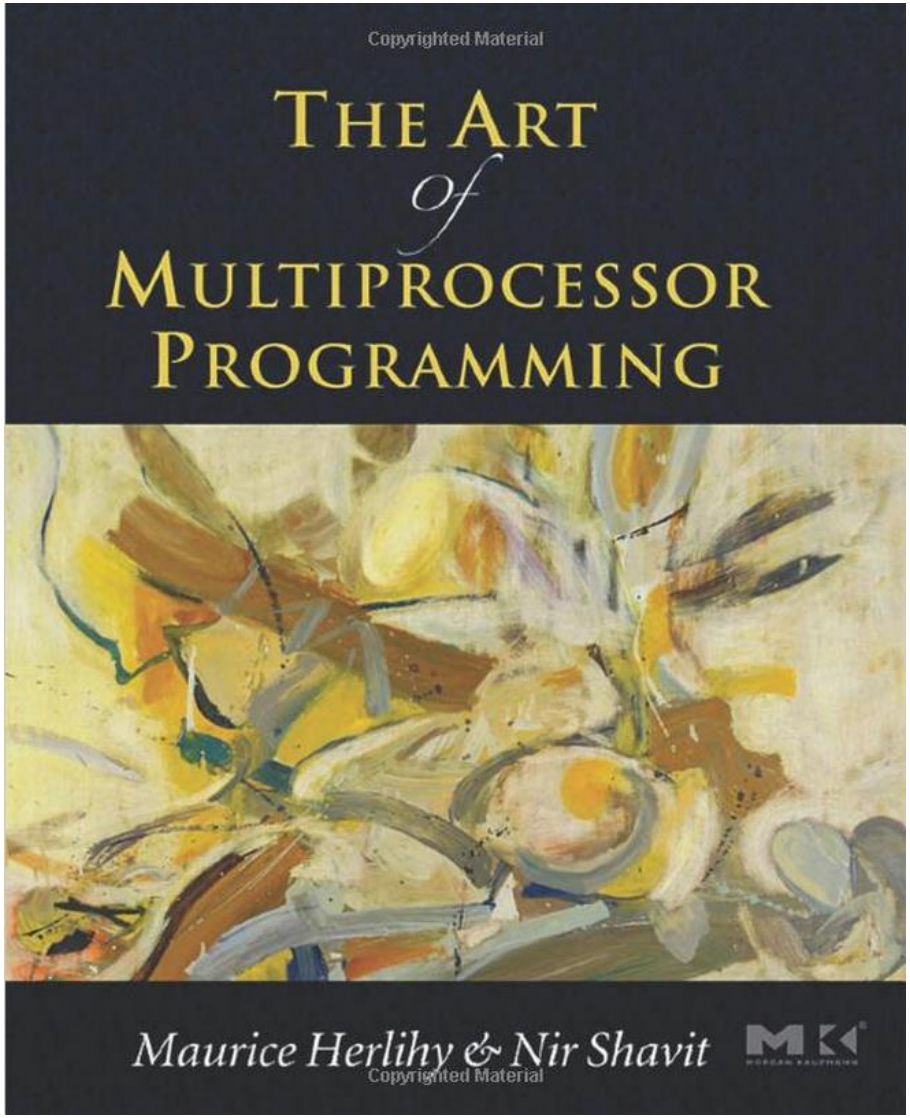
Лекции «Многопоточное программирование»

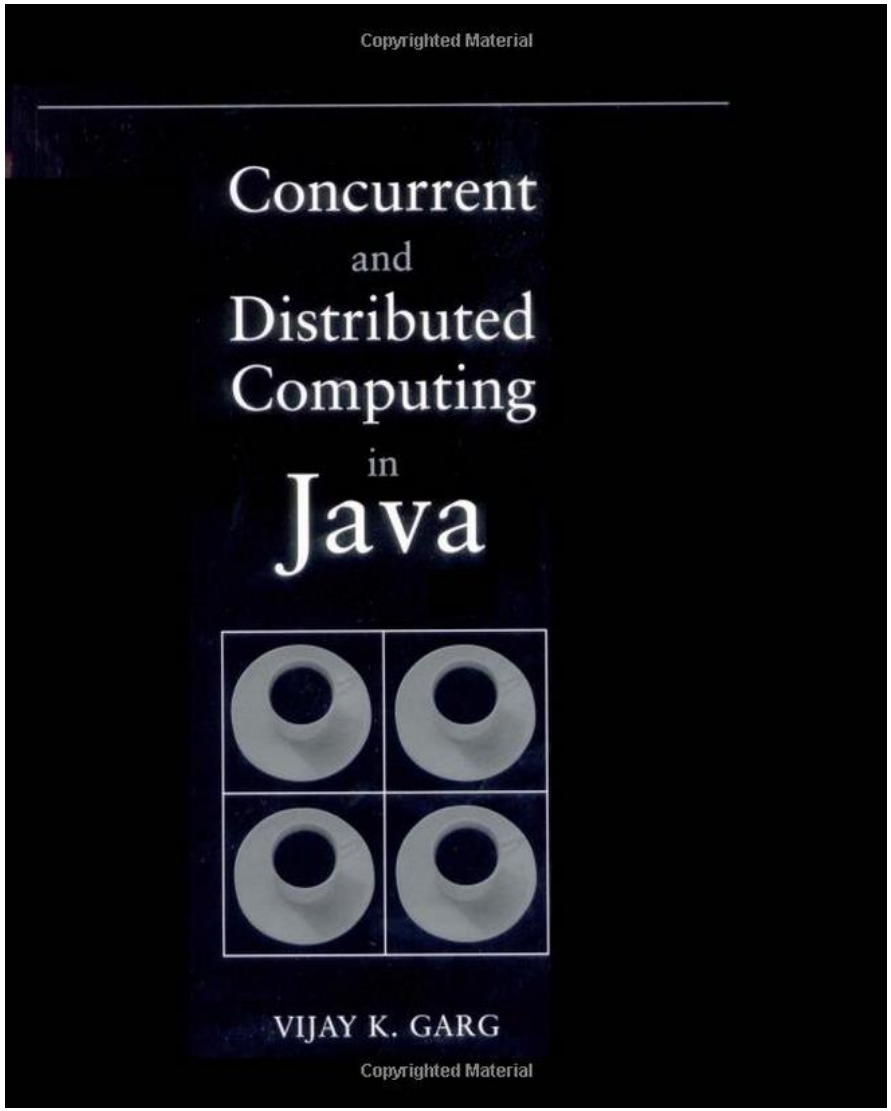
ЛИТЕРАТУРА

Для самостоятельной подготовки

DEVEXPERTS


Copyrighted Material
THE ART
of
MULTIPROCESSOR
PROGRAMMING
Maurice Herlihy & Nir Shavit
MK
Copyrighted Material

DEVEXPERTS


Copyrighted Material
Concurrent
and
Distributed
Computing
in
Java
VIJAY K. GARG
Copyrighted Material

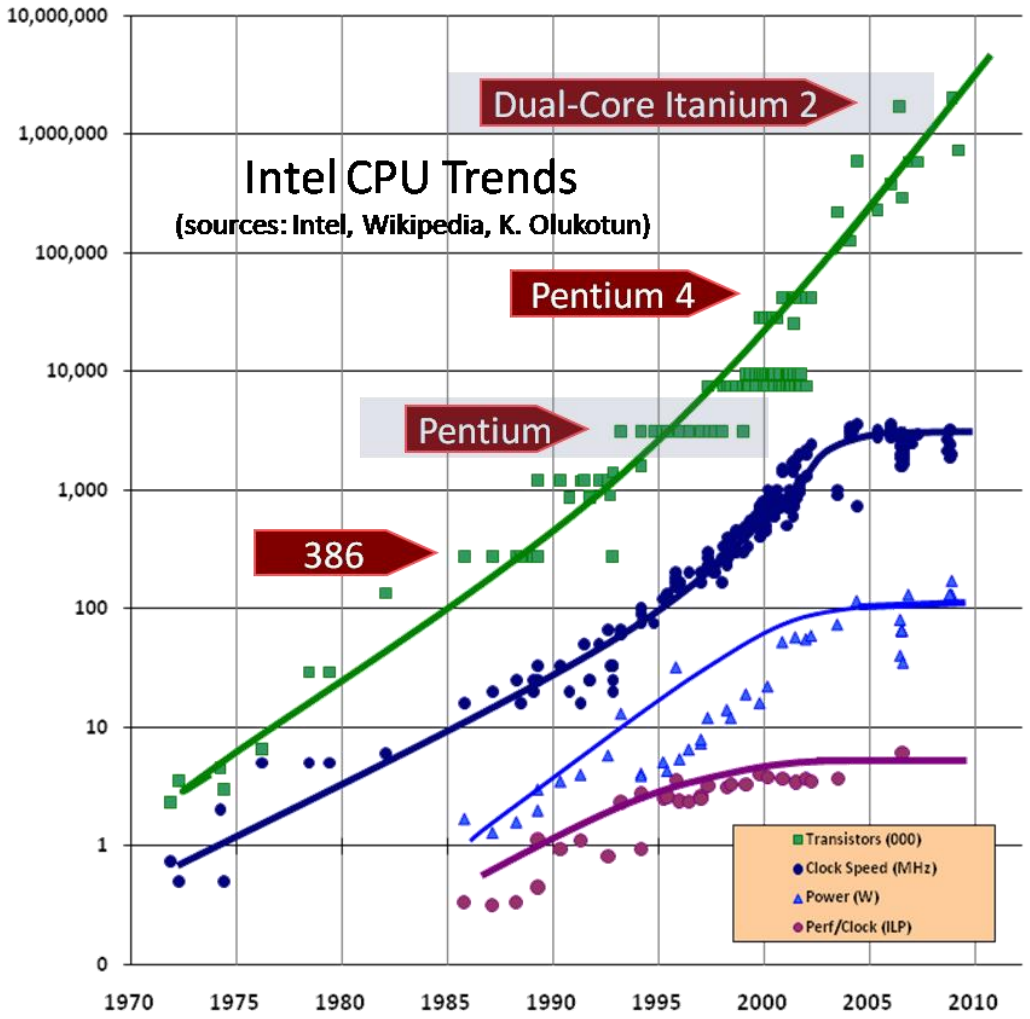
DEVEXPERTS

Лекция 1

ВВЕДЕНИЕ И МОТИВАЦИЯ

DEVEXPERTS

Закон Мура и “The free lunch is over”



Зеленая линия =
Тысяч транзисторов
aka “Закон Мура”

Под ней =
Частота (MHz)

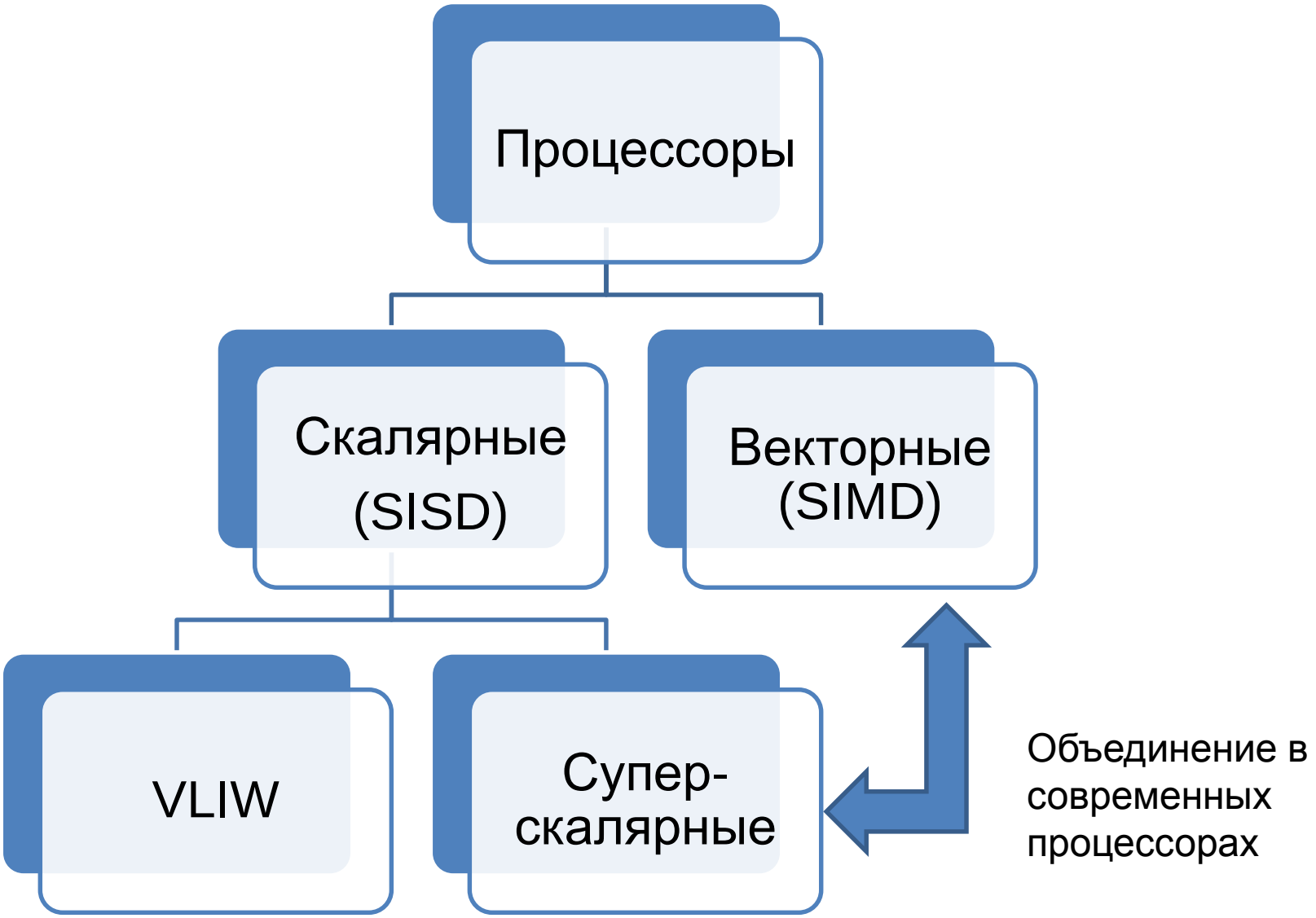
http://www.gotw.ca/publications/concurrency-ddj.htm

Параллелизм на уровне инструкций (ILP – instruction level parallelism)

```
a = b + c; // (1)
d = e + f;  // (2)
```

Нет зависимости по данным: можно выполнить (1) и (2) параллельно

- **Способы использования ILP:**
  - Конвейер
  - Суперскалярное исполнение
    - Внеочередное исполнение
    - Переименование регистров
    - Спекулятивное исполнение
    - Предсказание переходов
  - Длинное машинное слово (VLIW – very long instruction word)
  - Векторизация (SIMD)

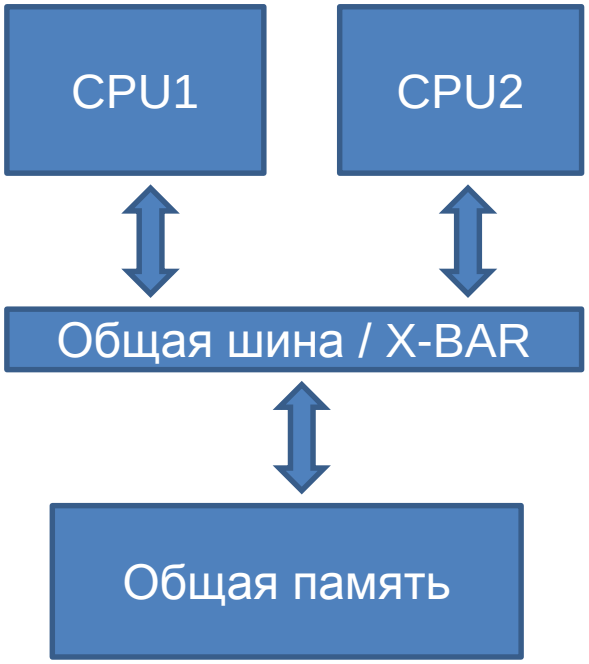

Процессоры
Скалярные (SISD)
Векторные (SIMD)
VLIW
Супер-скалярные
Объединение в современных процессорах

DEVEXPERTS

У параллелизма на уровне инструкций есть предел

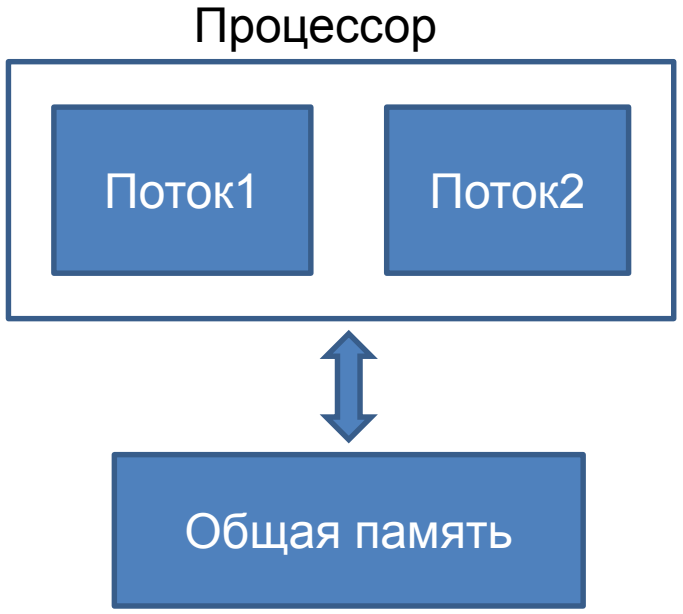
Параллельное программирование

Симметричная мультипроцессорность (SMP – symmetric multiprocessing)



Два (или больше) вычислительных **ядра**.
На каждом свой **поток** исполняемых инструкций

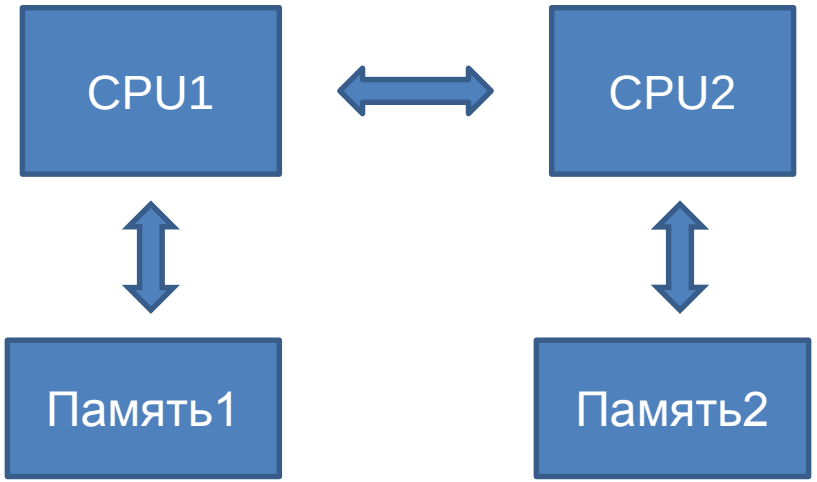
Одновременная многозадачность (SMT – simultaneous multithreading)



Два (или больше) **потока** одновременно исполняются одним физическим вычислительным ядром.

Для программиста выглядит как SMP.

Несимметричный доступ к памяти (NUMA – non-uniform memory access)



Модель программирования такая же что и в SMP – **общая память**

Операционные системы

- Типы
  - Однозадачные
  - Система с пакетными заданиями (batch processing)
  - Многозадачные / с разделением времени (time-sharing)
    - **Кооперативная многозадачность** (cooperative multitasking)
    - **Вытесняющая многозадачность** (preemptive multitasking)
- История многозадачности
  - Изначально нужно было для раздела одной дорогой машины между разными пользователями
  - Теперь нужно для использования ресурсов одной многоядерной машины для множества задач

Основные понятия в современных ОС

- **Процесс** – владеет памятью и ресурсами
- **Поток** – контекст исполнения внутри процесса
  - В одном процессе может быть несколько потоков
  - Все потоки работают с общей памятью процесса
- Но в теории мы их будем смешивать
  - В научных работах исторически сложилось называть потоки исполнения «процессы» и обозначать большими буквами P, Q, R…

DEVEXPERTS

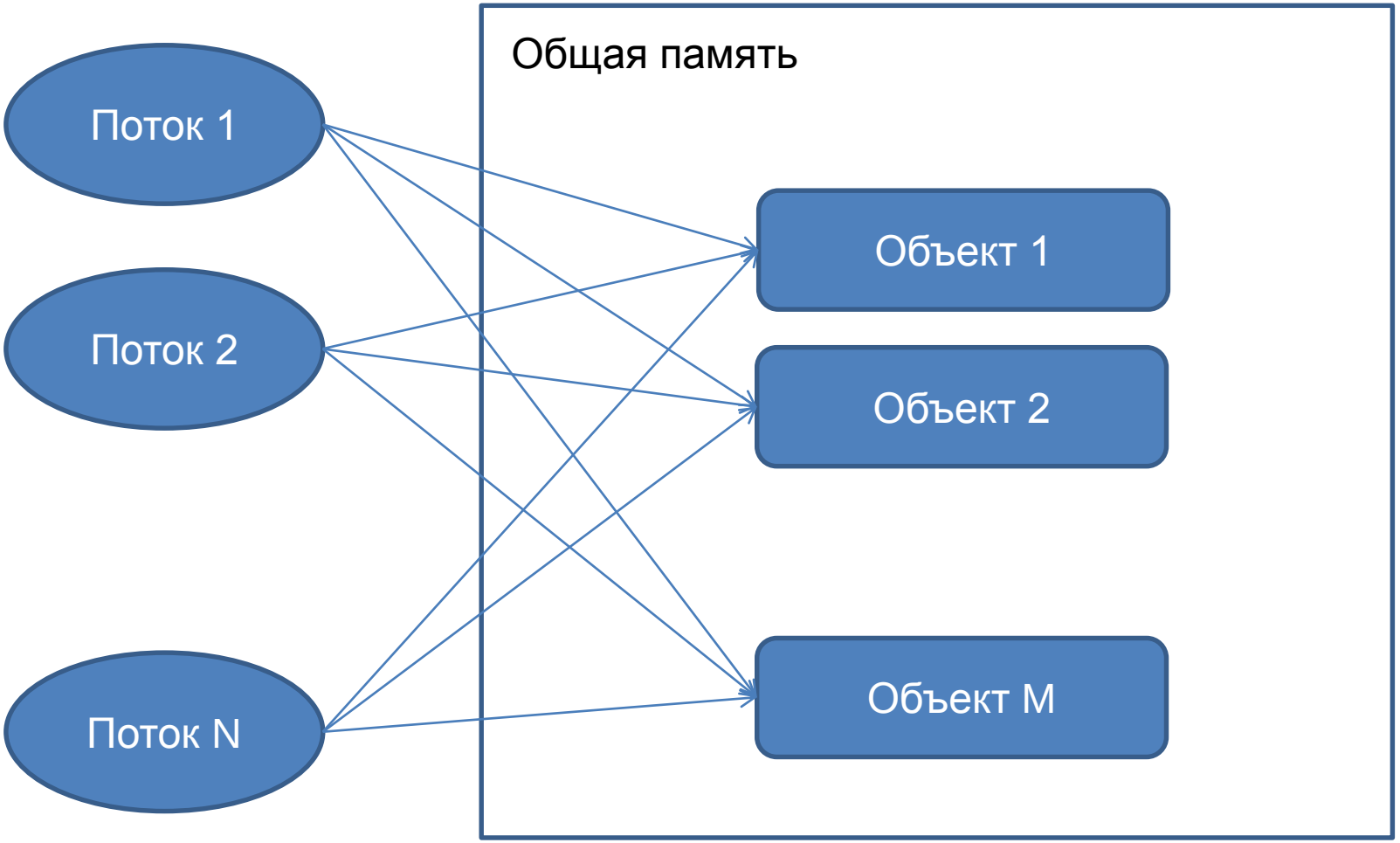
Нужна формальная модель параллельных вычислений

- 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**

Модели программирования

- «Классическое» однопоточное / однозадачное
  - Можем использовать ресурсы многоядерной системы только запустив множество разных, независимых задач
- Многозадачное программирование
  - Возможность использовать ресурсы многоядерной системы в рамках решения одной задачи
  - Варианты:
    - **Модель с общей памятью**
    - Модель с передачей сообщений (распределенное программирование)

Модель с общими объектами (общей памятью)



Общие объекты

- Потоки выполняют операции над общими, разделяемыми объектами
- В этой модели не важны операции внутри потоков:
  - Вычисления
  - Обновления регистров процессора
  - Обновления стека потока
  - Обновления любой другой локальной для потока памяти
- Важна только коммуникация между потоками
- В этой модели **единственный тип коммуникации между потоками – это работа с общими объектами**

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 многопоточных 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*

Свойства многопоточных программ

- Последовательные программы детерминированы
  - Если нет явного использования случайных чисел и другого общения с недетерминированным внешним миром
  - Их свойства можно установить, анализируя последовательное **исполнение** при данных входных параметрах
- **Многопоточные программы в общем случае недетерминированы**
  - Даже если код каждого потока детерминирован
  - Результат работы зависит от фактического **исполнения** при данных входных параметрах
    - А этих исполнений может быть много
  - Говорим «программа А имеет свойство P», если программа А имеет свойство P при любом допустимом исполнении

Как моделировать многопоточное выполнение?

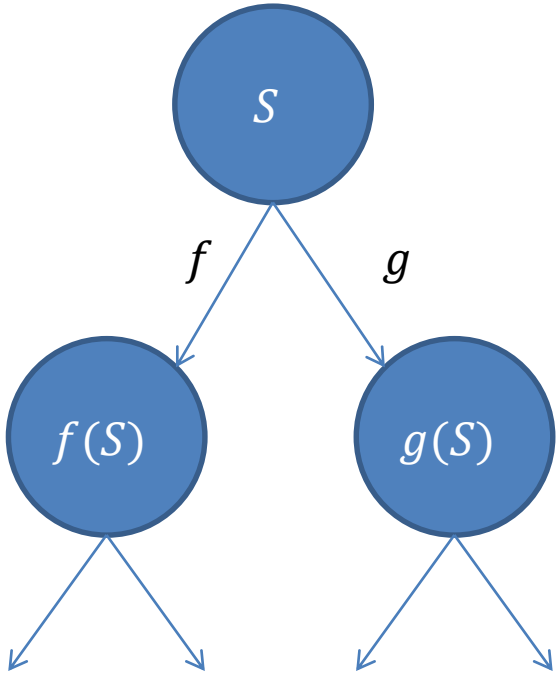
- **ПРИМЕР:** Очень простая программа. Всего два потока (P и Q), каждый из которых последовательно выполняет 2 действия и останавливается
  - У них есть общие переменные x и y (в начале равные 0)
  - Какие есть варианты r1 и r2 после исполнения?

```
shared int x
shared int y
```

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

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

Моделирование исполнений через чередование операций



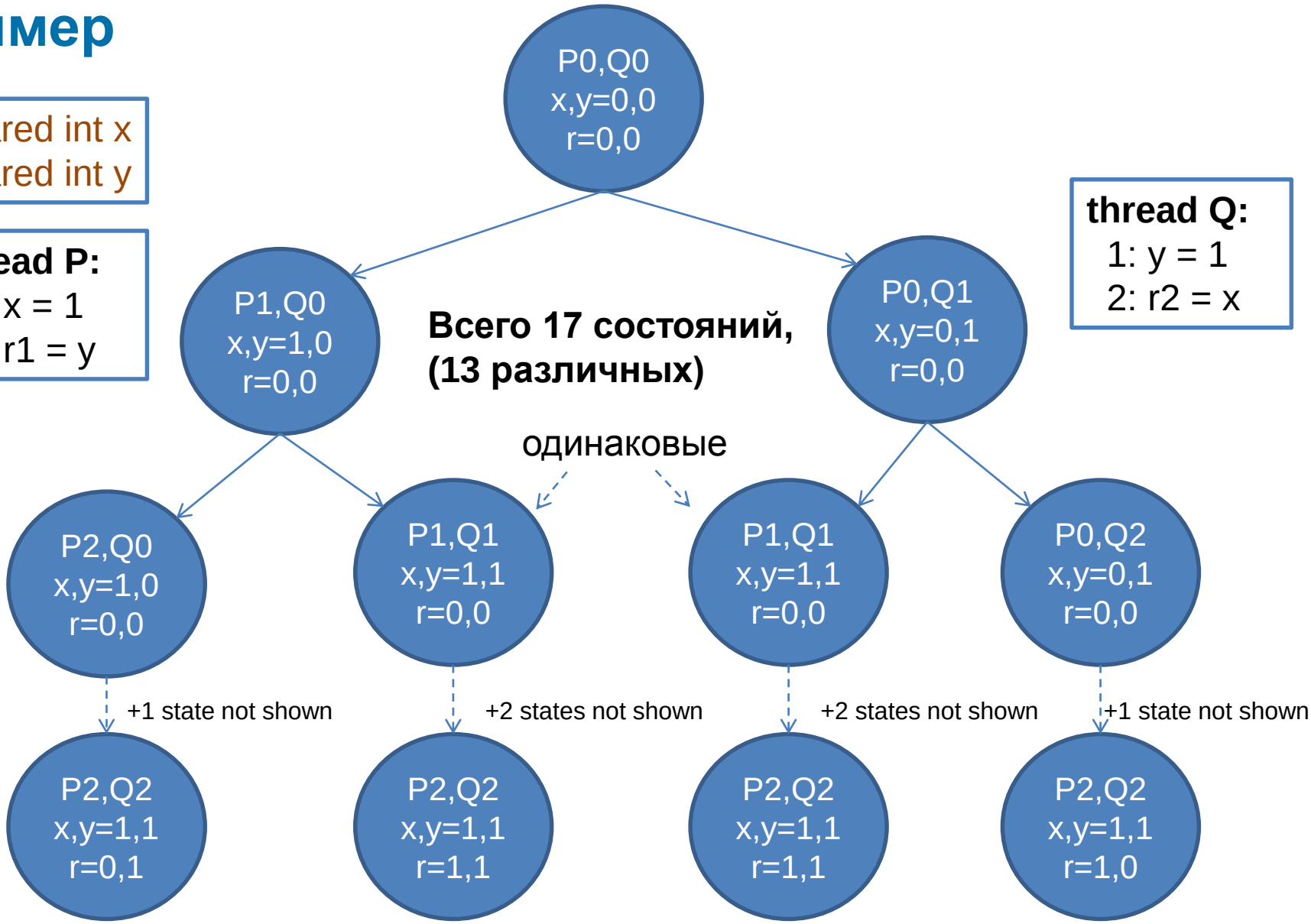
- S это **общее состояние**:
  - Состояние всех потоков
  - И состояние всех [общих] объектов
- f и g это **операции** над объектами
  - Для переменных это операции чтения (вместе с результатом) и записи (вместе с значением)
  - Количество активных операций равно количеству потоков
- $f(S)$ это новое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

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:
1: r1 = y
2: x = 1

thread Q:
1: r2 = x
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 (!)

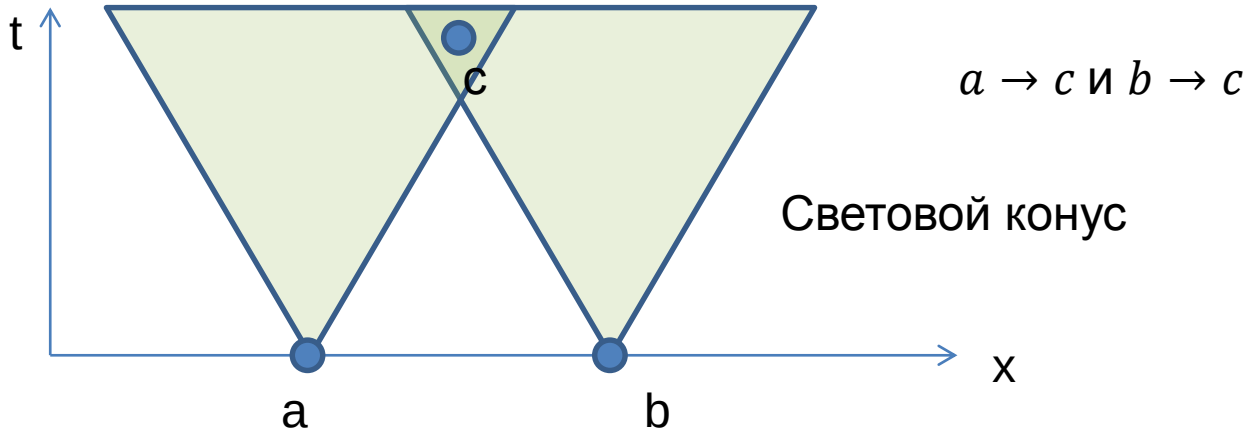
Модель чередования не отражает фактическую реальность. Когда же её можно использовать?

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 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» (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**

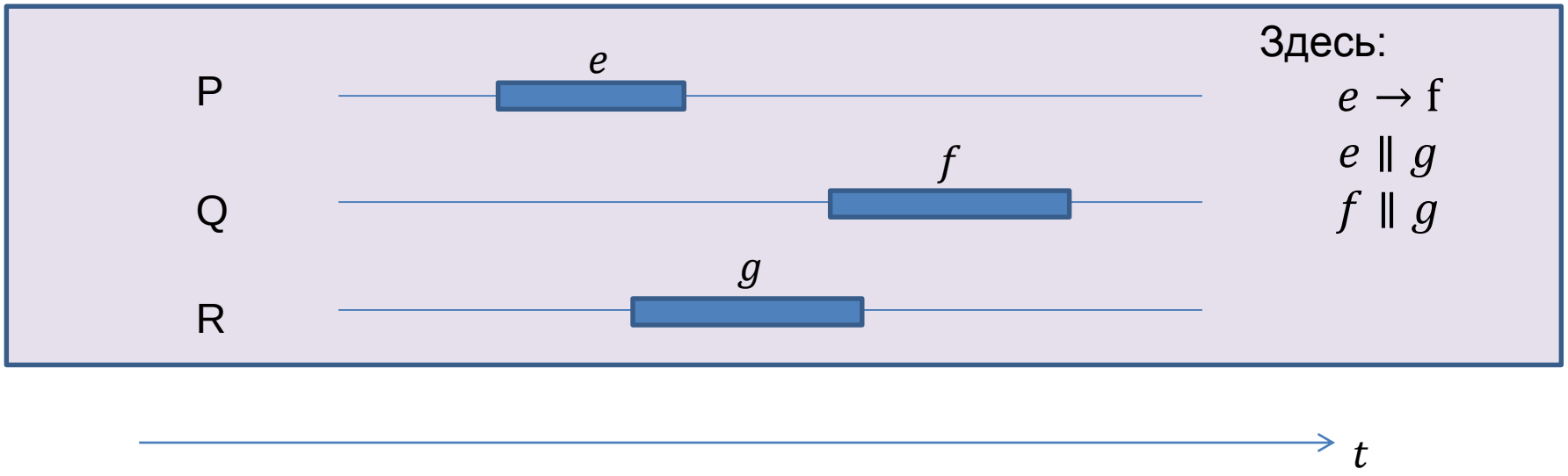
Модель «произошло до»

- 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

Конфликты (data race)

- Если две операции над переменной произошли **параллельно** (нет отношения произошло до) и одна из них это **запись**, то такая ситуация называется **конфликтом** (так же известным как **гонка данных**, data race).
  - Это свойство конкретного **исполнения**
- Программа, в любом допустимом исполнении которой (с точки зрения модели памяти) нет конфликтов, называется **корректно синхронизированной**
  - Модель памяти C++11 описывает поведение только корректно синхронизированных программ
  - Модель памяти Java 5 дает определенные (слабые и сложные для понимания) гарантии и программам с конфликтами

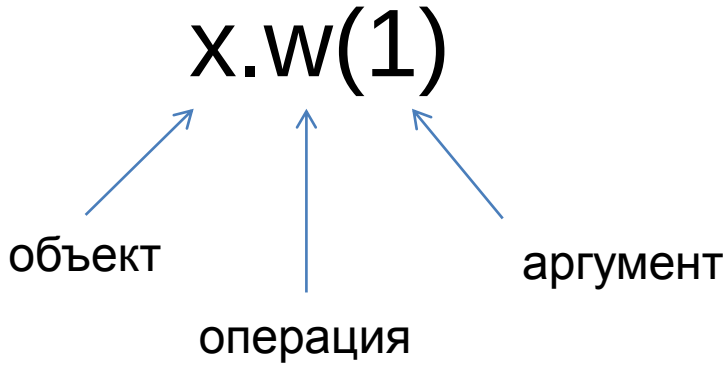
DEVEXPERTS

Свойства исполнений над общими объектами

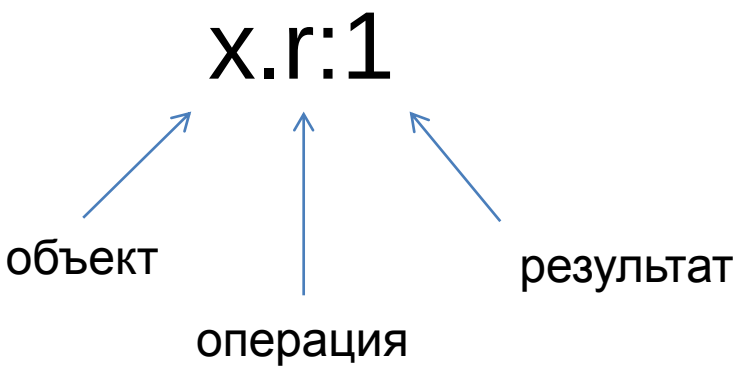
http://blog.lib.umn.edu/cspg/electionacademy/images/sharing.jpg

Операции над общими объектами

Запись (write)
общей переменной

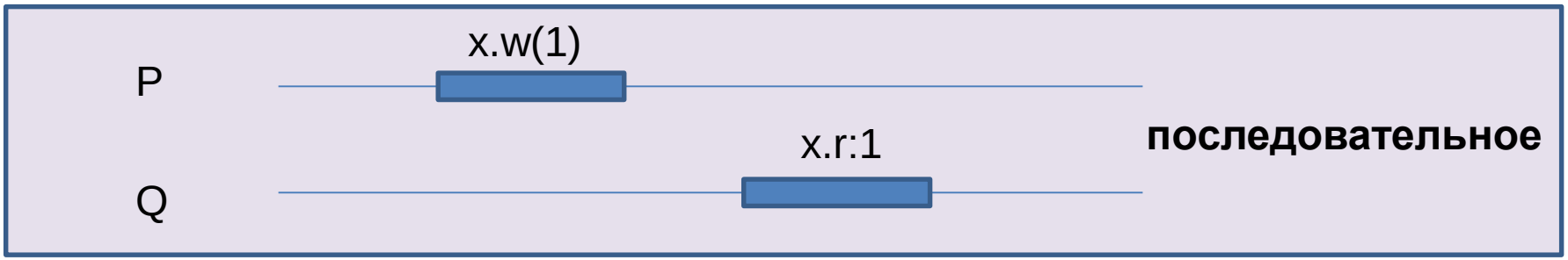


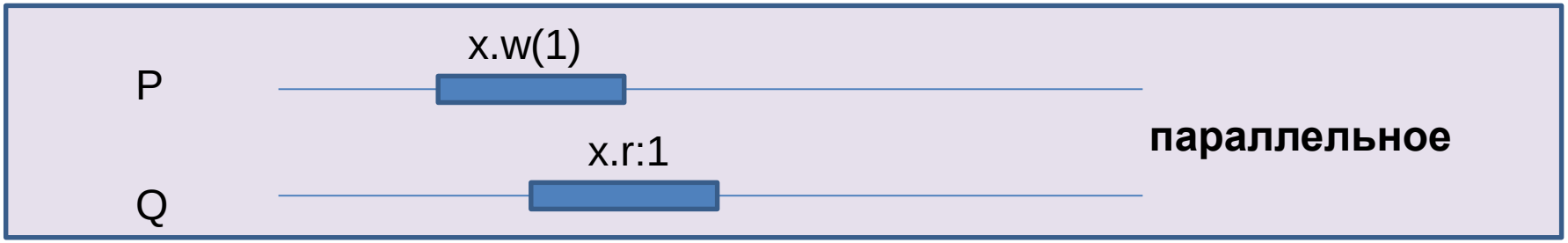
Чтение (read)
общей переменной



Последовательное исполнение

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



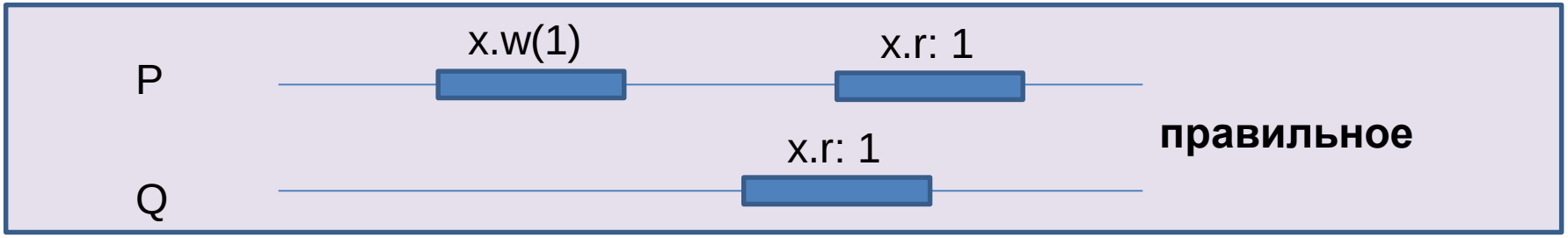


Последовательное исполнение: формально

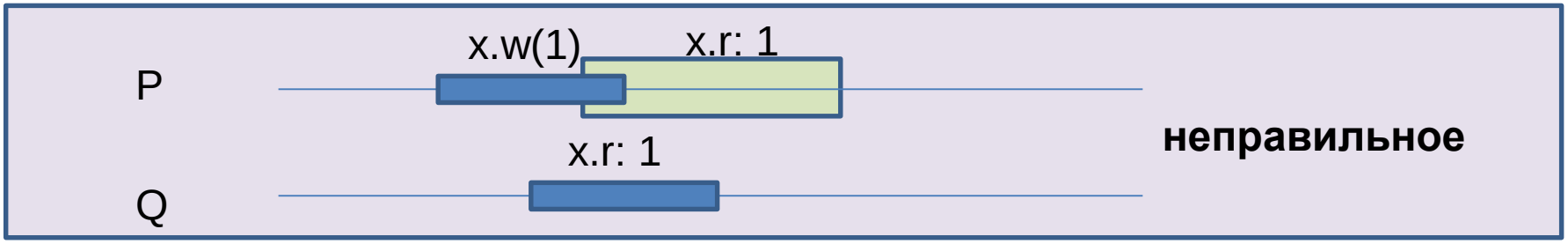
- Исполнение системы называется **последовательным**, если все **операции** линейно-упорядочены отношением «произошло до», то есть «произошло до» является полным строгим порядком на операциях
  - **Следствие1:** все **события** упорядочены отношением «произошло до» и более того,
    $inv(A_1) \rightarrow resp(A_1) \rightarrow inv(A_2) \rightarrow resp(A_2) \rightarrow \ldots$
  - **Следствие2**: В последовательной истории каждой **операции** можно сопоставить глобальное время так, что время разных операций различно и
    $$A \rightarrow B \Leftrightarrow t(A) < t(B)$$

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



Нас интересуют только правильные исполнения



Правильное исполнение: формально

- $H|_P$ – сужение исполнения на поток P. Все события a, такие что proc(a) = P
  - Соответственно и для всех операций будет proc(A) = P
  - Исполнение называется **правильным (well-formed),** если его сужение на любой поток P является последовательным.
    - Задается программой которую выполняет поток
  - Будем далее работать только с правильными исполнениями.
- $H|_x$ – сужение истории на объект x. Все события a, такие что obj(a) = x
  - Соответственно и для всех операций будет obj(A) = x
  - В правильном исполнении сужение на объекты не обязательно является последовательным(!)

Правильные исполнения и C++

- В C++ история с правильными исполнения несколько сложней, ибо сам язык не накладывает порядка исполнения операций между точками следования (*sequence points*):

```
x = f(i++) + g(i++);
// has undefined behavior in C++
```

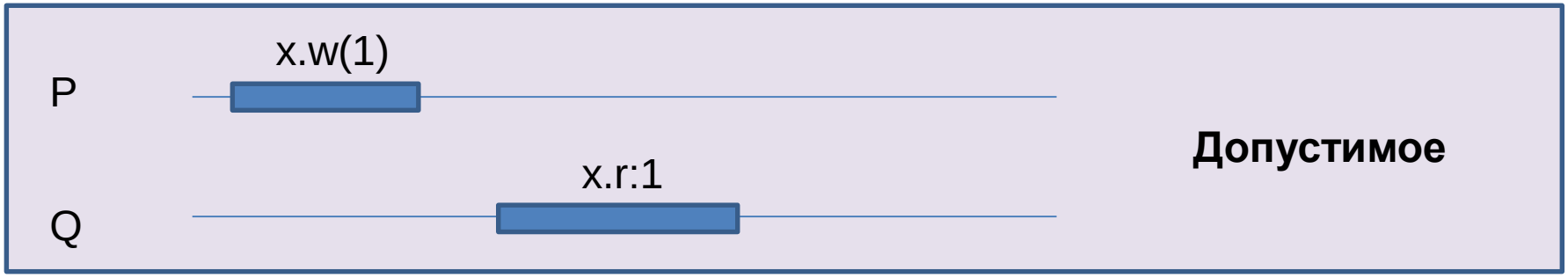
- На это можно посмотреть так, что внутри одного потока в C++ могут быть параллельные операции (здесь две операции i++)

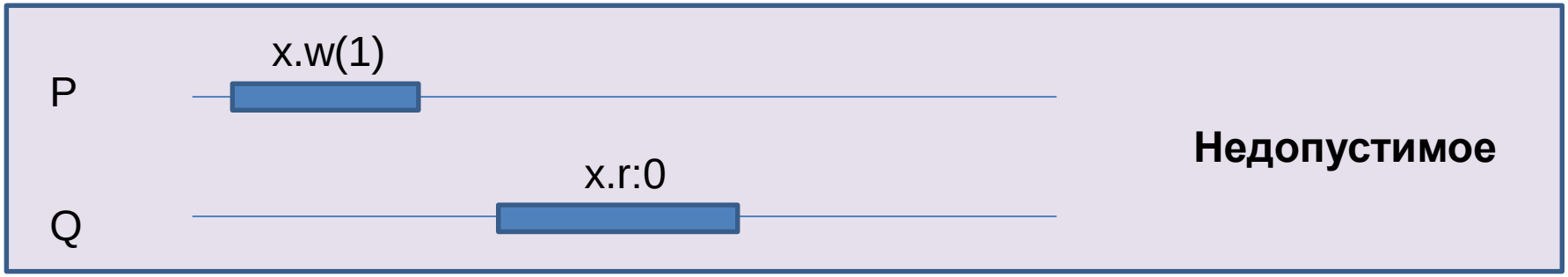
Последовательная спецификация объекта

- $H|_x$ – 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)

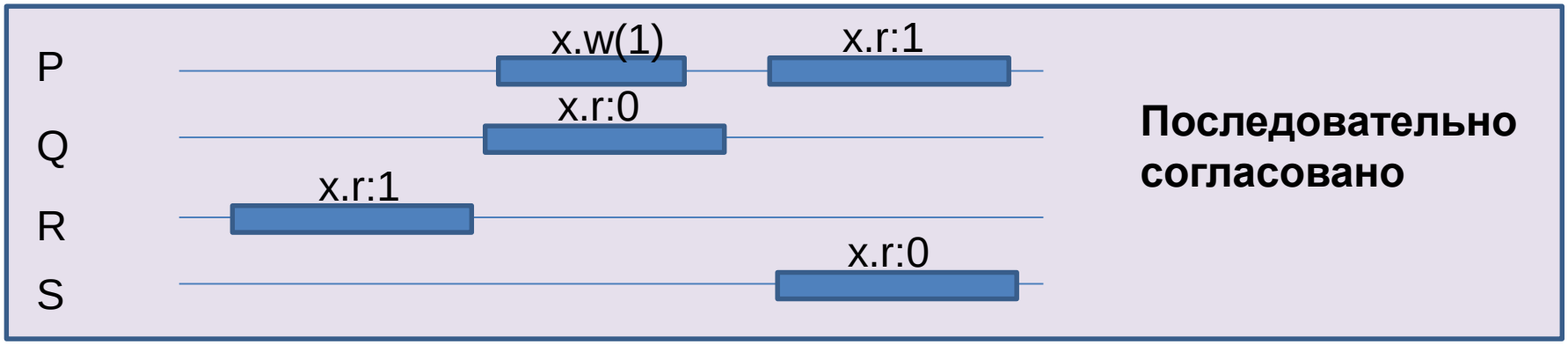
- Как определить допустимость **параллельного** исполнения?
  - Сопоставив ему **эквивалентное** (состоящее из тех же событий и операций) **допустимое последовательное исполнение**
  - Как именно – тут есть варианты: **условия согласованности**
- Согласованность это аналог «корректности» в мире многопоточного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

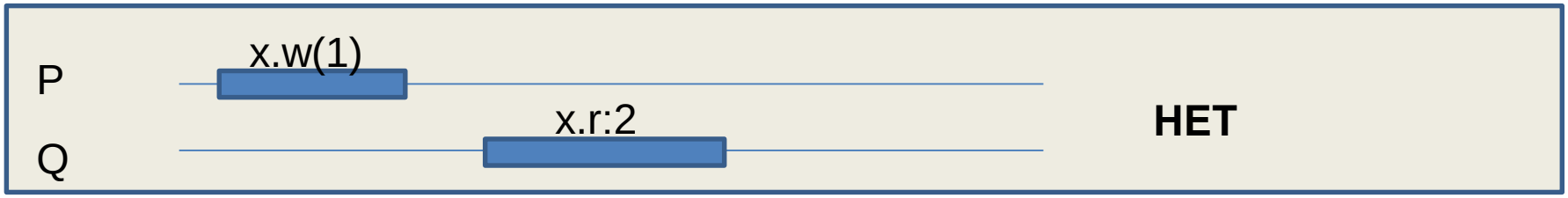
Базовое требование: корректные последовательные программы должны работать корректно в одном потоке

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

Последовательная согласованность (1)

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





Последовательная согласованность (2)

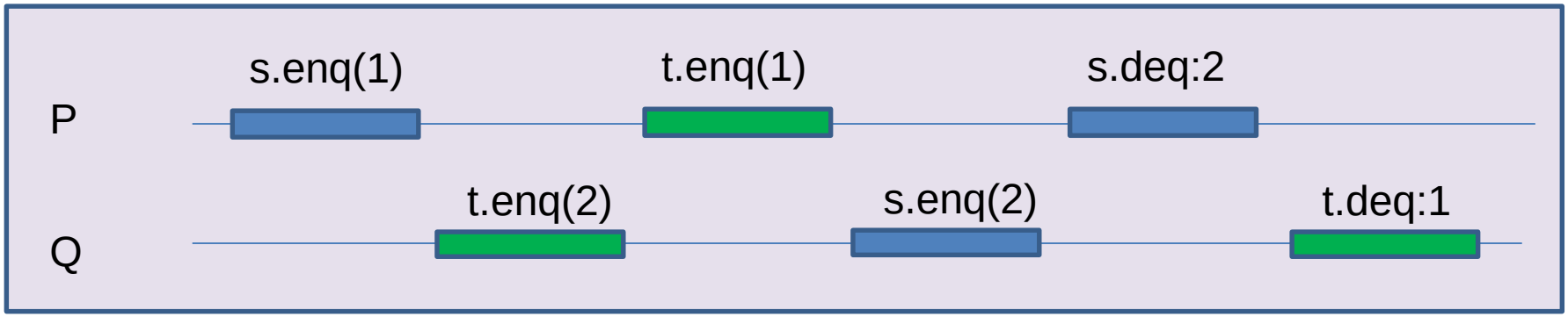
- Используется как условие корректности исполнения **всей системы в целом**
  - когда нет никакого «внешнего» наблюдателя, который может «увидеть» фактический порядок между операциями на разных процессах
- **Модель памяти** языка программирования и системы исполнения кода используют последовательную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

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

Последовательная согласованность (3)

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

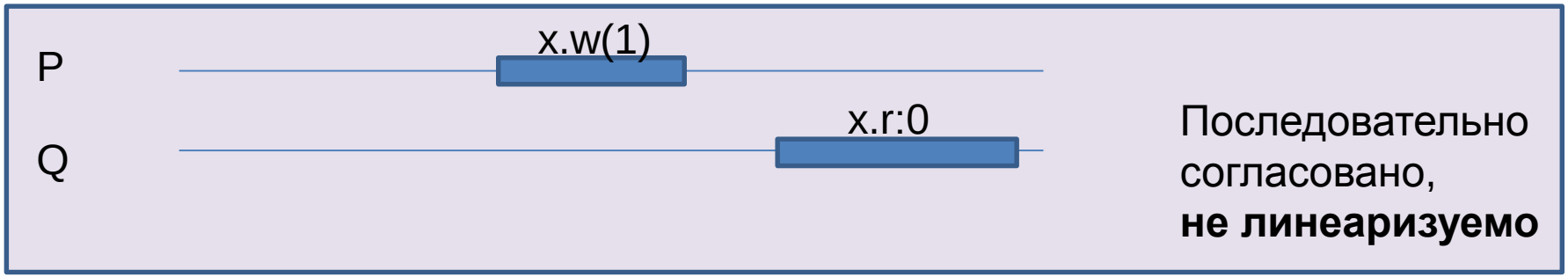
ПРИМЕР: (здесь s и t это две FIFO очереди)

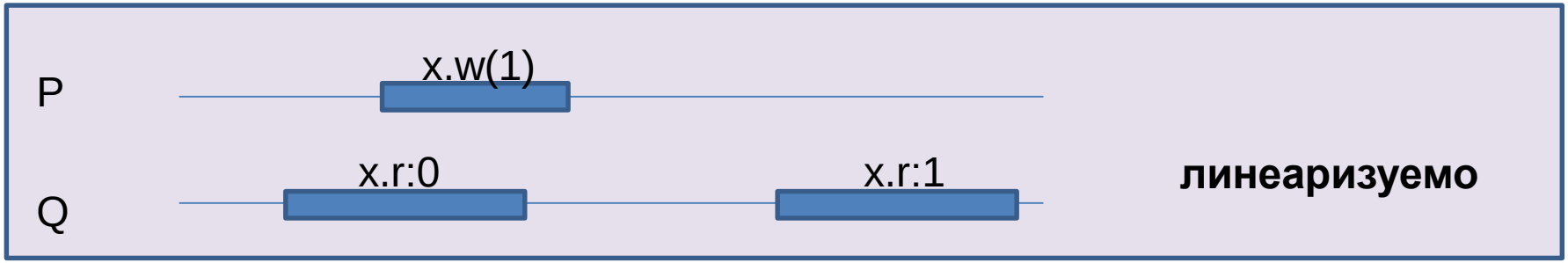


Последовательную согласованность нельзя использовать для спецификации поведения отдельных объектов в системе

Линеаризуемость

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





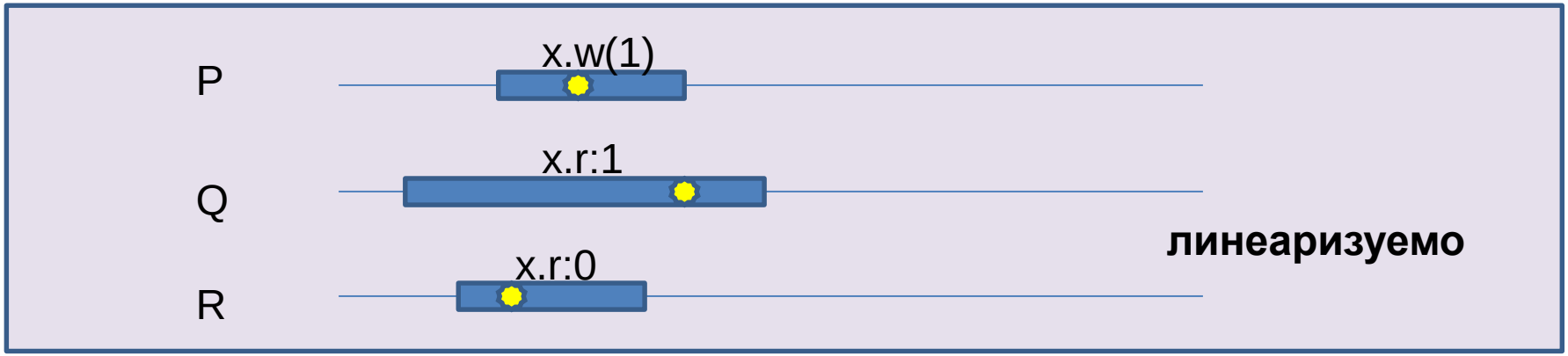
Свойства линеаризуемости

- 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

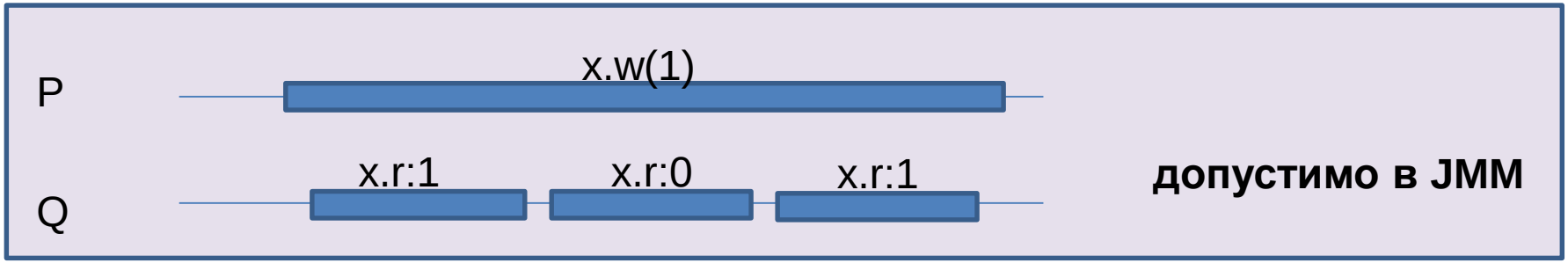
- **Из более простых линеаризуемых объектов можно делать линеаризуемые объекты более высокого уровня**

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 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 (2)

- 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 для получения аналогичных гарантий

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?

```java
@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;
    }
}
```

```java
@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

Java Memory Model

- Модель памяти Java гарантирует, что SampleTest2 может получить только эти, ожидаемые нами, исходы, при выполнении на любой архитектуре
  - И это позволяет писать корректные для любой архитектуры программы
- Но даже на x86 volatile не бесплатен. Для достижения максимальной производительности надо их использовать в меру. Что тогда будет?
  - JMM дает ответ и на этот вопрос

Лекция 2

ПОСТРОЕНИЕ АТОМАРНЫХ ОБЪЕКТОВ: БЛОКИРОВКИ

Построение линеаризуемых объектов

- **Как** построить линеаризуемый объект из последовательного?
  - *Если у нас есть атомарные регистры как аппаратные примитивы*

```
class Queue:
    int head
    int tail
    E elems[N]

    def enq(x):
      elems[tail] = x
      tail = (tail + 1) % N

    def deq:
      result = elems[head]
      head = (head + 1) % N
      return result
```

Не безопасно использовать из разных потоков (not thread-safe)

Это псевдо-код, где каждая отдельная операция **атомарна**

Блокировки (Locks)

http://www.flickr.com/photos/xserve/368758286/

Взаимное исключение

- Защитим каждую операцию специальным объектом mutex (mutual exclusion), также известного как **блокировка** (lock)

```
class Queue:
    Mutex mutex
    …

    def enq(x):
      mutex.lock
      …
      mutex.unlock

    def deq:
      mutex.lock
      …
      mutex.unlock
      return result
```

Так чтобы реализация (тело) каждого метода не выполнялось **одновременно**, то есть все тела методов выполнялись бы последовательно.

Значит будет работать обычный последовательный код

Взаимное исключение формально

Протокол

```
thread P_id:
    loop forever:
        1: nonCriticalSection
        2: mutex.lock
        3: criticalSection
        4: mutex.unlock
```

- Главное свойство называется **взаимное исключение**.
- X1: **Критические секции** не могут выполняться параллельно:

$$\forall i, j : i \neq j \Rightarrow CS_i \rightarrow CS_j \vee CS_j \rightarrow CS_i$$

- Это значит, что выполнение критических секций будет линеаризуемо
- Это требование **корректности** протокола взаимной блокировки

Взаимное исключение, попытка 1

```
shared boolean  want

def lock:
    1: while want:
    2:     pass // wait
    3: want = true

def unlock:
    4: want = false
```

- Этот протокол **не гарантирует взаимное исключение**
  - Оба потока могут оказаться в критической секции **одновременно**

Взаимное исключение, попытка 2

```
threadlocal int     id  // 0 or 1
shared boolean  want[2]

def lock:
    1: want[id] = true
    2: while want[1 - id]:
    3:     pass

def unlock:
    4: want[id] = false
```

- Этот протокол гарантирует **взаимное исключение**
- Но не нет никакой гарантии **прогресса**.
- X2: **Отсутствие взаимной блокировки** (deadlock-freedom). Если несколько потоков пытаются войти в критическую секцию, то хотя бы один из них должен войти в критическую секцию за конечное время. *

* При условии что критические секции выполняются за конечное время

Взаимное исключение, попытка 3

```
threadlocal int     id  // 0 or 1
shared int          victim

def lock:
    1: victim = id
    2: while victim == id:
    3:     pass

def unlock:
    4: pass
```

- Есть **взаимное исключение** и **прогресс**
- Но можем заходить только по очереди. Поток будет **голодать**.
- X3: Потребуем **отсутствие голодания** (starvation-freedom). Если какой-то поток пытается войти в критическую секцию, то он войдет в критическую секцию за конечное время. *

* При условии что критические секции выполняются за конечное время

Взаимное исключение, алгоритм Петерсона

```
threadlocal int     id  // 0 or 1
shared boolean want[2]
shared int          victim

def lock:
    1: want[id] = true
    2: victim = id
    3: while want[1-id] and
    4:         victim == id:
    5:     pass

def unlock:
    6: want[id] = false
```

- Гарантирует **взаимное исключение**, **отсутствие взаимной блокировки** и **отсутствие голодания**.
- Порядок операций присваивания (1 и 2) в этом коде важен
- Не первый изобретенный (1981), но простейший алгоритм для двух потоков.

Взаимное исключение, алгоритм Петерсона для N потоков

```
threadlocal int     id  // 0 to N-1
shared int          level[N]
shared int          victim[N]

def lock:
    1: for j = 1..N-1:
    2:     level[id] = j
    3:     victim[j] = id
    4:     while exist k: k != id and
    5:                level[k] >= j and
    6:                victim[j] == id:
    7:         pass

def unlock:
    8: level[id] = 0
```

- Гарантирует **взаимное исключение**, **отсутствие блокировки** и **отсутствие голодания**.
- Но не очень **честный**.
  - Невезучий поток может ждать пока другие потоки $O(N^2)$ раз войдут в критическую секцию (квадратичное ожидание)
  - Более честно было бы **линейное ожидание**

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 1)

```
threadlocal int     id  // 0 to N-1
shared boolean want[N]       init false
shared int       label[N]    init 0

def lock:
    want[id] = true
    label[id] = max(label) + 1          } doorway
    while exists k: k != id and
              want[k] and
              (label[k], k) < (label[id], id) :
          pass

def unlock:
    want[id] = false
```

- Есть корректность и все свойства прогресса
- Обладает свойством **первый пришел, первый обслужен** (first-come, first-served – FCFS)
  - Это сильнее чем **линейное ожидание**
- Но метки должны быть бесконечными (их можно заменить на конечные метки)

Взаимное исключение, алгоритм Лампорта (алгоритм булочника - вариант 2)

```
threadlocal int     id  // 0 to N-1
shared boolean choosing[N] init false
shared int         label[N]       init inf

def lock:
    choosing[id] = true
    label[id] = max(label) + 1      } doorway
    choosing[id] = false
    while exists k: k != id and
            (choosing[k] or
            (label[k], k) < (label[id], id)) :
        pass

def unlock:
    label[id] = inf
```

- Те же свойства
  - И метки тоже могут быть бесконечными, хотя мы их и сбрасываем при выходе из критической секции

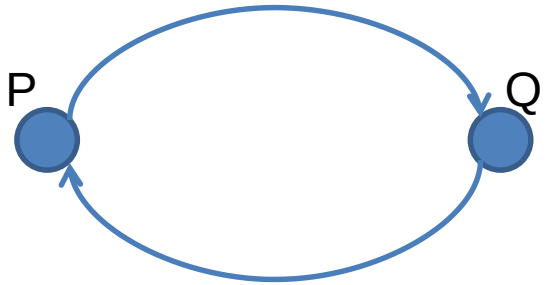
Использование взаимного исключения (блокировки)

- Любой последовательный объект можно сделать параллельным, **линеаризуемым**.
  - Это т.н. **грубая блокировка**. Блокируем всю операцию целиком.
  - Можно использовать **тонкую блокировку**.
    - Блокировать только операции над общими объектами внутри метода, но не вызов метода целиком.
    - Но тогда, для обеспечения линеаризуемости, нужно использовать **двухфазную блокировку**.

Проблемы взаимного исключения: взаимная блокировка (deadlock)

thread P:
1: mutex1.lock
2: mutex2.lock
….
3: mutex2.unlock
4: mutex1.unlock

thread Q:
1: mutex2.lock
2: mutex1.lock
….
3: mutex1.unlock
4: mutex2.unlock



Цикл в **графе ожидания** приводит к **взаимной блокировке**

Избежать появления такого цикла можно с помощью **иерархической блокировки** (глобально упорядочив все блокировки)

Проблемы взаимного исключения

- **Условные условия прогресса**
  - Отсутствие взаимной блокировки и отсутствие голодания гарантируют прогресс только если критические секции выполняются за конечное время
- **Инверсия приоритетов**
  - Возникает при блокировке потоков с разным приоритетом на одном объекте
- **Последовательное выполнение операций**
  - Нет параллелизма при выполнении критической секции, а значит ограничена **вертикальная масштабируемость**.

Закон Амдала для параллельной работы

- Максимальное убыстрение при запуске кода в N потоков если доля S выполняется последовательно

$$speedup = \frac{1}{S + \frac{1-S}{N}}$$

$$\lim_{N \to \infty} speedup = \frac{1}{S}$$

- Даже при 5% последовательного кода (S=0.05), максимальное убыстрение равно 20.

Лекция 3

АЛГОРИТМЫ БЕЗ БЛОКИРОВОК: РЕГИСТРЫ

Алгоритмы без блокировок (lock-free)

Безусловные условия прогресса

- **Отсутствие помех** (obstruction-freedom)
  - Если несколько потоков пытаются выполнить операцию, то любой из них должен выполнить её за конечное время, если все другие потоки остановить в любом месте (чтобы не мешали)
- **Отсутствие блокировки** (lock-freedom)
  - Если несколько потоков пытаются выполнить операцию, то хотя бы один из них должен выполнить её за конечное время (не зависимо от действия или бездействия других потоков)
- **Отсутствие ожидания** (wait-freedom)
  - Если какой-то поток пытается выполнить операцию, то он выполнит её за конечное время (не зависимо от действия или бездействия других потоков)

Объекты без блокировки

- Используя **блокировку** (lock) мы не можем получить объект **без блокировки** (lock-free) и даже **без помех** (obstruction-free)
  - Если поток пытаются выполнить операцию, когда какой-то другой поток находится внутри критической секции и ничего не делает (не получает время от операционной системы), то ничего не получается. Он ждет неограниченное время (освобождения блокировки), независимо от того, сколько процессорного времени ему будет выделено

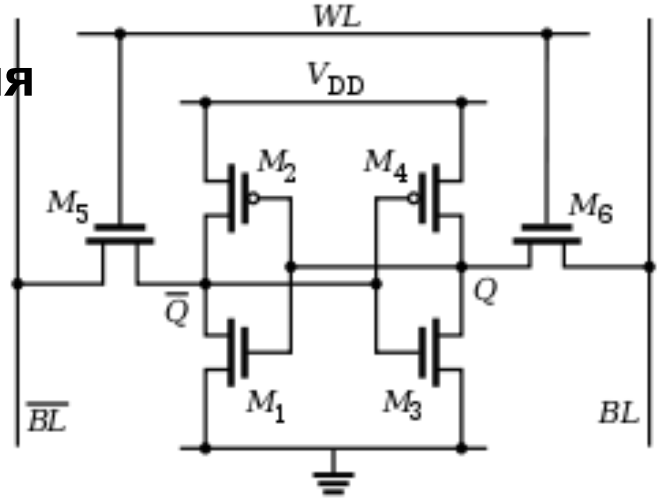
Регистры без блокировки

- Общие регистры – базовый объект для общения потоков между собой

```
// Последовательная спецификация
class Register:
    int r

    def write(x):
        r = x

    def read():
        return r
```

Физические регистры

- Физические регистры **неатомарны**
  - Но работают **без ожидания**
  - **Булевы** (хранят только один бит)
  - Поддерживают только **одного читателя** и **одного писателя**
  - Попытка чтения и записи **одновременно** приводит к непредсказуемым результатам
  - Но они **безопасны** (safe)
    - После завершение записи, будет прочитано последнее записанное значение

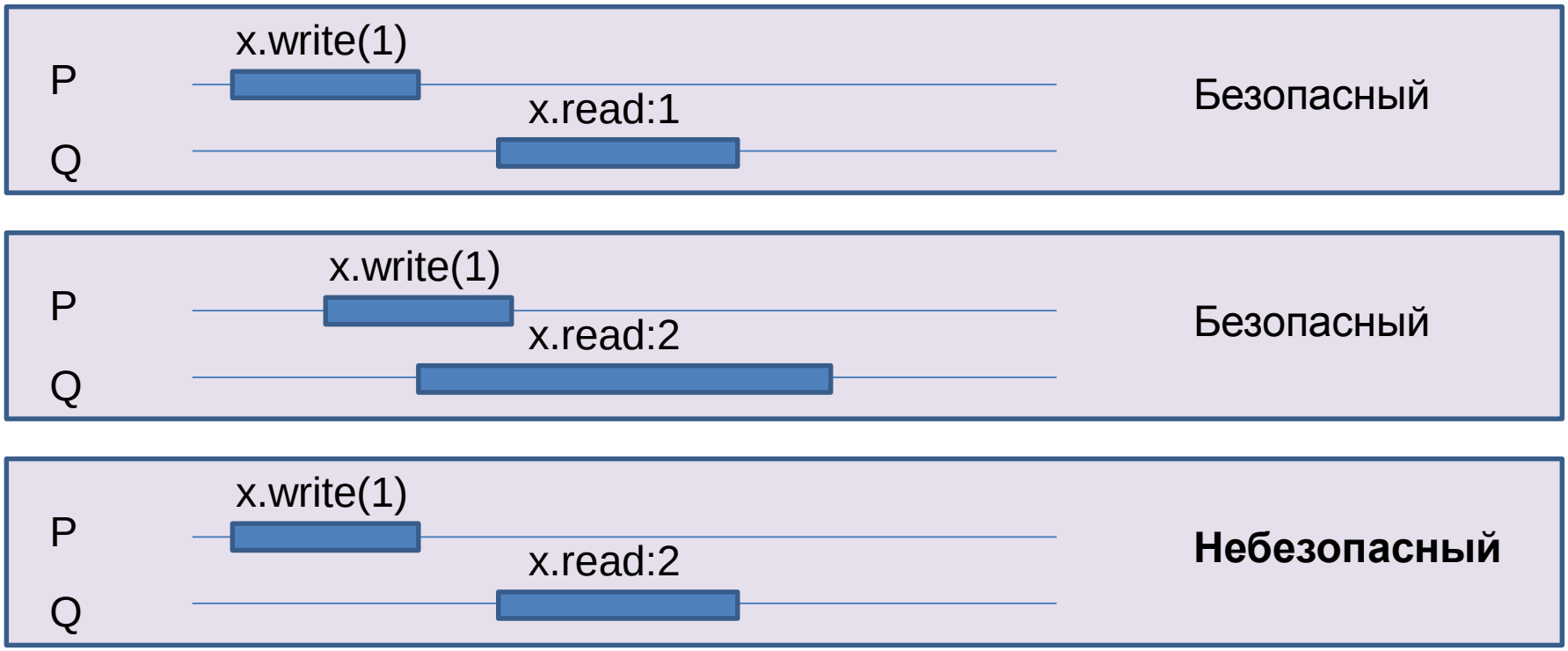


Классификация регистров

- По условиям согласованности или корректности
  - Безопасные (safe), регулярные (regular), атомарные (atomic)
- По количеству потоков
  - Один читатель, много читателей (SR, MR)
  - Один писатель, много писателей (SW, MW)
- По количеству значений
  - Булевские значение (boolean), множественные значения (M-valued)
- Иерархия типов регистров
  - Самый примитивный регистр – Safe SRSW Boolean register
  - Самый сложный регистр – Atomic MRMW M-Valued register

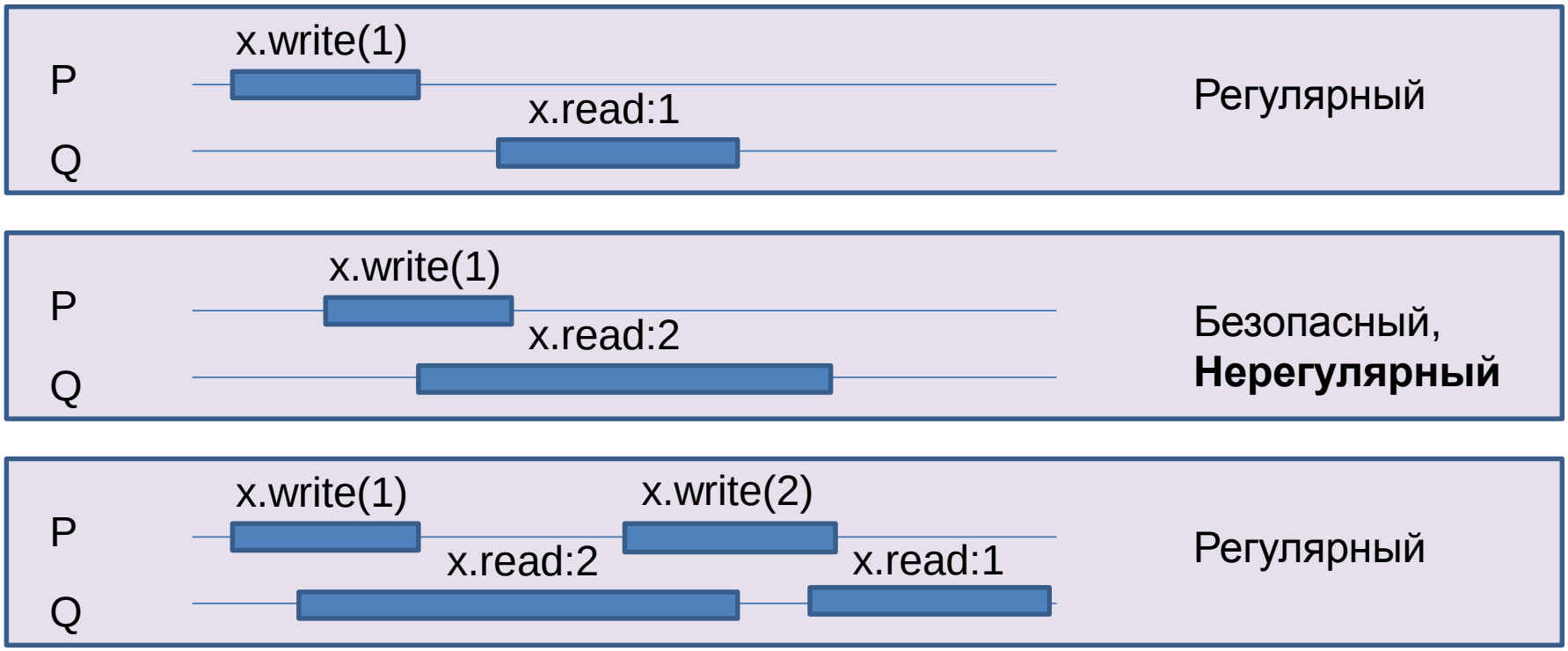
Безопасные (safe) регистры

- Гарантирует получение последнего записанного значения, если операция чтения не параллельна операциям записи



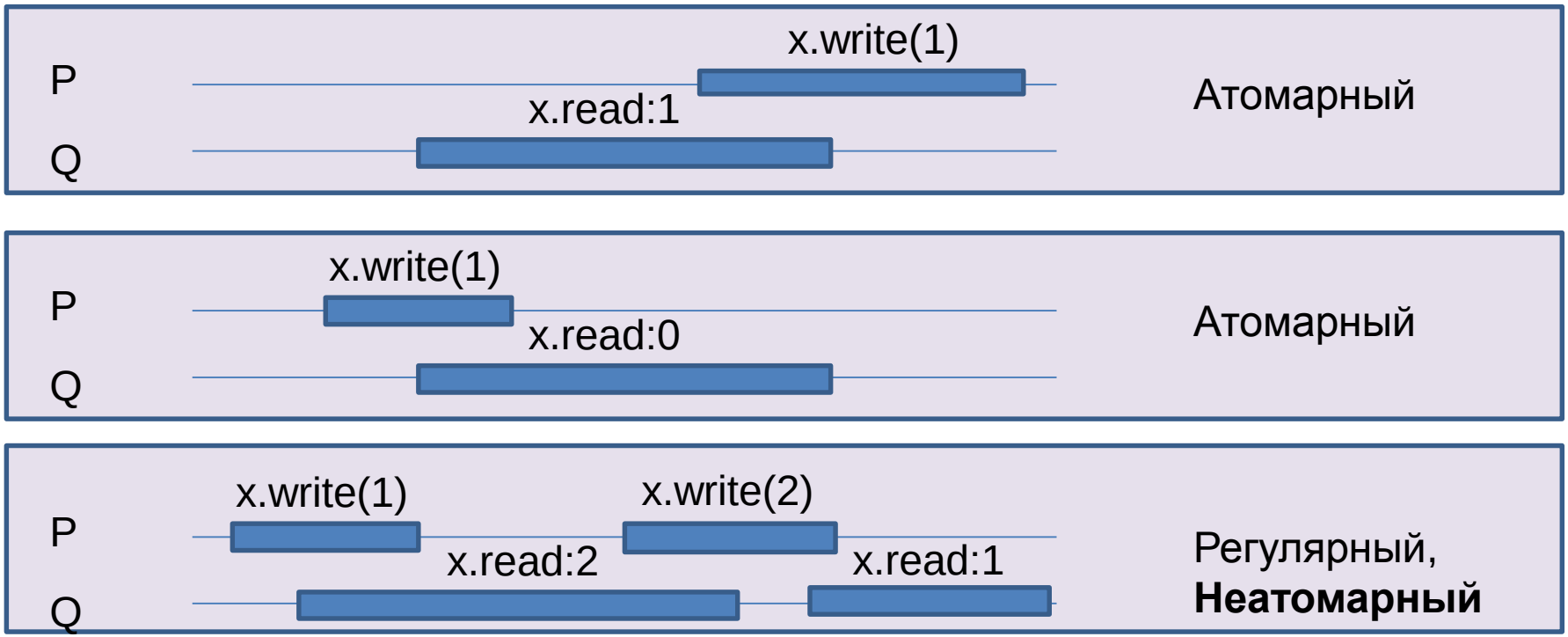
Регулярные (regular) регистры

- При чтении выдает либо последнее записанное, либо одно из тех значений, который сейчас пишутся



Атомарные (atomic) регистры

- Исполнение линеаризуемо



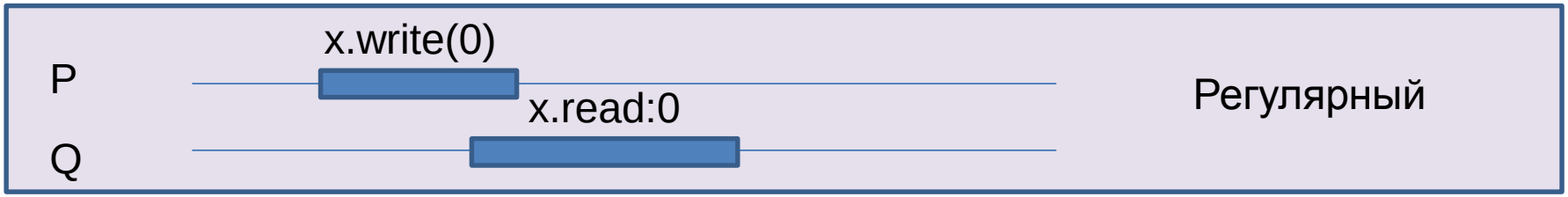
Построение регистров

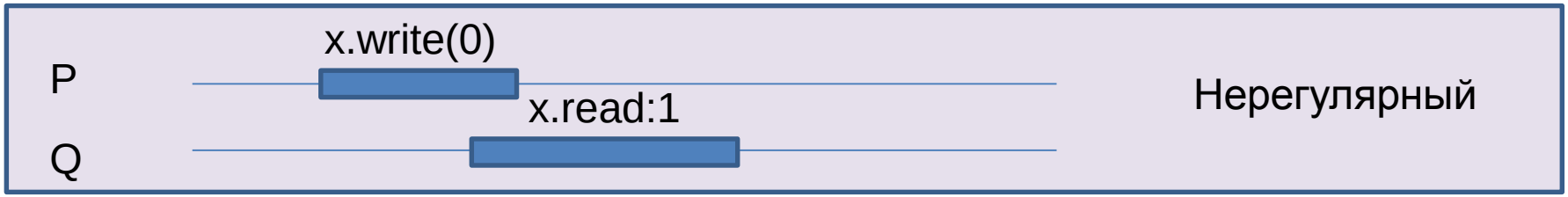
- Будем строить более сложные регистры из более простых требуя, чтобы реализация была **без ожидания** (wait-free образом).
  - **Safe SRSW Boolean register** – дан в начале
  - Regular SRSW Boolean register
  - Regular SRSW M-Valued register
  - Atomic SRSW M-Valued register
  - Atomic MRSW M-Valued register
  - **Atomic MRMW M-Valued register** – хотим построить

Регулярный SRSW булев регистр

```
safe shared boolean r
threadlocal  boolean last

def write(x):
    if x != last:
        last = x
        r = x

def read(): return r
```

- У булева регистра только два значения – 0 и 1.
- Запоминаем последнее записанное значение и не перезаписываем его





Регулярный SRSW регистр с множеством значений

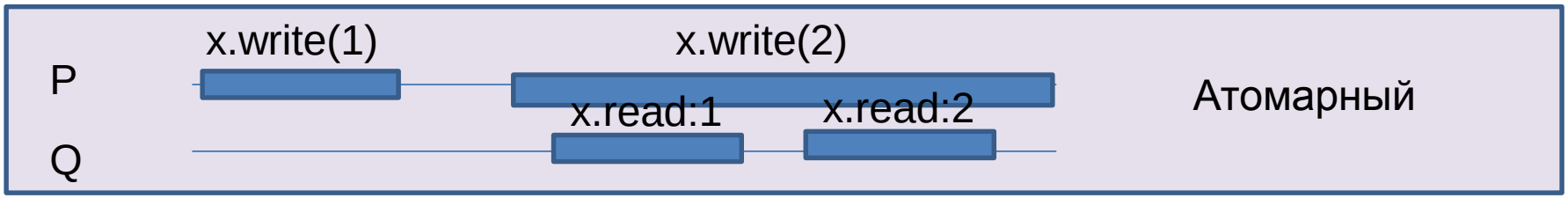
```
regular shared boolean[M] r

def write(x): // Справа-на-лево
    r[x] = 0
    for i = x-1 downto 0: r[i] = 1

def read(): // Слева-на-право
    for i = 0 to M-1:
        if r[i] == 0:
            return i
```

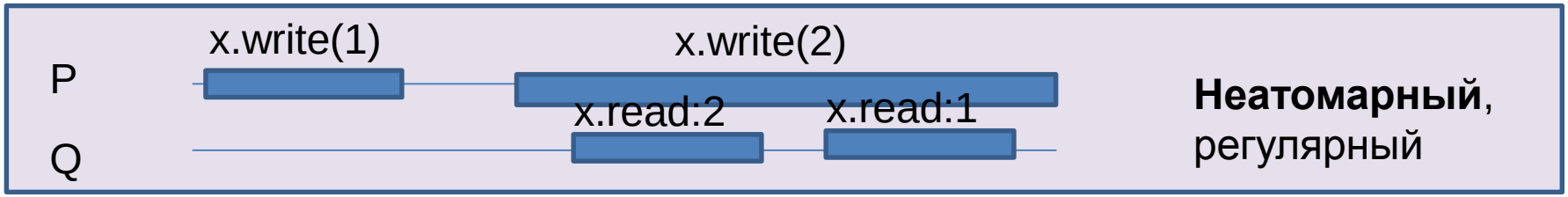
- Запоминаем M значение в унарном коде используя M регистров
  - Индекс первого нуля определяет значение
- Чтение и запись происходят в разном порядке
- Результирующий регистр регулярен

Индекс i	**0**	**1**	**2**	**3**	
r[i]	1	1	1	0	Значение 3
r[i]	1	0	0	0	Значение 1

Атомарный SRSW регистр с версиями

- Атомарный регистр не может «возвращаться назад во времени» если несколько чтений перекрываются с одной записью
  - Идея: Отследим время через версию значения





Атомарный SRSW регистр с версиями

```
safe shared (int x, int v) r

threadlocal (int x, int v) lastRead
threadlocal  int lastWriteV

def write(x):
    lastWriteV++
    r = (x, lastWriteV)

def read():
    cur = r
    if cur.v > lastRead.v:
        lastRead = cur
    return lastRead.x
```

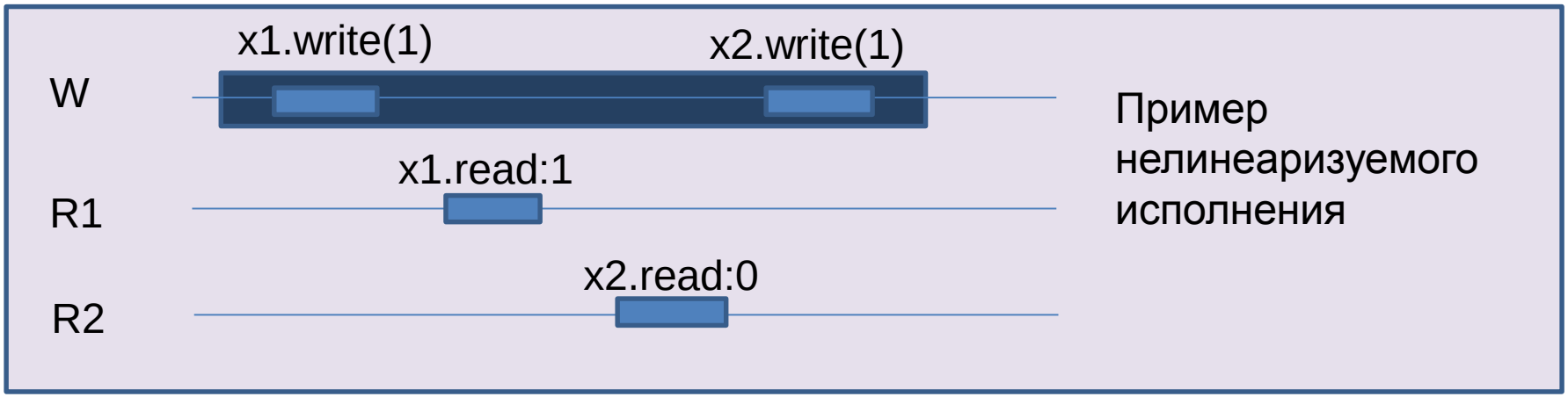
- На практике это отличное решение, ибо размер версии можно разумно ограничить практическими соображениями

Атомарный регистр: проблемы

- **Версии**
  - Может хранить пару (версия, значение) в регулярном регистре
  - Но версии растут неограниченно
- **Блокировки**
  - Алгоритм Лампорта будет работать на регулярных регистрах
  - Но это не дает нам алгоритм **без ожидания**
- **Теорема:**
  - Не существует алгоритма построения атомарного регистра без ожидания, которые использует конечное число регулярных регистров конечного размера так, чтобы их писал только писатель, а читал только читатель
    - Нужна обратная связь от читателя к писателю!

Атомарный MRSW регистр

- Нужна поддержка N **читателей**
- Заводим по SRSW регистру для каждого читателя и пишем в каждый из них.
  - Не получается линеаризуемый (атомарный) алгоритм



Атомарный MRSW регистр с версиями

- Отслеживаем версию записанного значения храня пару (x,v) в каждом из N регистров в которые пишет писатель
- Заводим N*(N-1) регистров для общения между читателями
  - Каждый читатель выбирает более позднее значение из записанного писателем и из прочитанных значений других читателей.
  - После этого читатель записывает свое прочитанное значение и версию для всех остальных читателей.

Атомарный MRMW регистр с версиями

- Нужна поддержка N **писателей**
- Отслеживаем версию записанного значения
  - Каждый читатель выбирает более позднюю версию
  - Для проставления версий писателями используем doorway секцию из алгоритма булочника (алгоритма взаимного исключения Лампорта)
    - Версия будет состоять из пары номера потока писателя и собственно числа.

Атомарный снимок состояния N регистров

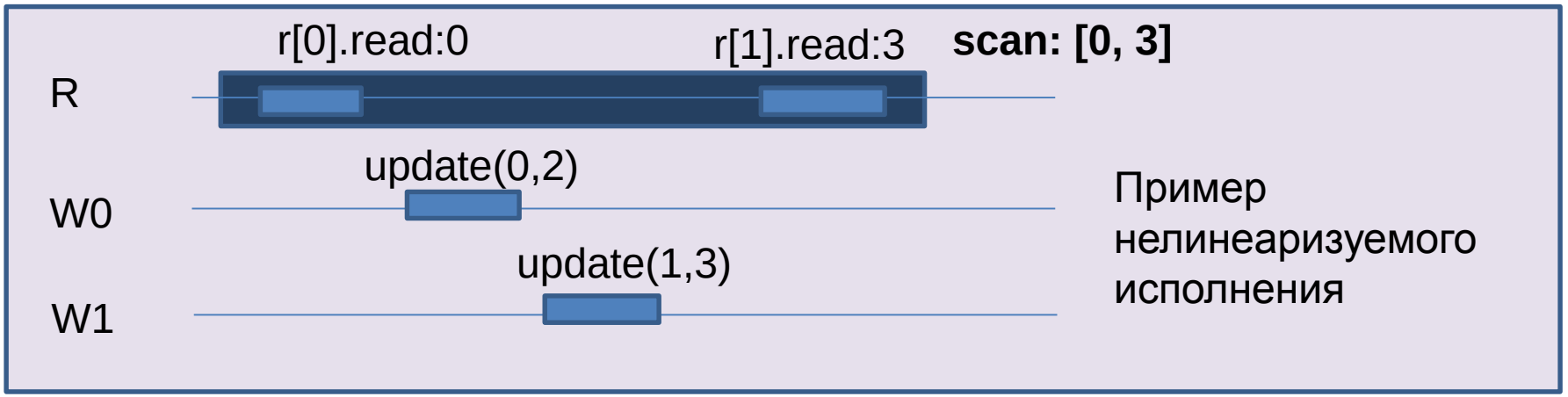
```
// Последовательная спецификация
class Snapshot:
    shared int r[N]

    def update(i, x):
        r[i] = x

    def scan():
        return copy()

    private def copy():
        res = new int[N]
        for i = 0..N-1: res[i] = r[i]
        return res
```

- Набор SW атомарных регистров (по регистру на поток)
- Любой поток может вызвать scan() чтобы получить снимок состояния всех регистров
- Методы должны быть атомарными (линеаризуемыми)

Атомарный снимок состояния N регистров

- Наивная реализация не обеспечивает атомарность

Операция	r[0]	r[1]
update(0,2)	2	0
update(1,3)	2	3



Атомарный снимок состояния N регистров, без блокировок (lock free)

```
shared (int x, int v) r[N]

// wait-free
def update(i, x):
    r[i] = (x, r[i].v + 1)

// lock-free
def scan():
    old = copy()
    loop:
        cur = copy()
        if forall i: cur[i].v == old[i].v:
            return cur.x
        old = cur
```

- Каждый регистр хранит версию
- При обновлении версию увеличиваем на один
- При чтении читает до тех пор, пока не получит два подряд одинаковых снимка

Атомарный снимок состояния N регистров, без ожидания (wait-free) – update

```
shared (int x, int v, int[N] s) r[N]

def update(i, x):
    s = scan()
    r[i] = (x, r[i].v + 1, s)
```

- Для реализации **без ожидания** надо чтобы потоки **помогали** друг другу
- Каждый регистр так же хранит копию снимка s
- При обновлении делаем вложенный scan, чтобы помочь параллельно работающим операциям

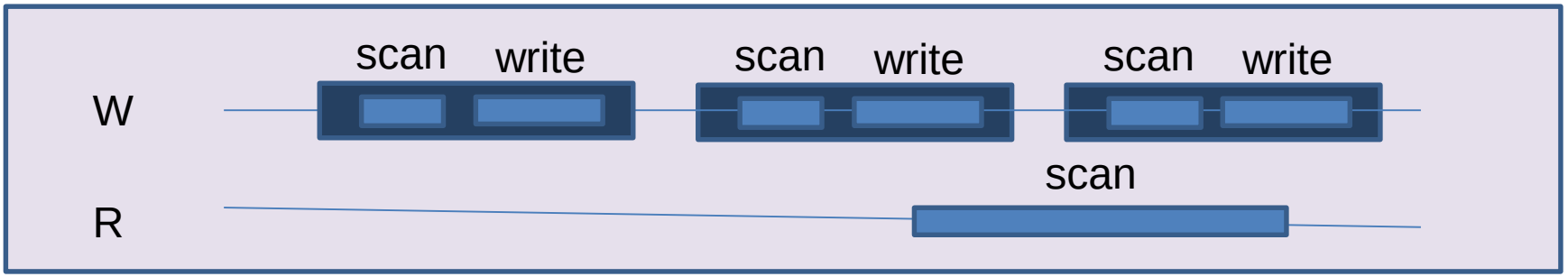
Атомарный снимок состояния N регистров, без ожидания (wait-free) – scan

```
shared (int x, int v, int[N] s) r[N]

// wait-free, O(N^2)
def scan():
    old = copy()
    boolean updated[N]
    loop:
        cur = copy()
        for i = 0..N-1:
            if cur[i].v != old[i].v:
                if updated[i]: return cur.s
                else:
                    update[i] = true
                    old = cur
                    continue loop
        return cur.x
```

- **Лемма**: Если значение поменялось второй раз, то хранящаяся там копия снимка s была получена вложенной операцией scan.
- **Следствие**: Этот алгоритм выдает атомарный снимок

Атомарный снимок состояния N регистров, без ожидания (wait-free) – лемма

- **Лемма**: Если значение поменялось второй раз, то хранящаяся там копия снимка s была получена вложенной операцией scan.



Лекция 4

УНИВЕРСАЛЬНЫЕ ОПЕРАЦИИ И КОНСЕНСУС

Консенсус

Задача о консенсусе

```
class Consensus:
  def decide(val):
    …
    return decision
```

✓ Каждый поток использует объект Consensus **один раз**

- **Согласованность** (consistent): все потоки должны вернуть одно и то же значение из метода decide
- **Обоснованность** (valid): возвращенное значение было входным значением какого-то из потоков

Будем пытаться реализовать протокол консенсуса используя регистры и другие примитивы

Консенсус с блокировкой

```
shared int decision // init NA
Mutex mutex

def decide(val):
    mutex.lock()
    if decision == NA:
        decision = val
    mutex.unlock()
    return decision
```

- Тривиальная реализация протокола консенсуса с помощью взаимного исключения для любого количества потоков
- Но мы хотим реализацию **без ожидания** (wait-free)

Консенсусное число

- Если с помощью класса [атомарных] объектов C и атомарных регистров можно реализовать консенсусный протокол **без ожидания** (wait-free) для N потоков (и не больше), то говорят что у класса C **консенсусное число** равно N.
- **ТЕОРЕМА:** Атомарные регистры имеют консенсусное число 1.
  - Т.е. с помощью атомарных регистров даже 2 потока не могут придти к консенсусу без ожидания (докажем от противного) для для 2-х возможных значений при T = {0, 1}
  - С ожиданием задача решается очевидно (с помощью любого алгоритма взаимного исключения).

Определения и леммы для любых классов объектов

- Определения и концепции:
  - Исходные объекты **атомарны**. Значит любое исполнения можно рассматривать как последовательное в каком-то порядке
  - Рассматриваем дерево состояния, листья – конечные состояния помеченные 0 или 1 (в зависимости от значения консенсуса).
  - x-валентное состояние системы (x = 0,1) – консенсус во всех нижестоящих листьях будет x.
  - Бивалентное состояние – возможен консенсус как 0 так и 1.
  - Критическое состояние – такое бивалентное состояние, все дети которого одновалентны.
- **ЛЕММА**: Существует начальное бивалентное состояние.
- **ЛЕММА**: Существует критическое состояние.

Доказательство для атомарных регистров

- Рассмотрим возможные пары операций в критическом состоянии:
  - Операции над разными регистрами – коммутируют.
  - Два чтения – коммутируют.
  - любая операция + Запись – состояние пишущего потока не зависит от порядка операций.

Read-Modify-Write регистры

```
class RMWRegister:
    private shared int reg

    def read():
        return reg

    def getAndF(args):
        do atomically:
            old = reg
            reg = F(args)(reg)
            return old
```

- Для функции или класса функций F(args)
  - getAndSet (exchange), getAndIncrement, getAndAdd и т.п.
  - get (read) это тоже [тривиальная] RMW операция без дополнительных аргументов для F() == id.

Нетривиальные RMW регистры

```
threadlocal int     id  // 0 or 1

shared RMWRegister rmw
shared int proposed[2]

def decide(val):
    proposed[id] = val
    if rmw.getAndF() == v0:
        return proposed[i]
    else:
        return proposed[1-i]
```

Реализация протокола для 2-х потоков

- Консенсусное число нетривиального RMW регистра >= 2.
  - Нужно чтобы была хотя бы одна «подвижная» точка функции F, например F(v0) == v1 != v0.

Common2 RMW регистры

- Определения
  - F1 и F2 коммутируют если F1(F2(x)) == F2(F1(x))
  - F1 перезаписывает F2 если F1(F2(x)) == F1(x)
  - Класс C RMW регистров принадлежит Common2, если любая пара функций либо коммутирует, либо одна из функций перезаписывает другую.
- **ТЕОРЕМА**: Нетривиальный класс Common2 RMW регистров имеет консенсусное число 2.
  - Третий поток не может отличить глобальное состоянием при изменения порядка выполнения коммутирующих или перезаписывающих операций в критическом состоянии.

Универсальные объекты

```
class CASRegister:
    private shared int reg

    def CAS(expect, update):
        do atomically:
            old = reg
            if old == expect:
                reg = update
                return true
            return false
```

- Объект с консенсусный числом бесконечность называется «**универсальный объект**».
  - По определению, с помощью него можно реализовать консенсусный протокол для любого числа потоков.

Примеры:

- compareAndSet (CAS) aka testAndSet (возвращает boolean)
- compareAndExchange aka CMPXCHG (возвращает старое значение как и положено RMW операции)

CAS и консенсус

```
def decide(val):
    if CAS(NA, val):
        return val
    else:
        return read()
```

Реализация протокола через CAS+READ

```
def decide(val):
    old = CMPXCHG(NA, val):
    if old == NA:
        return val
    else:
        return old
```

Реализация протокола через CMPXCHG

Универсальность консенсуса

- **ТЕОРЕМА**: Любой последовательный объект можно реализовать без ожидания (wait-free) для N потоков используя консенсусный протокол для N объектов.
  - Такое построение называется **универсальная конструкция**
  - **Следствие 1**: С помощью любого класса объектов с консенсусным числом N можно реализовать любой объект с консенсусным числом <= N.
  - **Следствие 2**: С помощью универсального объекта можно реализовать вообще любой объект
    - Сначала реализуем консенсус для любого числа потоков (по определению универсального объекта)
    - Потом через консенсус любой другой объект используя универсальную конструкцию.

Универсальная конструкция без блокировки (lock-free) через CAS

```
shared CASRegister reg

def concurrentOperationX(args):
    loop:
        old = reg.read()
        upd = old.deepCopy()
        res = upd.serialOperationX(args)
    until reg.CAS(old, upd)
    return res
```

✓ Если r хранит указатель на данные, то deepCopy должен сделать полную копию

- **Без блокировки** универсальная конструкция проста и практична, если использовать **CAS** в качестве примитива.
  - Для реализации **через консенсус** надо чтобы каждый объект консенсуса использовался потоком один раз
  - Для реализации **без ожидания** нужно чтобы потоки помогали друг другу

Универсальная конструкция через консенсус

```
class Node:
    val                     // readonly
    Consensus next  // init fresh obj

shared Node root        // readonly
threadlocal Node last // init root

def concurrentOperationX(args):
    loop:
        old = last.val
        upd = old.deepCopy()
        res = upd.serialOperationX(args)
        node = new Node(upd)
        last = last.next.decide(node)
    until last == node // until we're in list
    return res
```

- **Идея:** Представим объект в виде односвязного списка состояний.
  - Последний элемент в списке это текущее состояние
  - Объект консенсуса для списка следующий состояний. Каждый поток предлагает свой вариант и они приходят к консенсусу
- Получаем реализацию **без блокировки** очевидным образом

Универсальная конструкция без ожидания (1)

```
class Node:
    args                    // readonly
    Consensus next  // init fresh obj

threadlocal Node last // init root
threadlocal my

def concurrentOperationX(args):
    Node node = new Node(args)
    while last != node:  // until we're in list
        last = last.next.decide(node)
        res = my.serialOperationX(last.args)
    return res
```

- **Идея 1:** Хранить в узле операцию которую надо выполнить, а не результат её выполнения
  - Каждый поток будет хранить и обновлять свою локальную копию объекта

Универсальная конструкция без ожидания (2)

```
class Node:
    int seq                 // init 0
    args                    // readonly
    Consensus next  // init fresh obj

def concurrentOperationX(args):
    Node node = new Node(args)
    while last != node: // until we're in list
        Node prev = last
        last = prev.next.decide(node)
        last.seq = prev.seq + 1
        res = my.serialOperationX(last.args)
    return res
```

- **Идея 2:** Будет нумеровать выполненные операции последовательными целыми числами, заведя переменную seq:
  - После выполнения будем прописывать номер выполненной операции в Node.seq как только она успешно выполнена

Универсальная конструкция без ожидания (3)

```
shared Node[] know      // init root

def concurrentOperationX(args):
    Node node = new Node(args)
    know[id] = maxSeqFrom(know)
    while know[id] != node: // until we're in
        Node prev = know[id]
        know[id] = prev.next.decide(node)
        know[id].seq = prev.seq + 1
    return updateMyLastTo(node)

def updateMyLastTo(node):
    while last != node:
        res = my.serialOperationX(last.args)
        last = last.next
    return res
```

- **Идея 3:** Каждый поток будет хранить последнее известное ему значение конца списка в элементе массива know[id] который виден всем остальным объектам
  - Перед началом работы будем выбирать там элемент с макс. seq
  - Это позволит более медленным потокам догонять более быстрые и конкурировать в decide

Универсальная конструкция без ожидания (4)

```
shared Node[] announce // init root

def concurrentOperationX(args):
    announce[id] = new Node(args)
    know[id] = maxSeqFrom(know)
    // loop until we're is in list
    while announce[id].seq == 0:
        Node help =
            announce[know[id].seq % N]
        Node prev = help if help.seq == 0
            else announce[id]
        know[id] = prev.next.decide(node)
        know[id].seq = prev.seq + 1
    know[id] = announce[id]
    return updateMyLastTo(announce[id])
```

- **Идея 4:** Каждый поток будет заранее записывать операция которую он планирует выполнить в массив announce
  - И пользуясь этим массивом помогать потоку seq % N (N – число потоков)
  - Предпочитая помощь своему выполнению
  - Тогда за N шагов каждому потоку помогут

Иерархия объектов

Объект	**Консенсусное число**
Атомарные регистры, снимок состояния нескольких регистров	1
getAndSet (атомарный обмен), getAndAdd, очередь, стек	2
Атомарная запись m регистров из m(m+1)/2 регистров	m
compareAndSet, LoadLinked/StoreConditional	∞