

Многопоточное Программирование: Введение и Мотивация

Роман Елизаров, JetBrains, elizarov@gmail.com

Никита Коваль, JetBrains, ndkoval@ya.ru

ИТМО 2020



ITMO UNIVERSITY

Базовые правила игры

- **Домашние задания**

- После каждой ($\pm \epsilon$) лекции
- Через GitHub
- Инструкции будут позже

- **Контрольные вопросы**

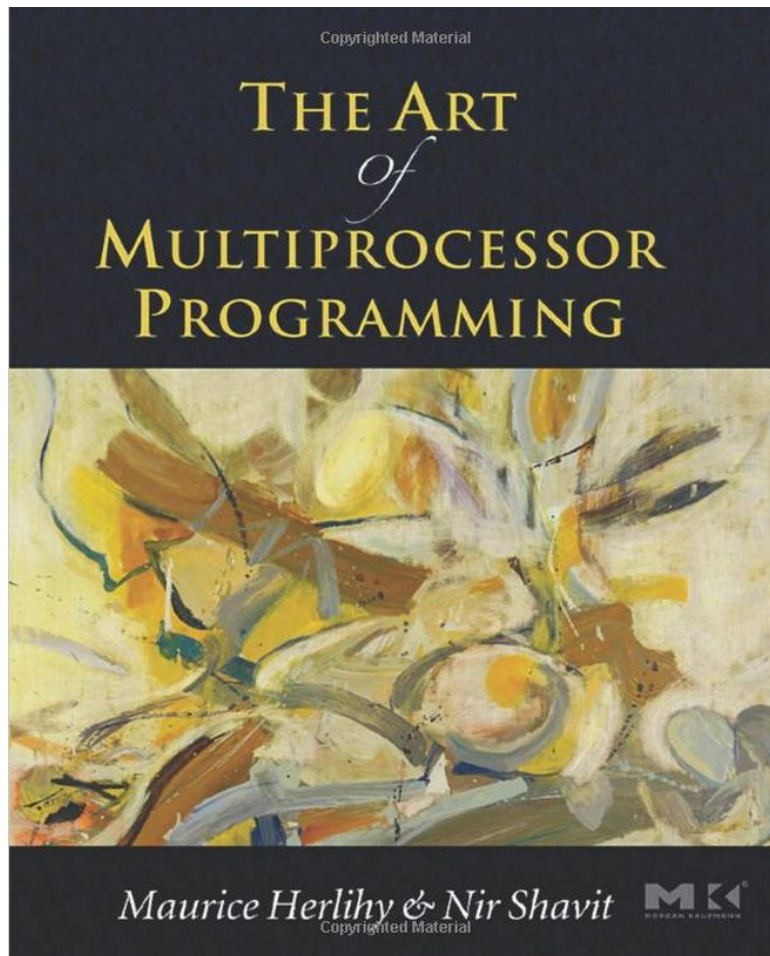
- Для вас, чтобы проверить понимание
- (запаситесь бумагой и ручкой)

- **Зачет по предмету**

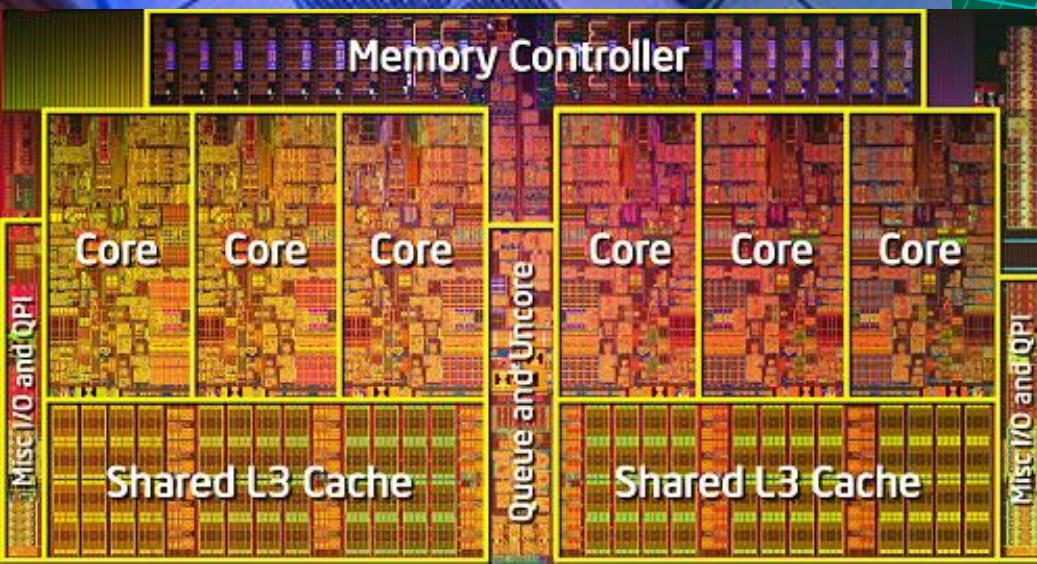
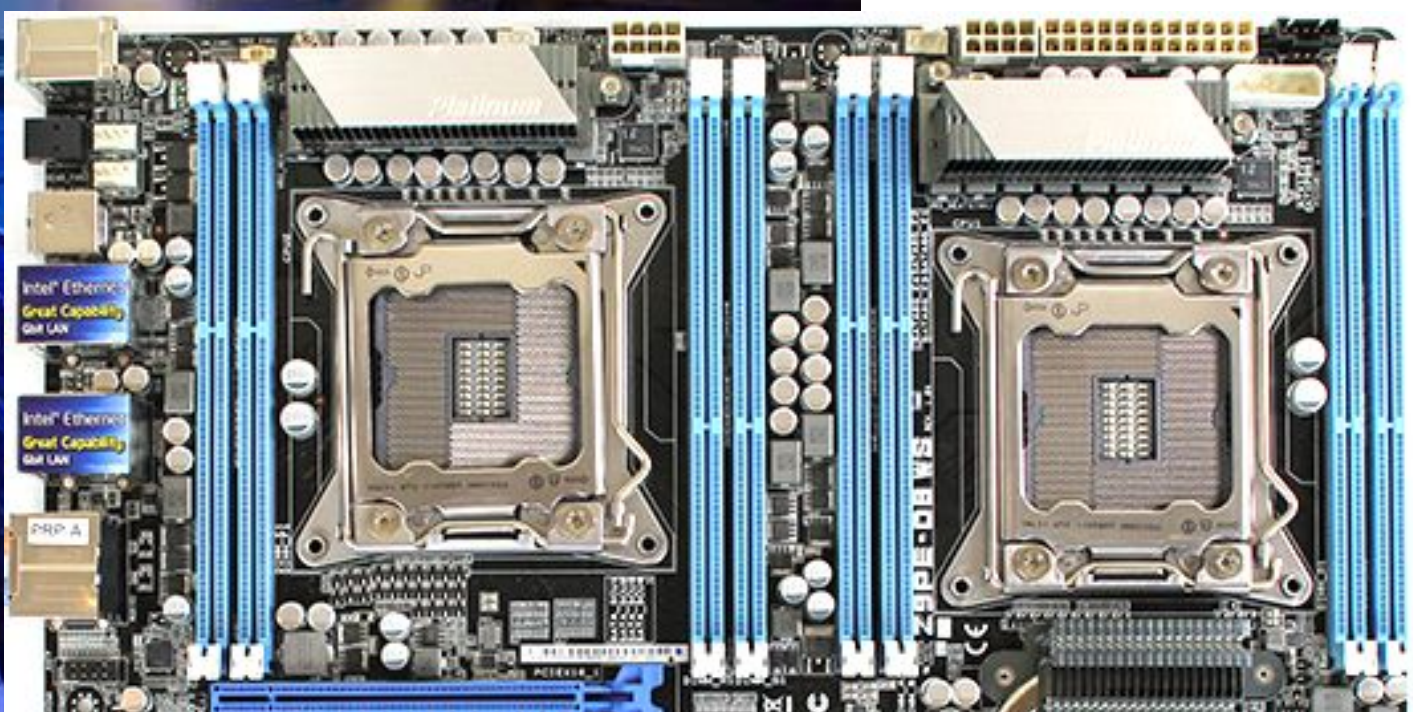
- Сдача всех обязательных ДЗ => Зачет

ЛИТЕРАТУРА

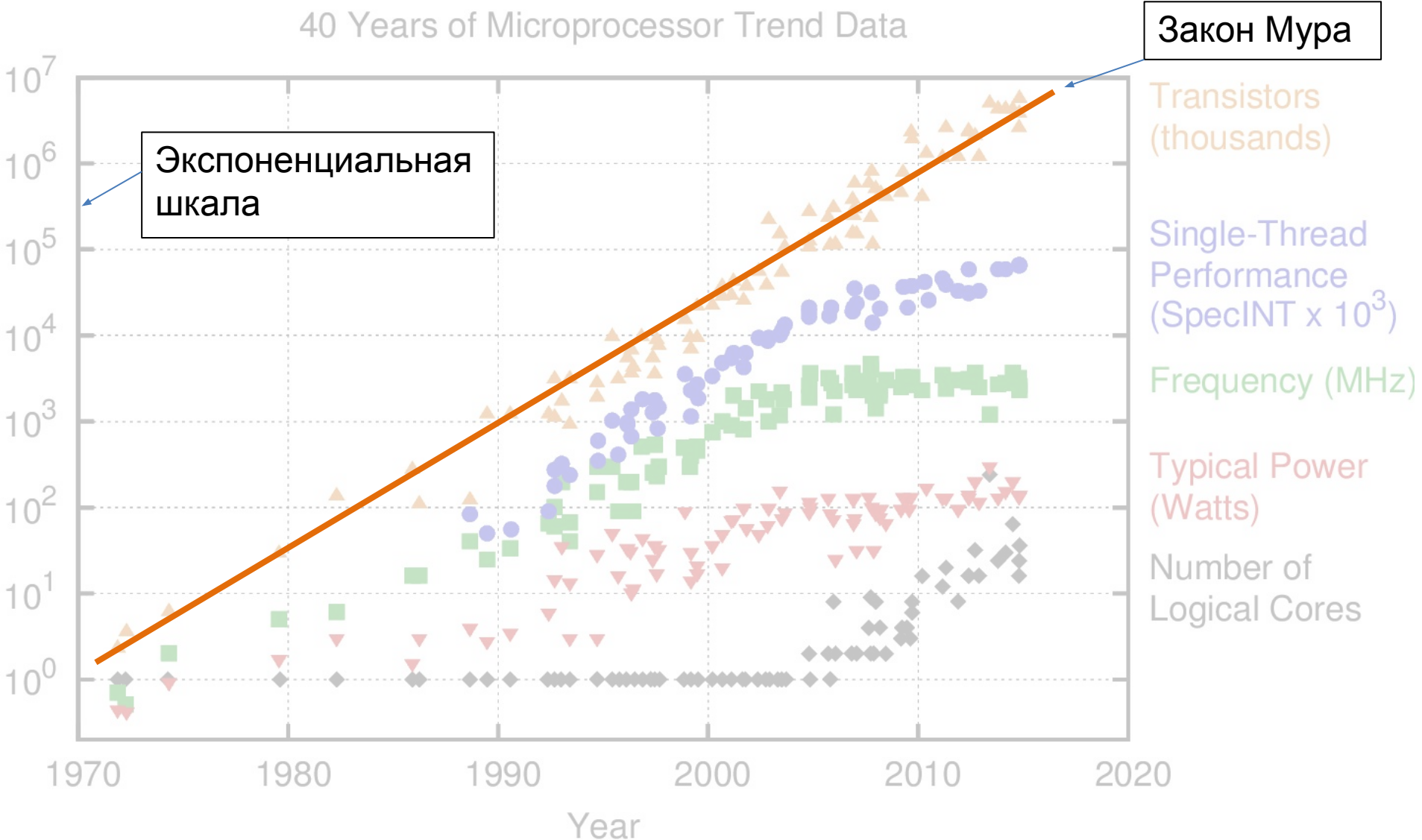
Для самостоятельной подготовки



Зачем нам это?



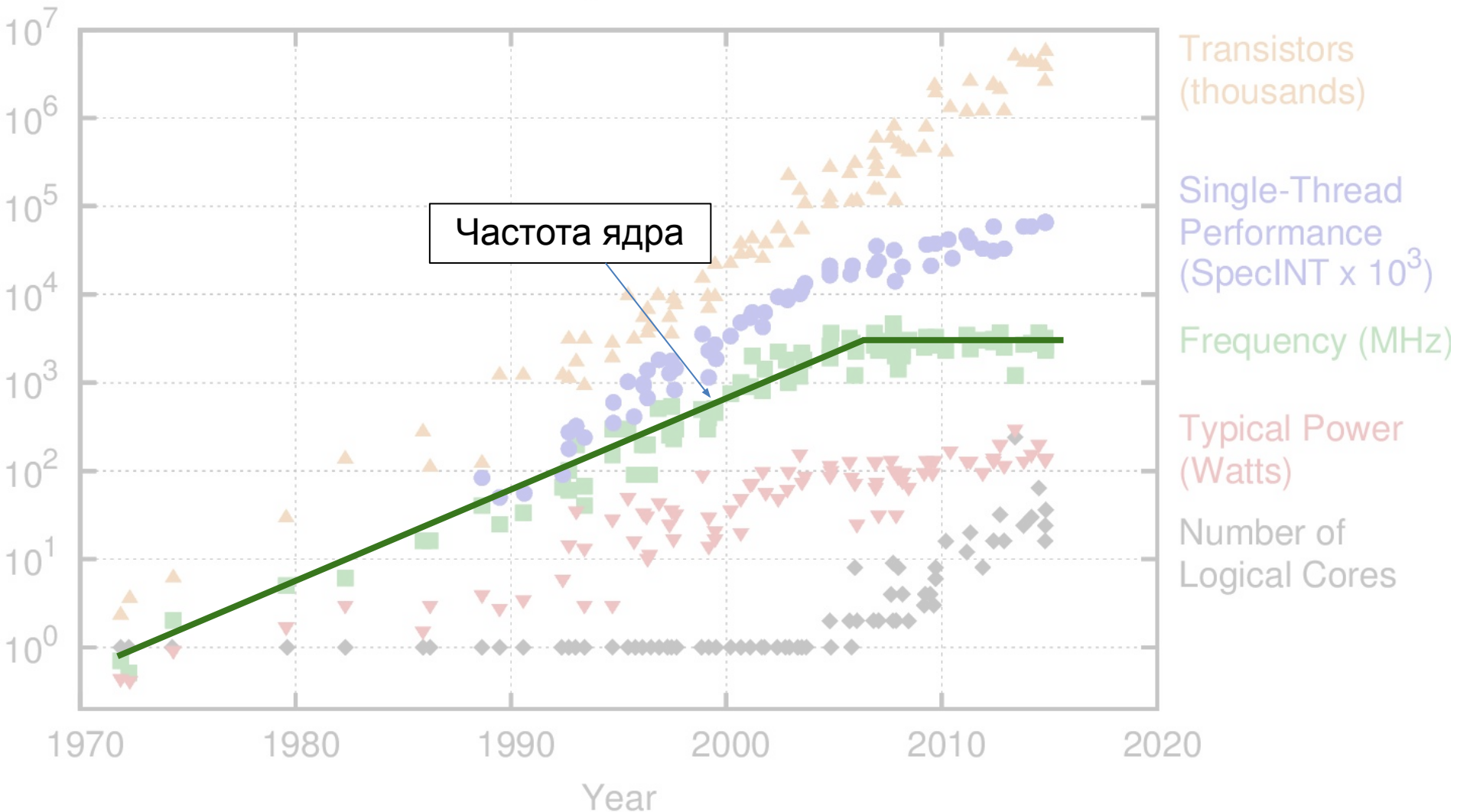
Закон Мура и “The free lunch is over”



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

Закон Мура и “The free lunch is over”

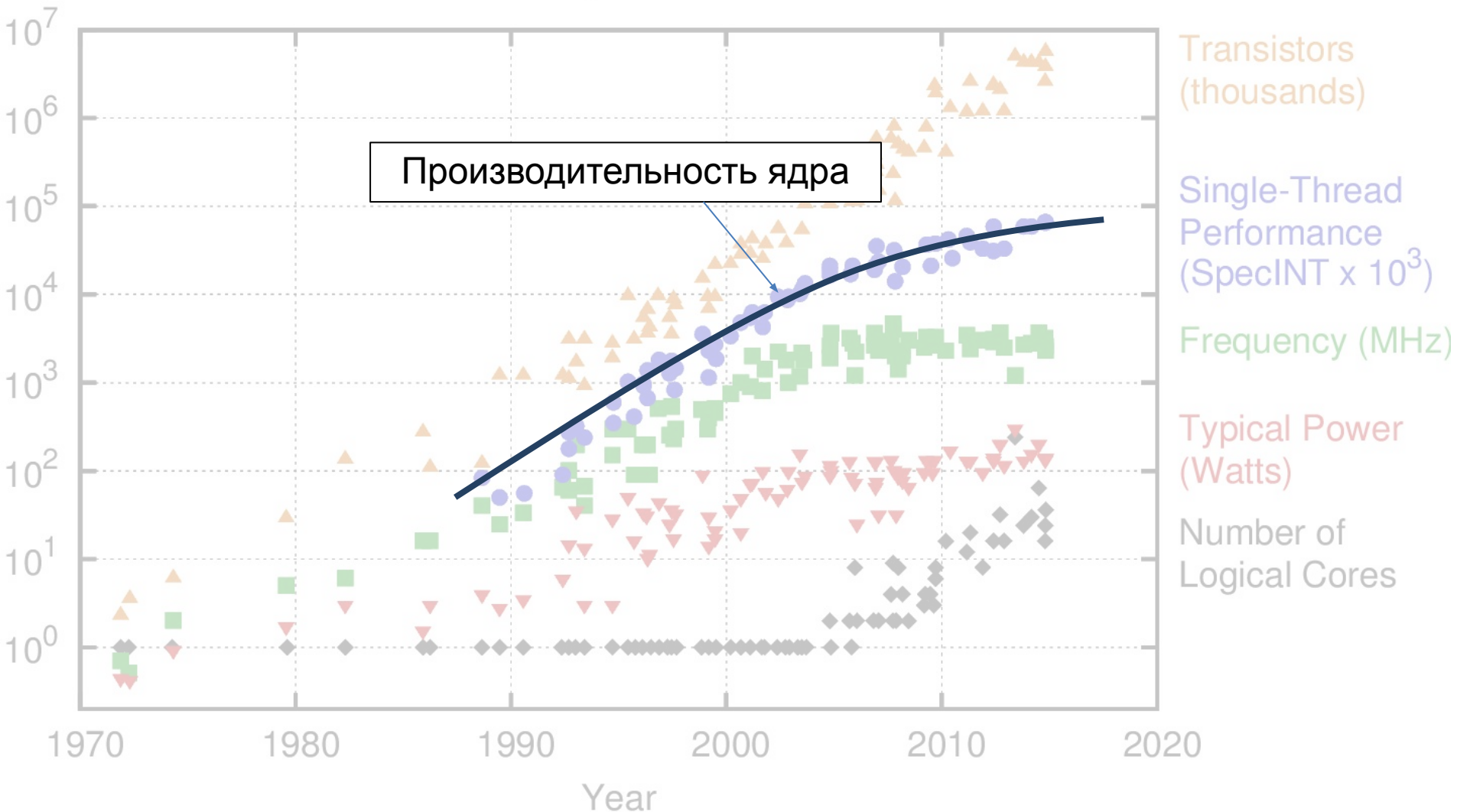
40 Years of Microprocessor Trend Data



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

Закон Мура и “The free lunch is over”

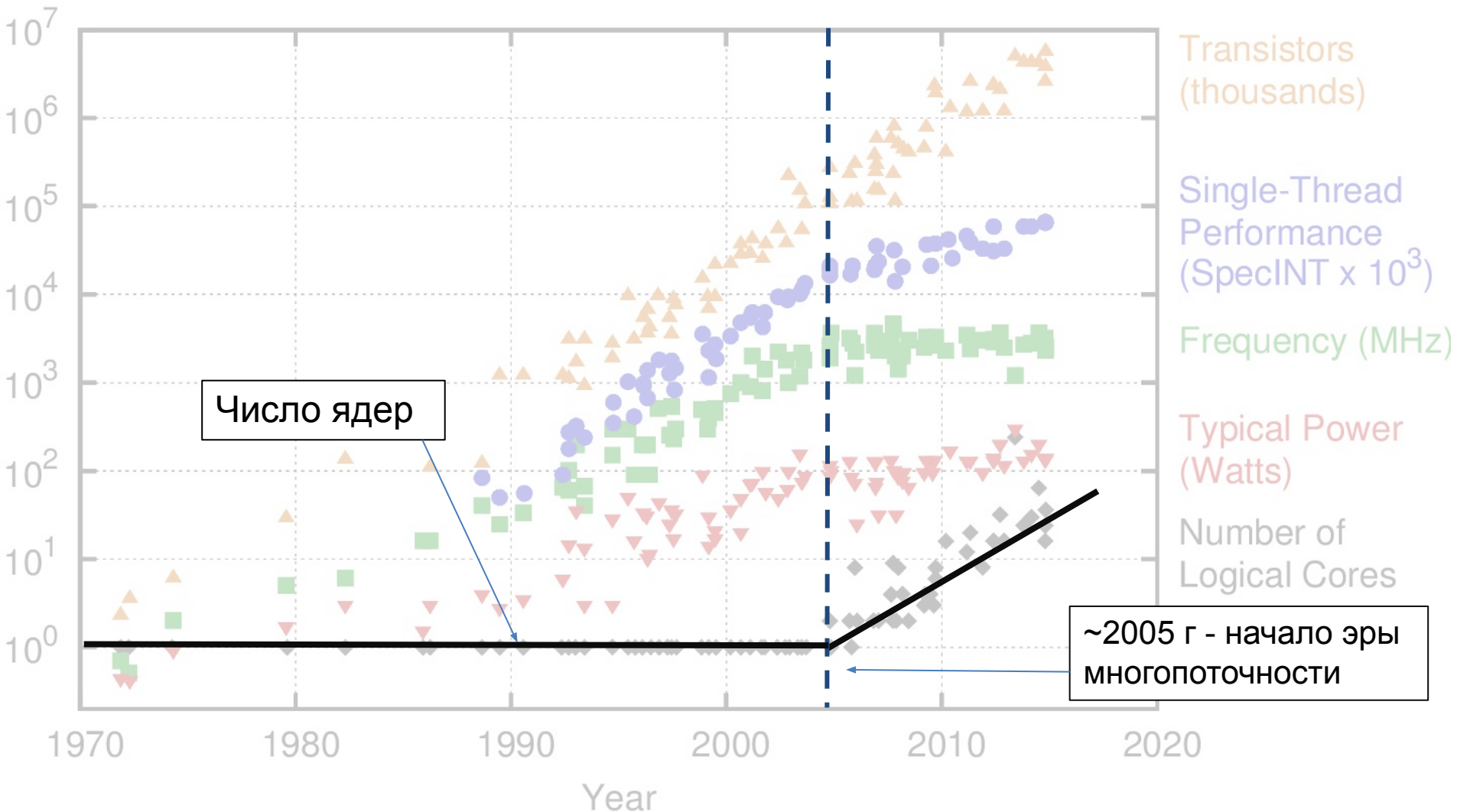
40 Years of Microprocessor Trend Data



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

Закон Мура и “The free lunch is over”

40 Years of Microprocessor Trend Data



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

Масштабирование: Закон Мура (традиционное)

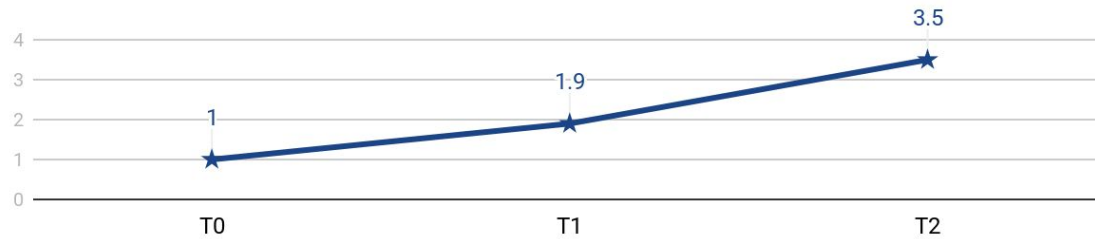
Код



Ядро
процессора



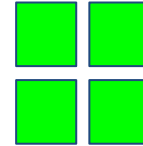
Скорость



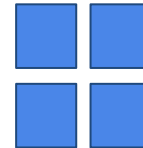
Время

Масштабирование: Эра Многопоточности (идеал)

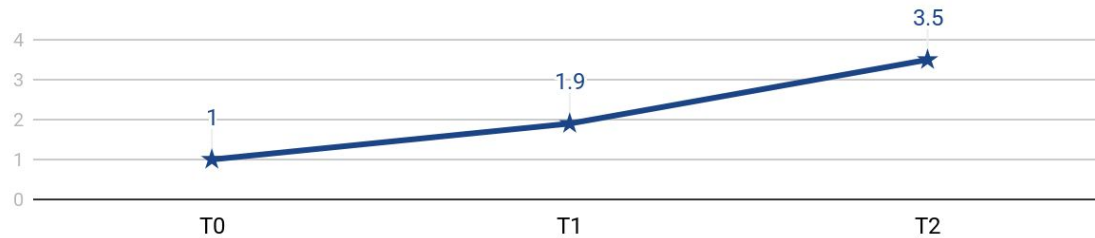
Код



Ядро
процессора



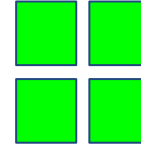
Скорость



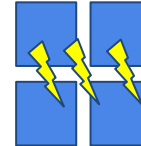
Время

Масштабирование: Эра Многопоточности (реально)

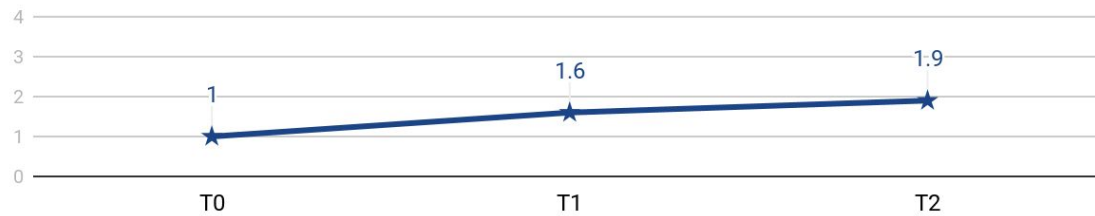
Код



Ядро
процессора



Скорость



Время

Закон Амдала

N - число потоков

$$\text{Ускорение кода } S = \frac{\text{Время на 1 ядре}}{\text{Время на } N \text{ ядрах}}$$

Закон Амдала

N - число потоков

P - доля параллельного кода

$$\text{Ускорение кода } S = \frac{1}{1-P + P/N}$$

Закон Амдала

N - число потоков

P - доля параллельного кода

$$\text{Ускорение кода } S = \frac{1}{\underbrace{1-P}_{\text{Последовательная часть}} + P/N}$$

Закон Амдала

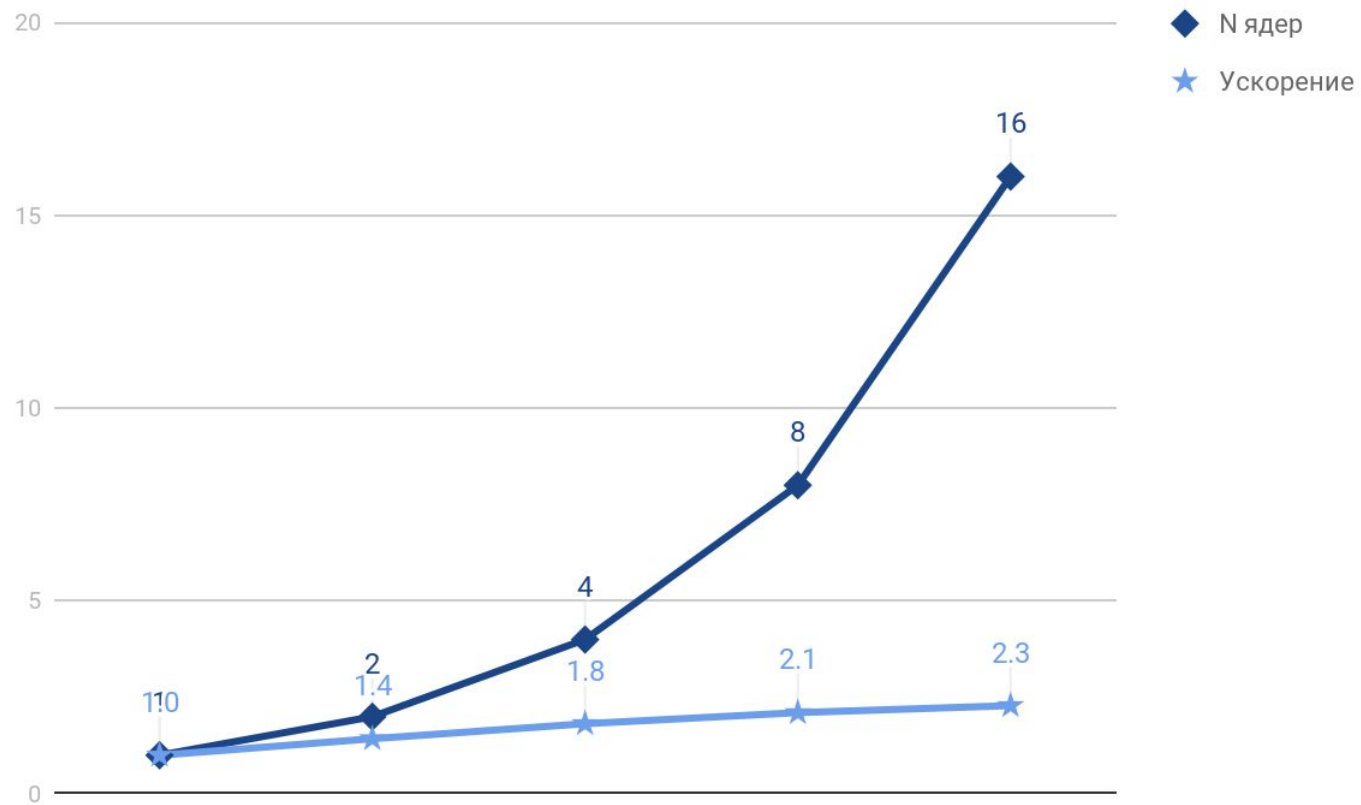
N - число потоков

P - доля параллельного кода

$$\text{Ускорение кода } S = \frac{1}{1-P + \underbrace{P/N}_{\text{Параллельная часть}}}$$

Закон Амдала

P - доля параллельного кода = 60%



Закон Амдала

N - число потоков $\rightarrow \infty$

P - доля параллельного кода

$$\text{Макс ускорение кода } S = \frac{1}{1-P}$$

Закон Амдала

N - число потоков $\rightarrow \infty$

P - доля параллельного кода = 60%

$$\text{Макс ускорение кода } S = \frac{1}{1-P} = 2.5$$

Закон Амдала

N - число потоков $\rightarrow \infty$

P - доля параллельного кода = 95%

$$\text{Макс ускорение кода } S = \frac{1}{1-P} = 20$$

Закон Амдала

N - число потоков $\rightarrow \infty$

P - доля параллельного кода = 99%

$$\text{Макс ускорение кода } S = \frac{1}{1-P} = 100$$

Контрольный вопрос

Некий алгоритм может быть практически полностью выполнен параллельно на разных процессорах, но 5% от всех вычислений все равно должны быть выполнены последовательно. Во сколько раз может увеличиться скорость работы этого алгоритма при запуске его на системе с 16 процессорами вместо системы с 1 процессором?

✓ $1 / (0.05 + 0.95 / 16) = \mathbf{9.14}$

**Для масштабируемости
нужно больше
параллелизма**

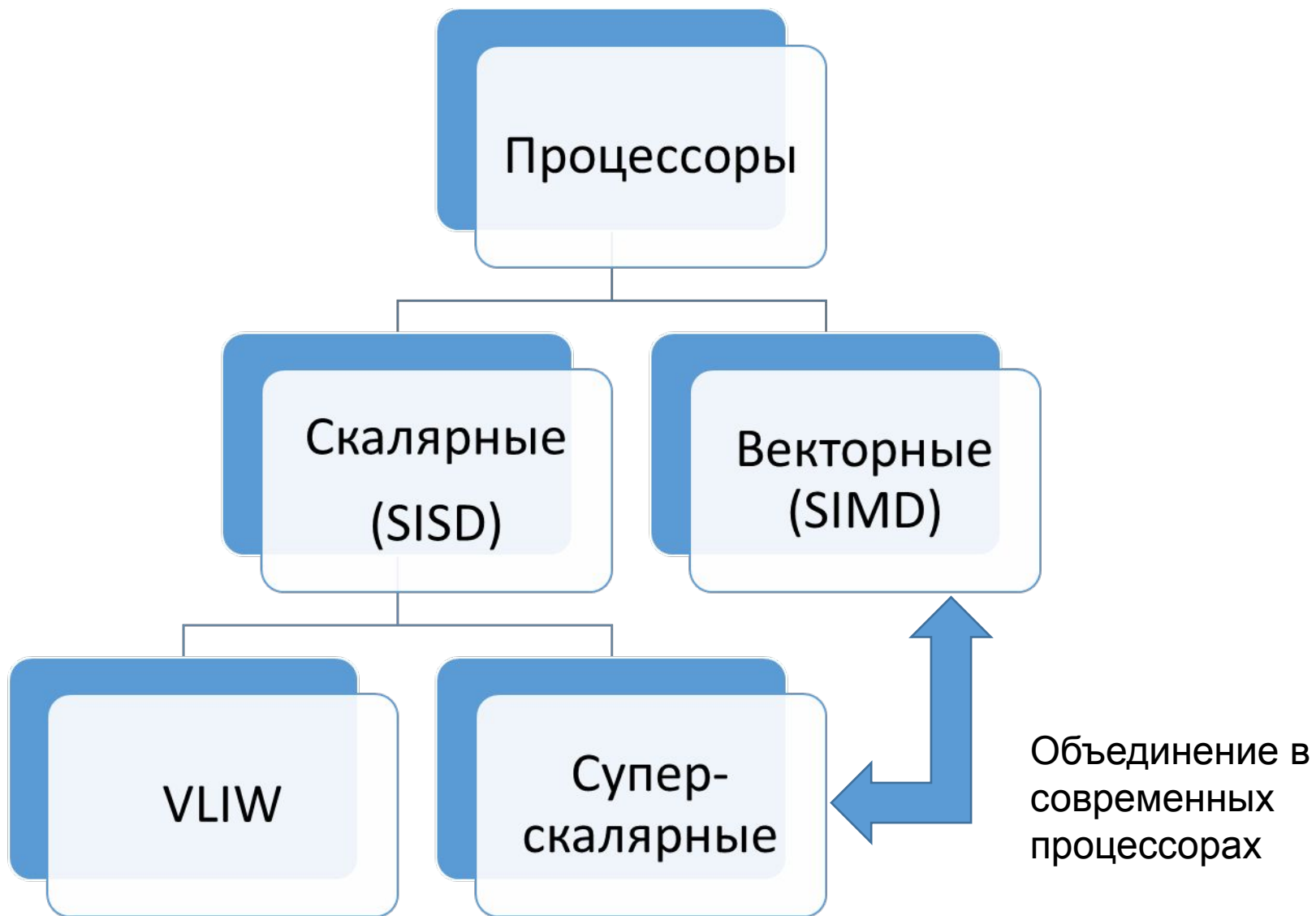
Разные типы параллелизма

Параллелизм на уровне инструкций (ILP – instruction level parallelism)

<pre>a = b + c; // (1) d = e + f; // (2)</pre>

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)

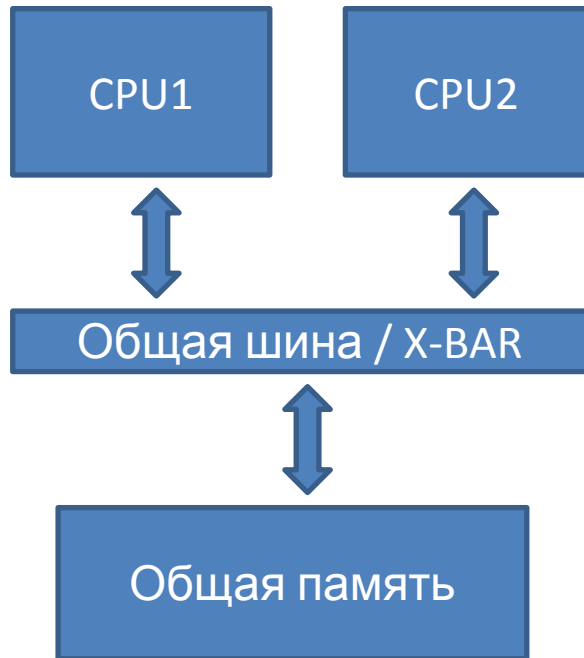


У параллелизма на уровне
инструкций есть предел



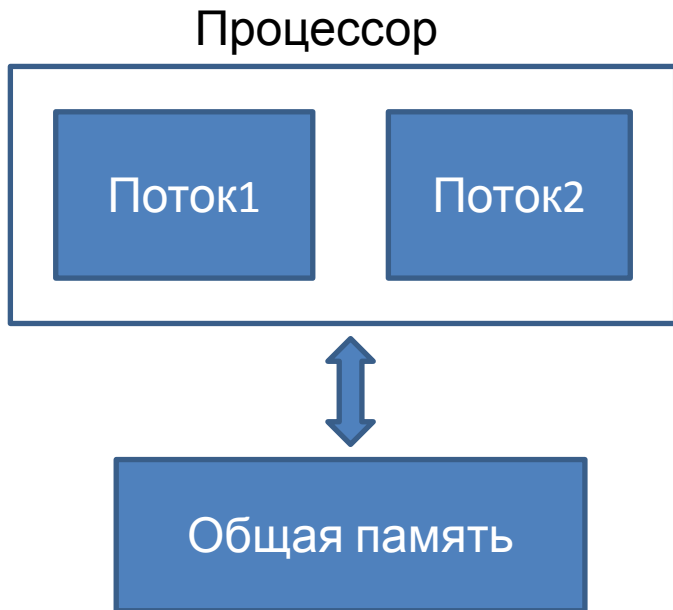
Параллельное
программирование

Симметричная мультипроцессорность (SMP – symmetric multiprocessing)



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

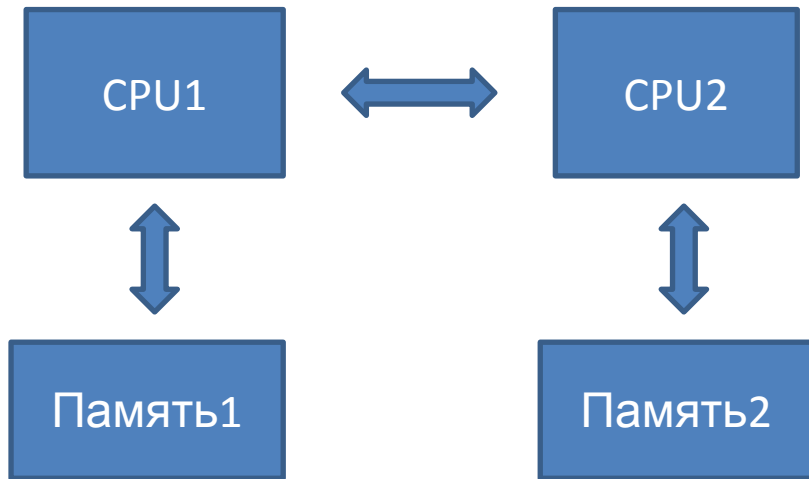
Одновременная многозадачность (SMT – simultaneous multithreading)



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

Для программиста выглядит как SMP.

Асимметричный 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...

Контрольный вопрос

1. Какие утверждения верные?

-  а. В операционной системе всегда выполняет один процесс, но множество потоков может выполняться одновременно.
-  б. Внутри одного процесса может работать множество потоков.
-  в. Каждый процесс имеет свою собственную область памяти, отличную от памяти других процессов.

Формализм

Нужна формальная модель параллельных вычислений

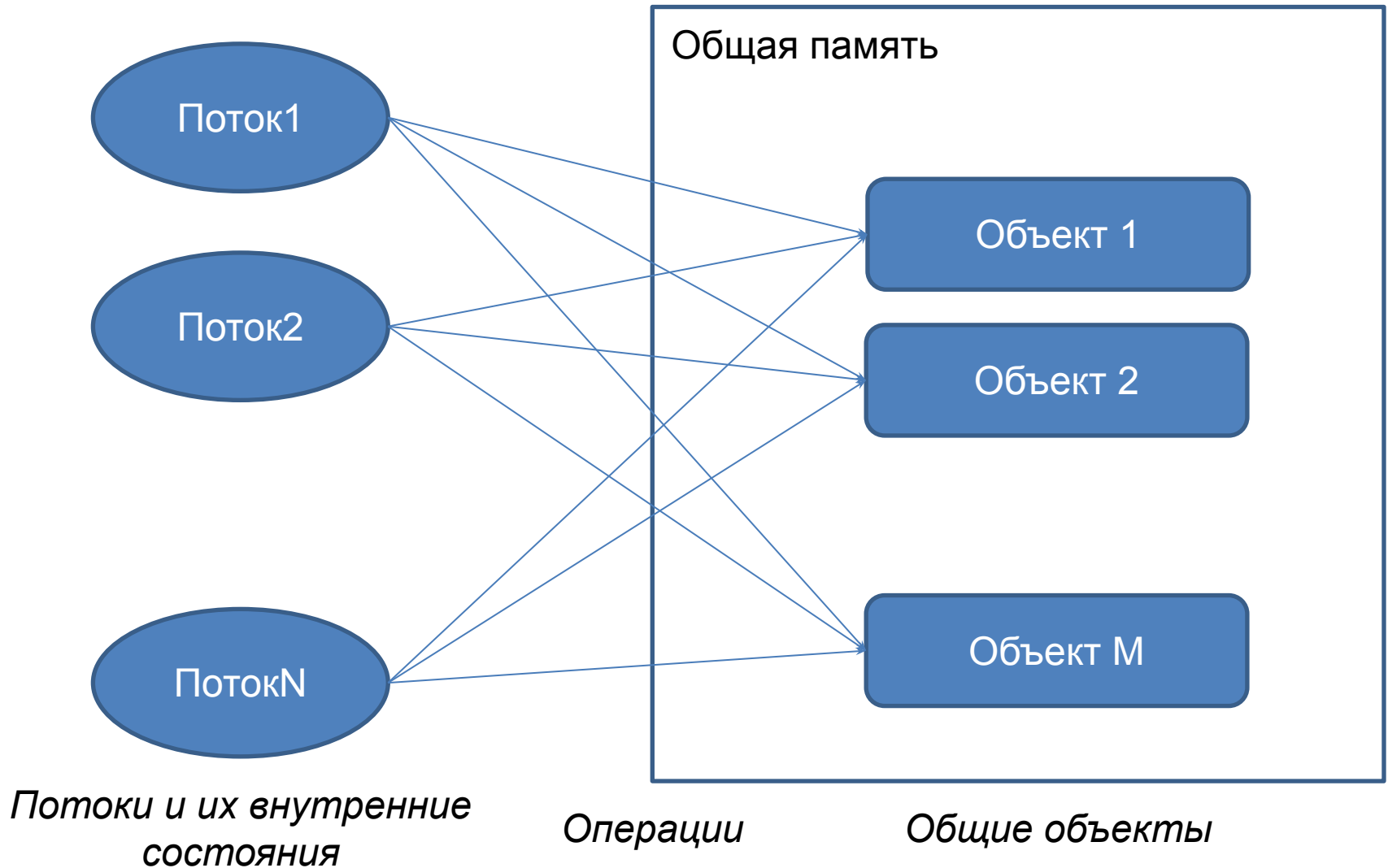
- Чтобы доказывать:
 - корректность алгоритмов
 - невозможность построения тех или иных алгоритмов
 - минимально-необходимые требования для тех или иных алгоритмов
- **Для формализации отношений между прикладным программистом и разработчиком компилятора и системы исполнения кода**



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 Р**», если *программа А имеет свойство Р при любом 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)
 - И у каждого есть по одной локальной переменной (r1 в P, r2 в Q)
 - Какие есть варианты r1 и r2 после завершения исполнения?

shared int x = 0, y = 0

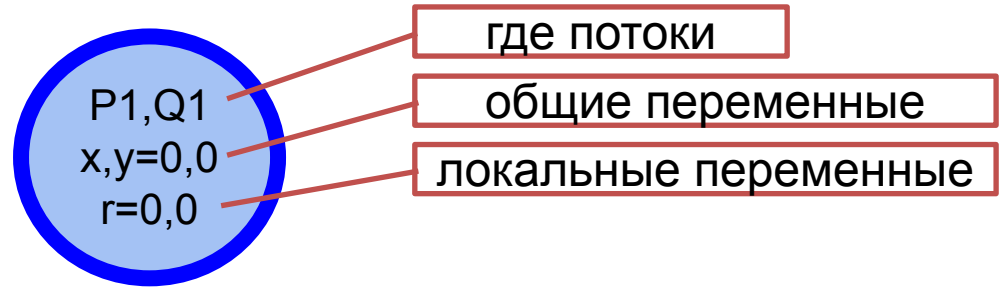
thread P:

1: x = 1
2: r1 = y
3: stop

thread Q:

1: y = 1
2: r2 = x
3: stop

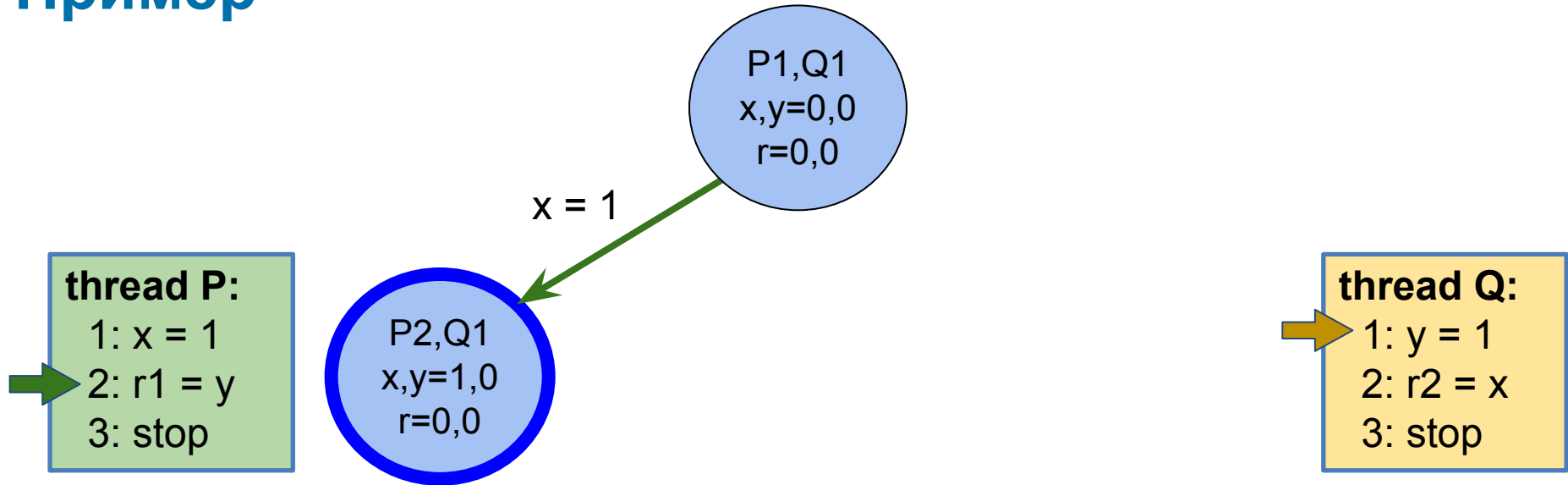
Пример



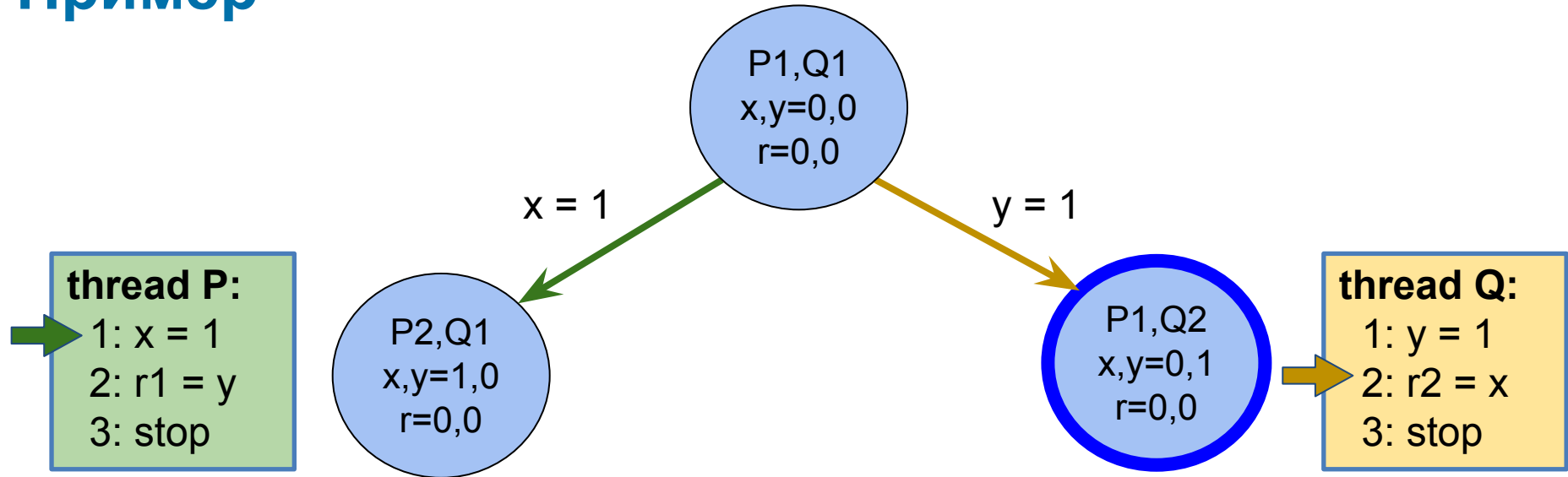
→ **thread P:**
1: x = 1
2: r1 = y
3: stop

→ **thread Q:**
1: y = 1
2: r2 = x
3: stop

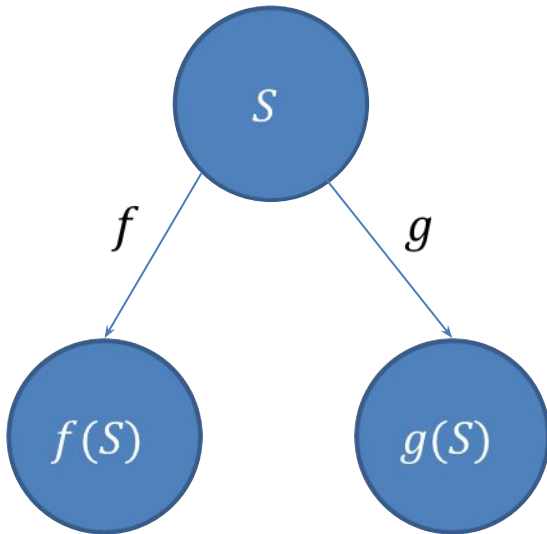
Пример



Пример

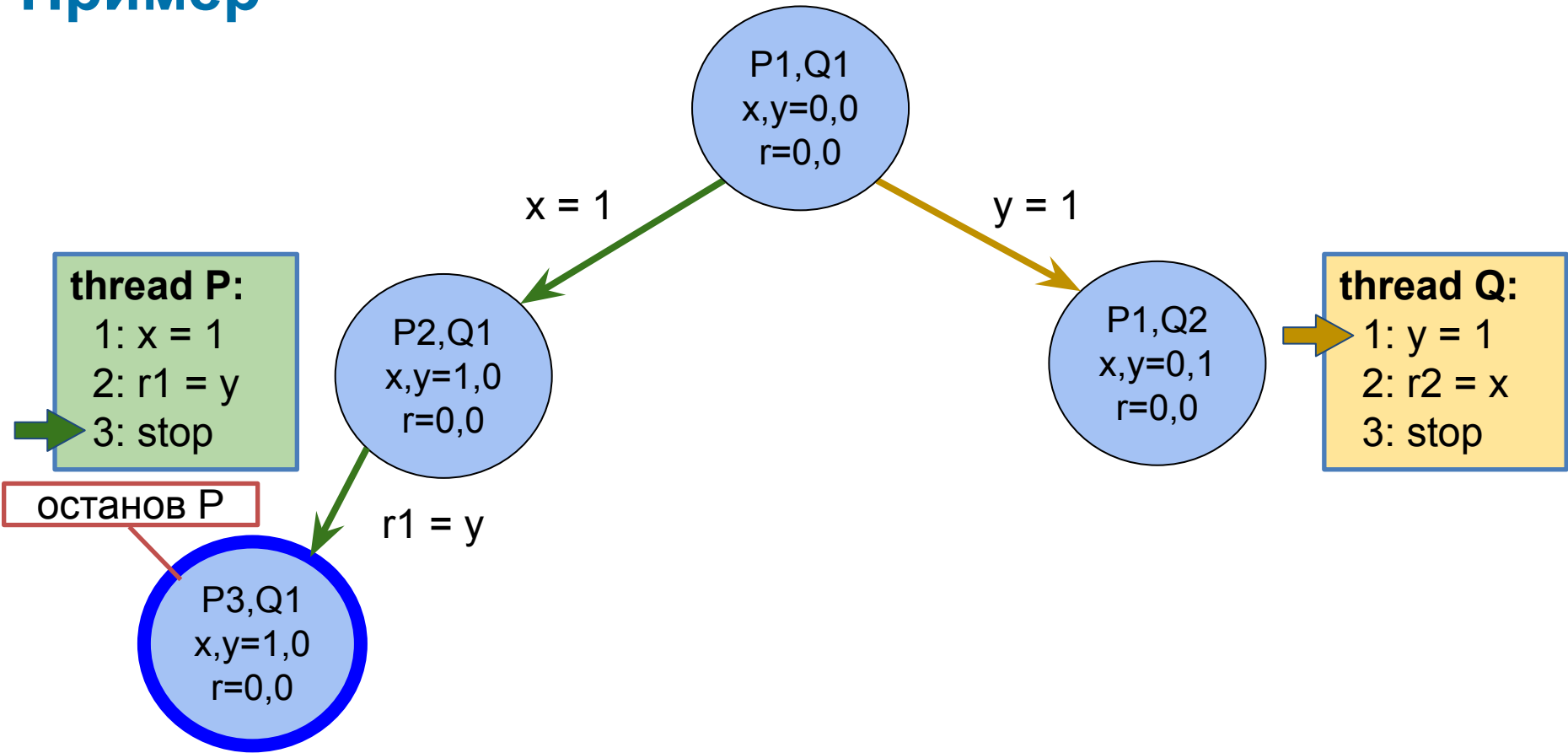


Моделирование исполнений через чередование операций

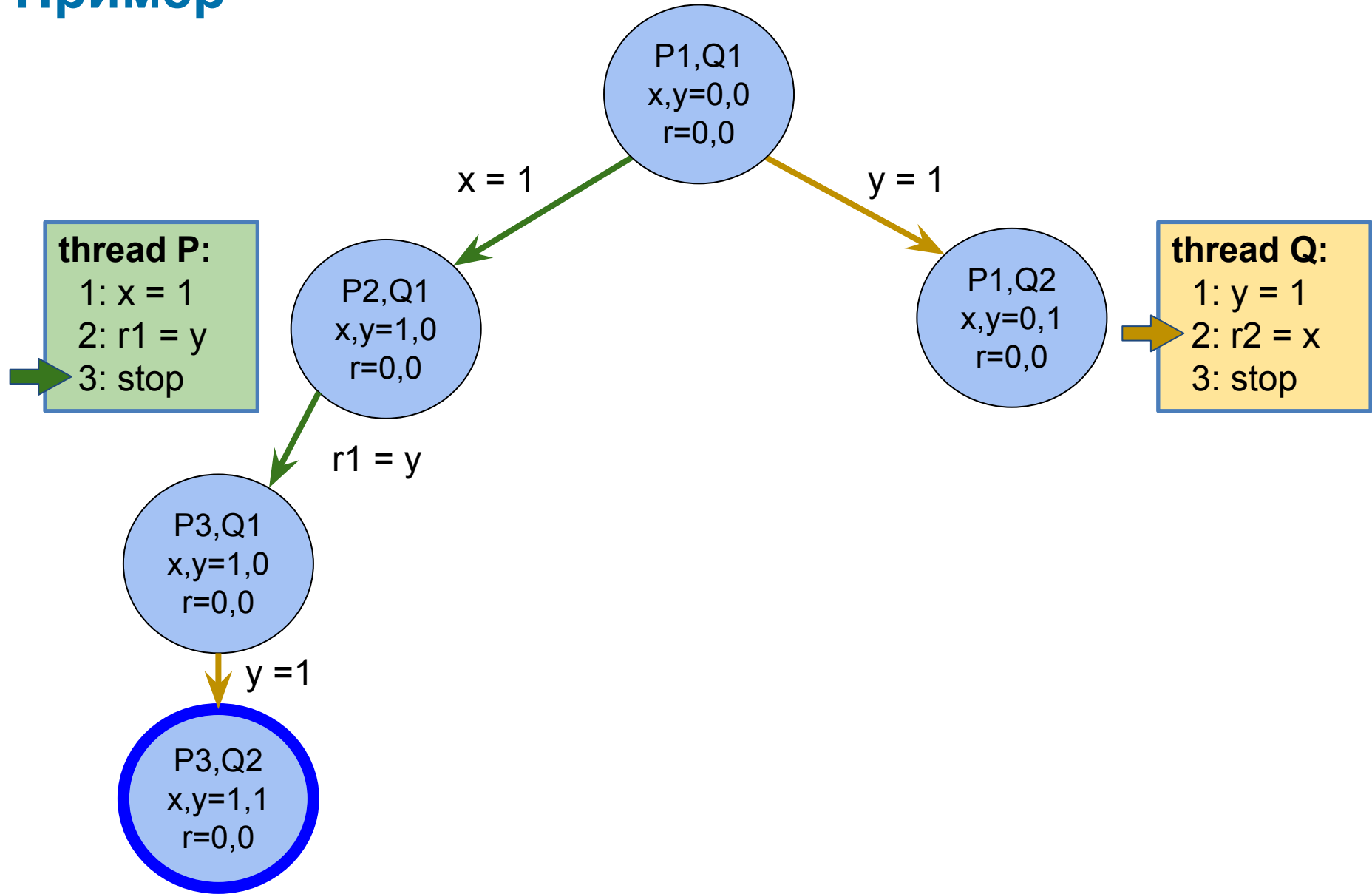


- S это **общее состояние**:
 - Состояние всех потоков (IP+locals)
 - И состояние всех [общих] объектов
- f и g это **операции**
 - Количество возможных операций в каждом состоянии равно количеству потоков
- $f(S)$ это новое состояние после применения операции f в состоянии S

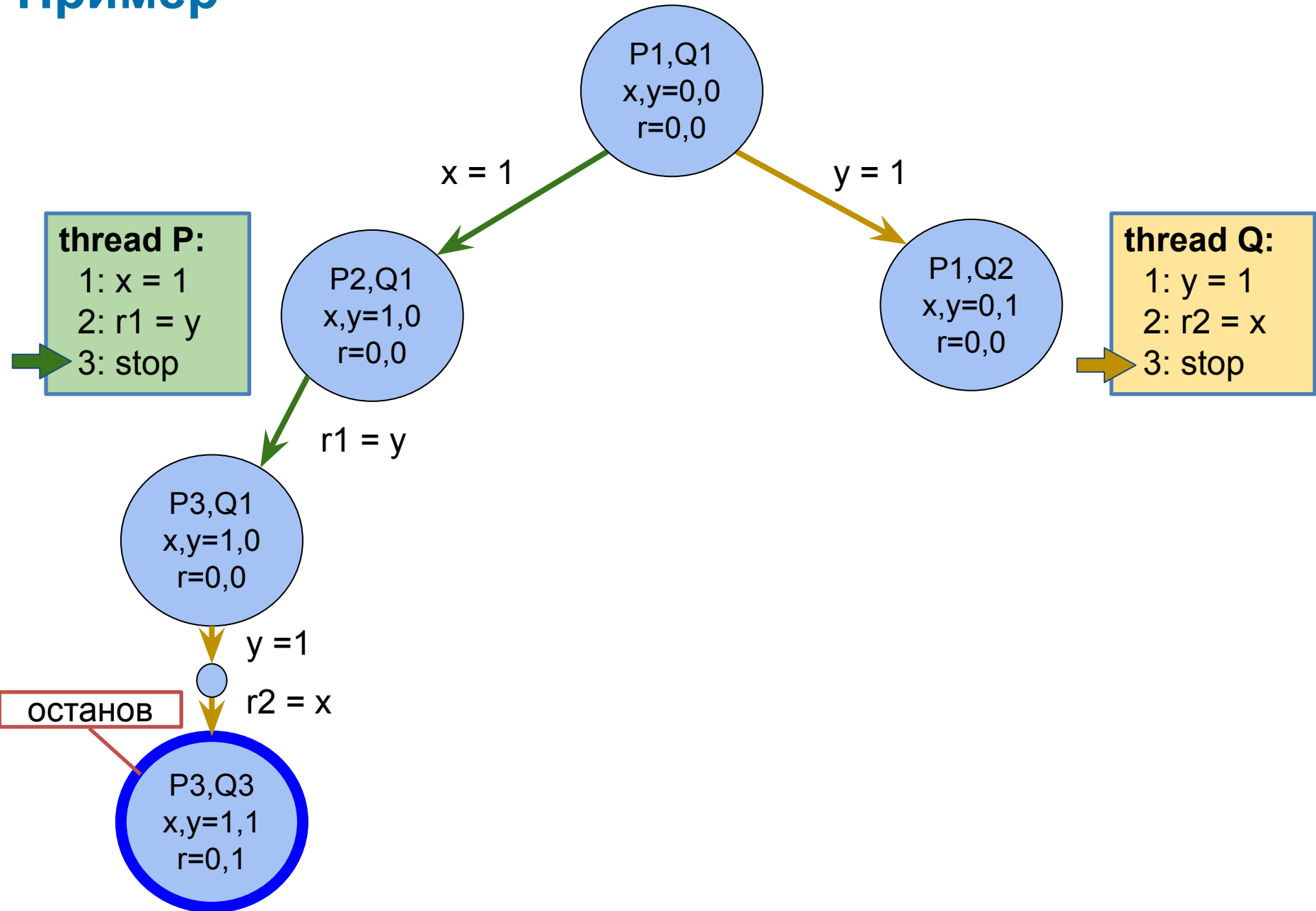
Пример



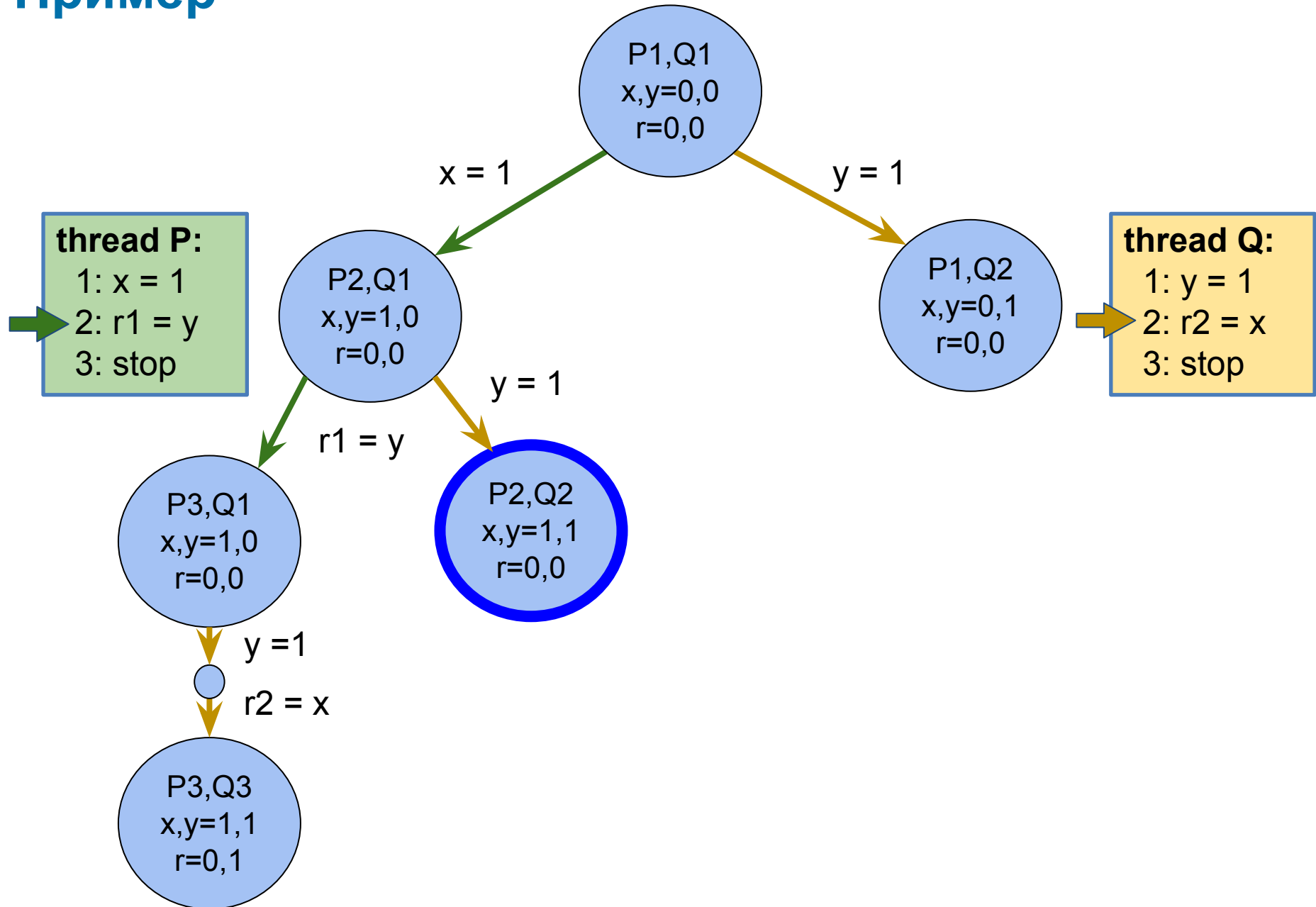
Пример



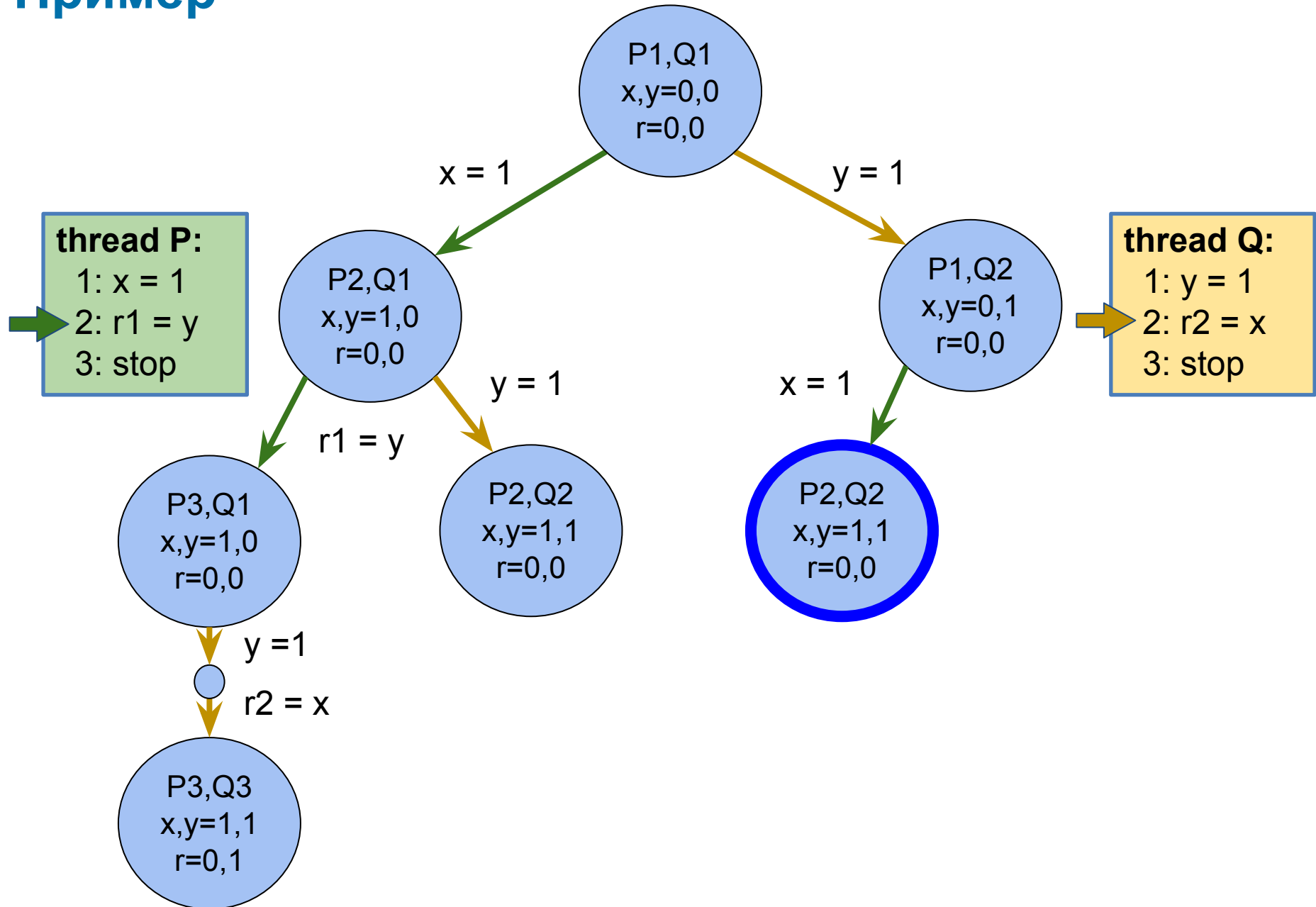
Пример



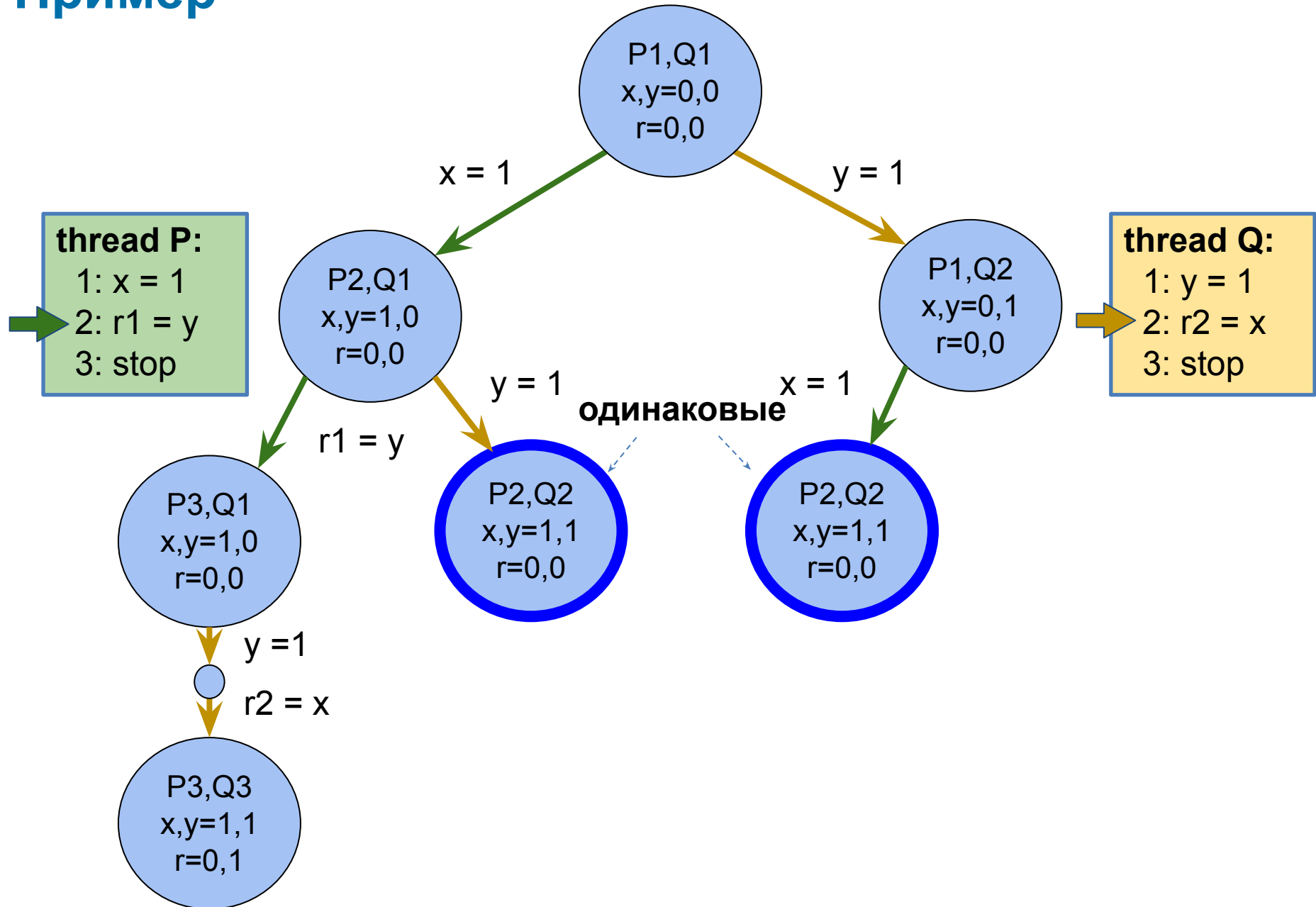
Пример



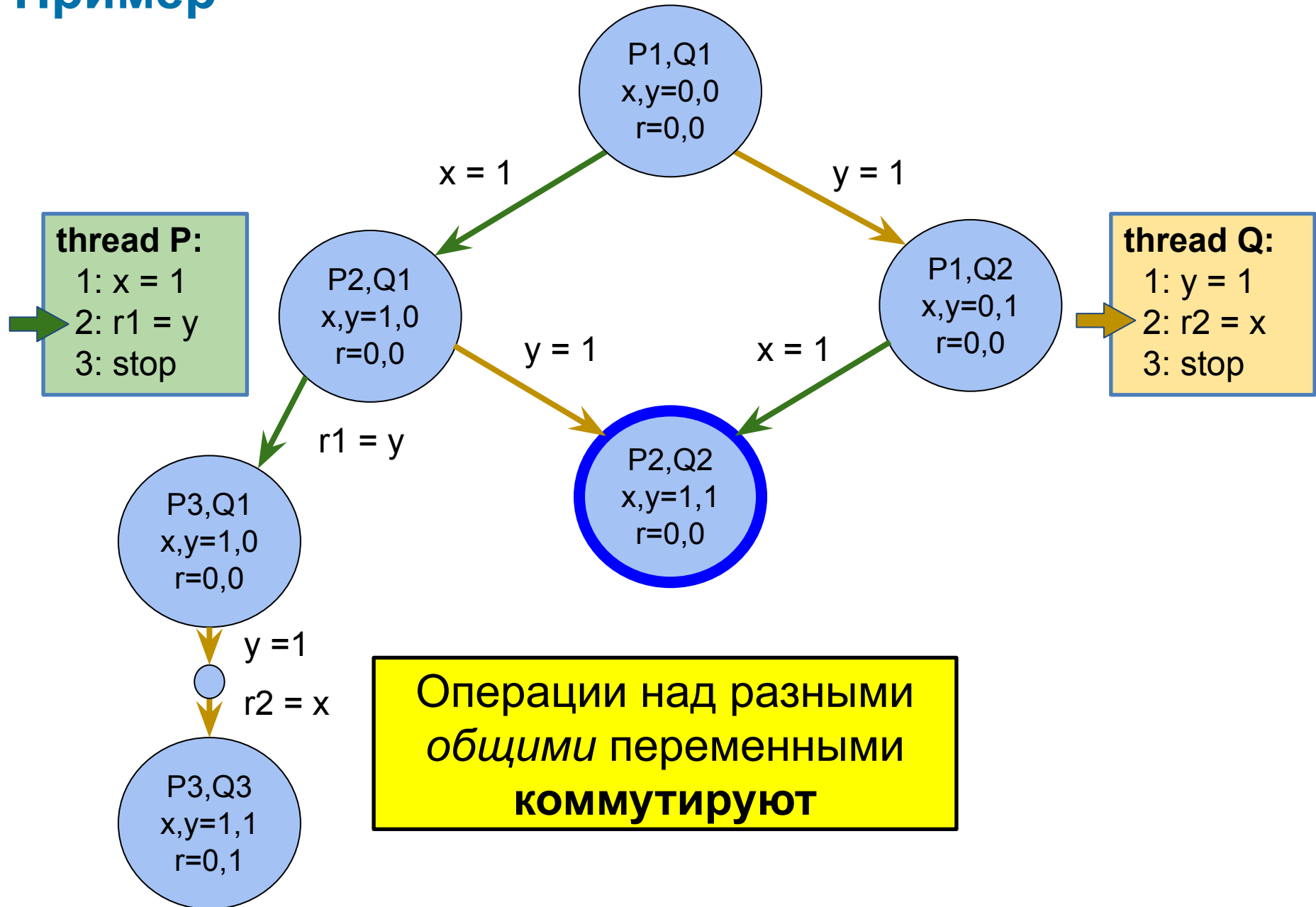
Пример



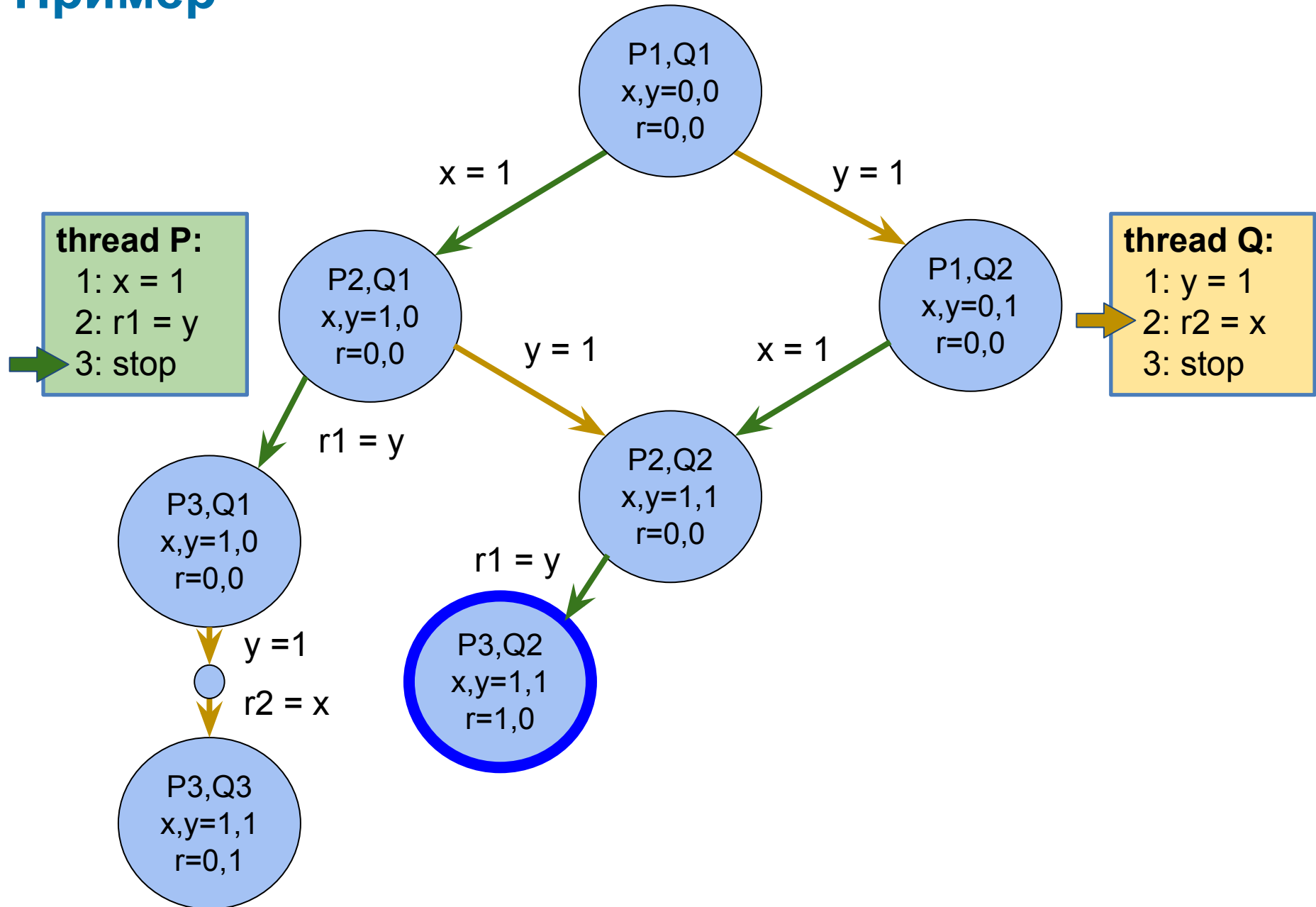
Пример



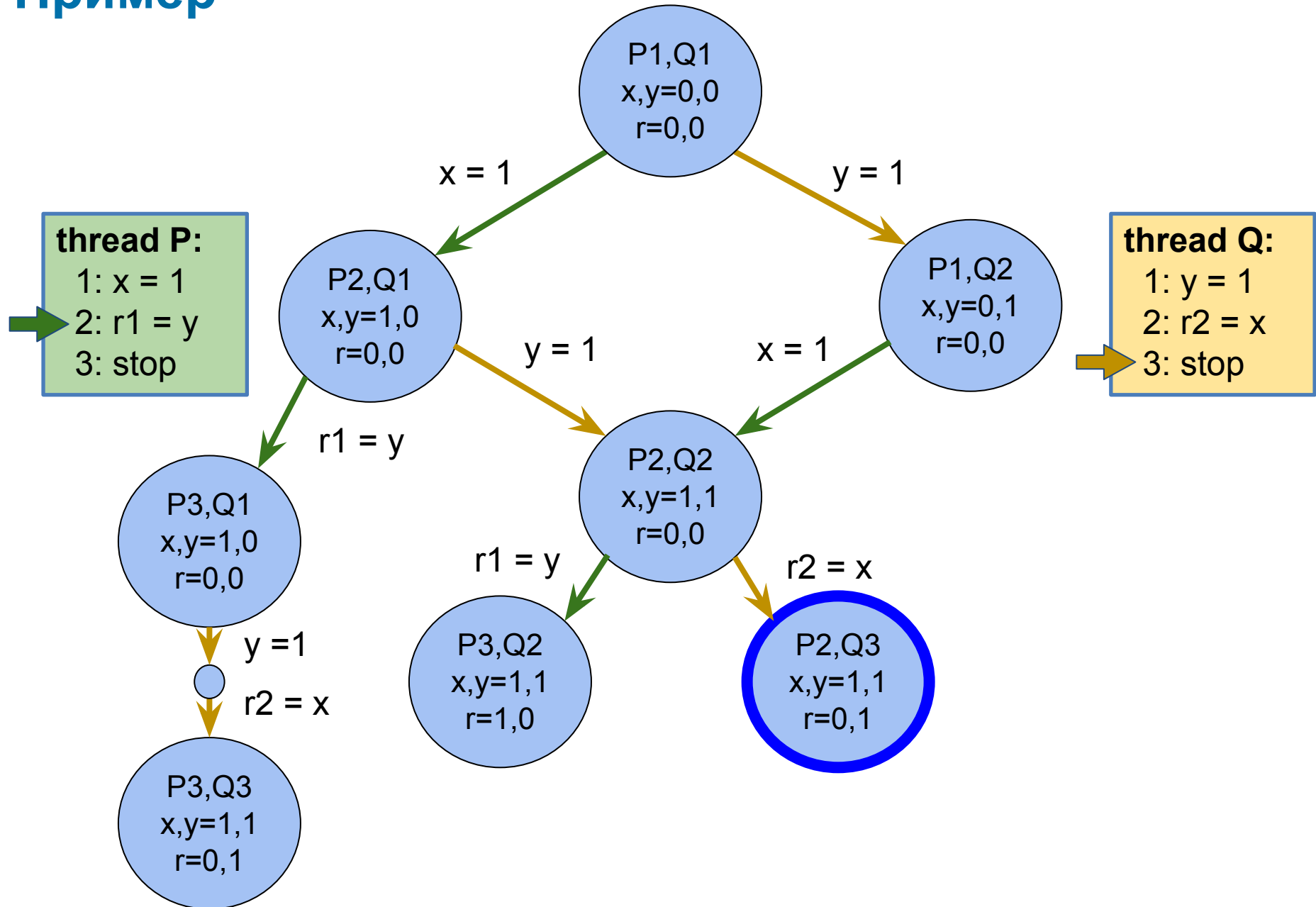
Пример



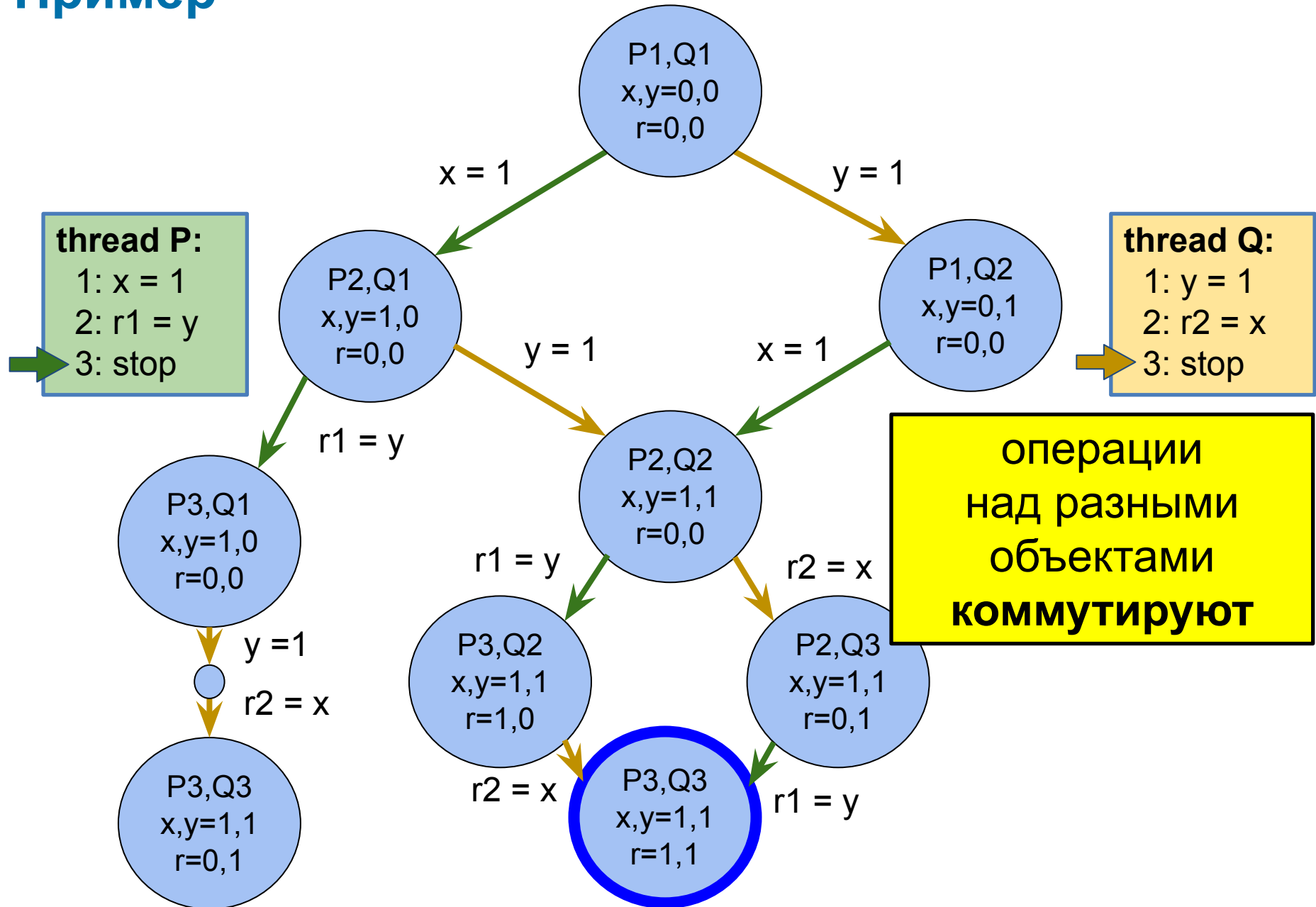
Пример



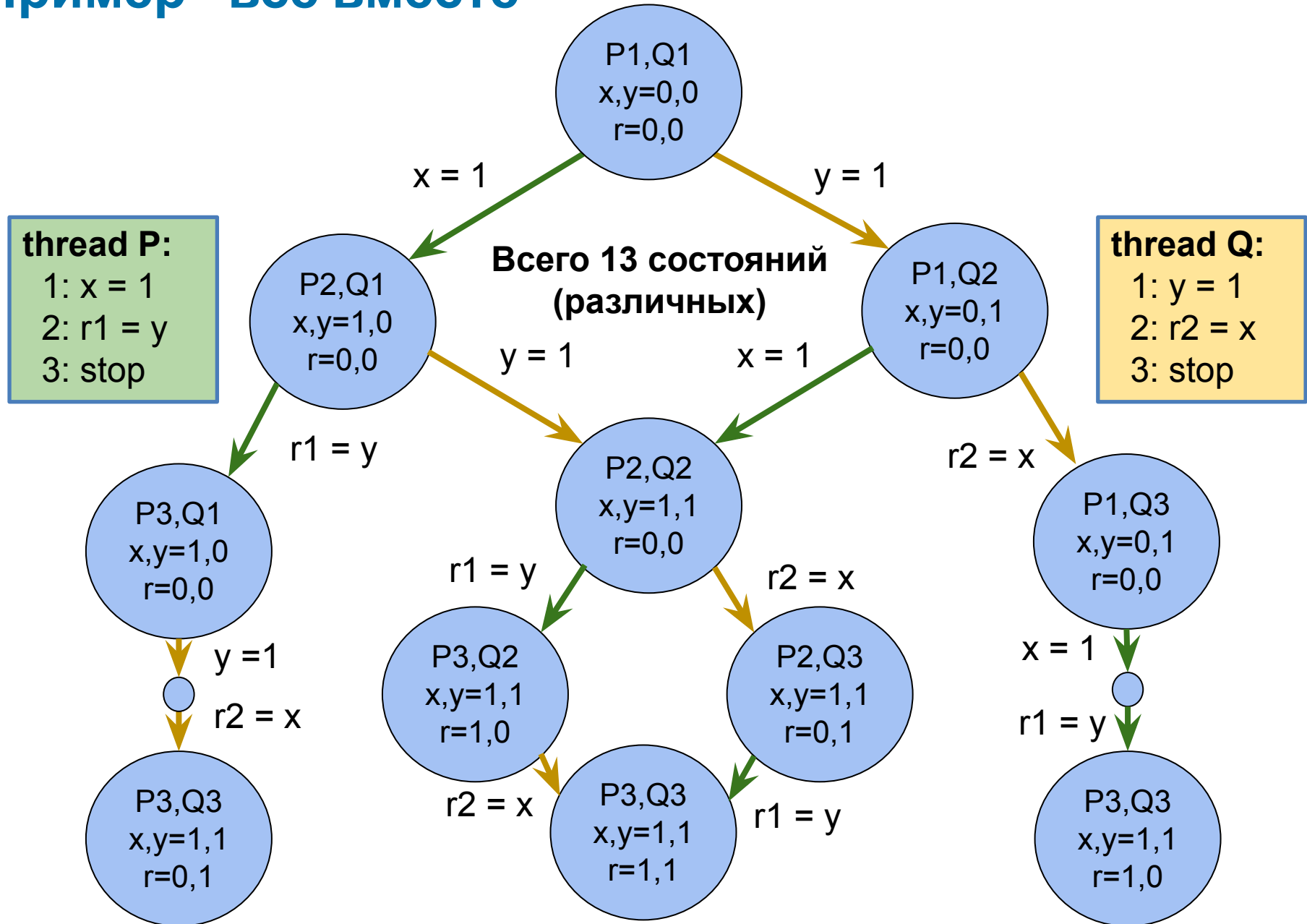
Пример



Пример

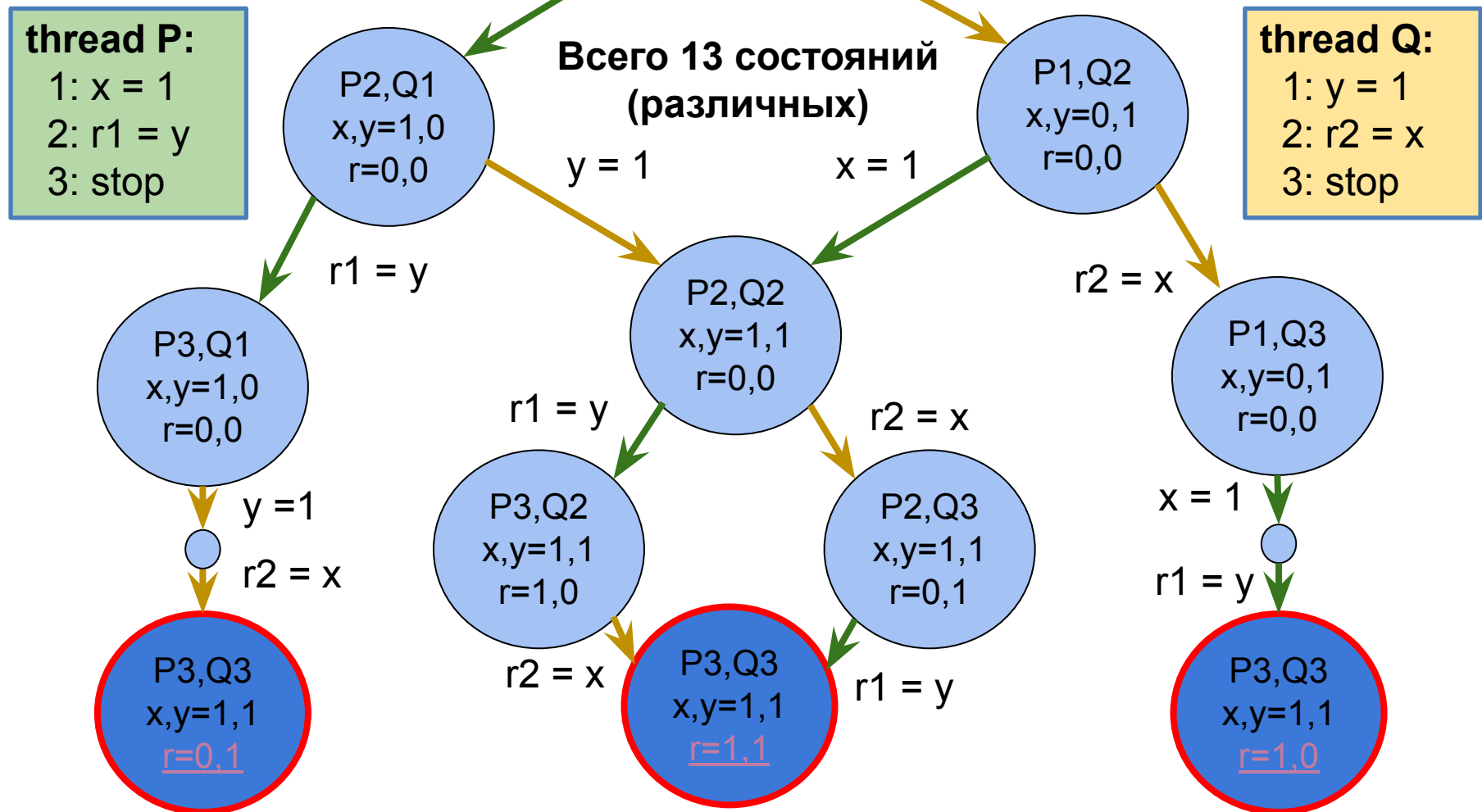


Пример - всё вместе



Пример - результаты

Недетерминизм!



Попробуем на практике (1)

```
@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 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;
    }
}
```

Попробуем на практике (1)

```
@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 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;
    }
}
```

Попробуем на практике (1)

```
@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 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;
    }
}
```

Попробуем на практике (1)

```
@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 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;
    }
}
```

Попробуем на практике (1)

```
@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 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;
    }
}
```

Запустим!

Попробуем на практике (1)

```
@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 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;
    }
}
```

Запустим!

[FAILED]	SimpleTest1
State	Occurrences
0, 0	54,186,890
0, 1	74,036,686
1, 0	53,219,852
1, 1	372

Попробуем на практике (1)

```
@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 SimpleTest1 {
    int x;
    int y;

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

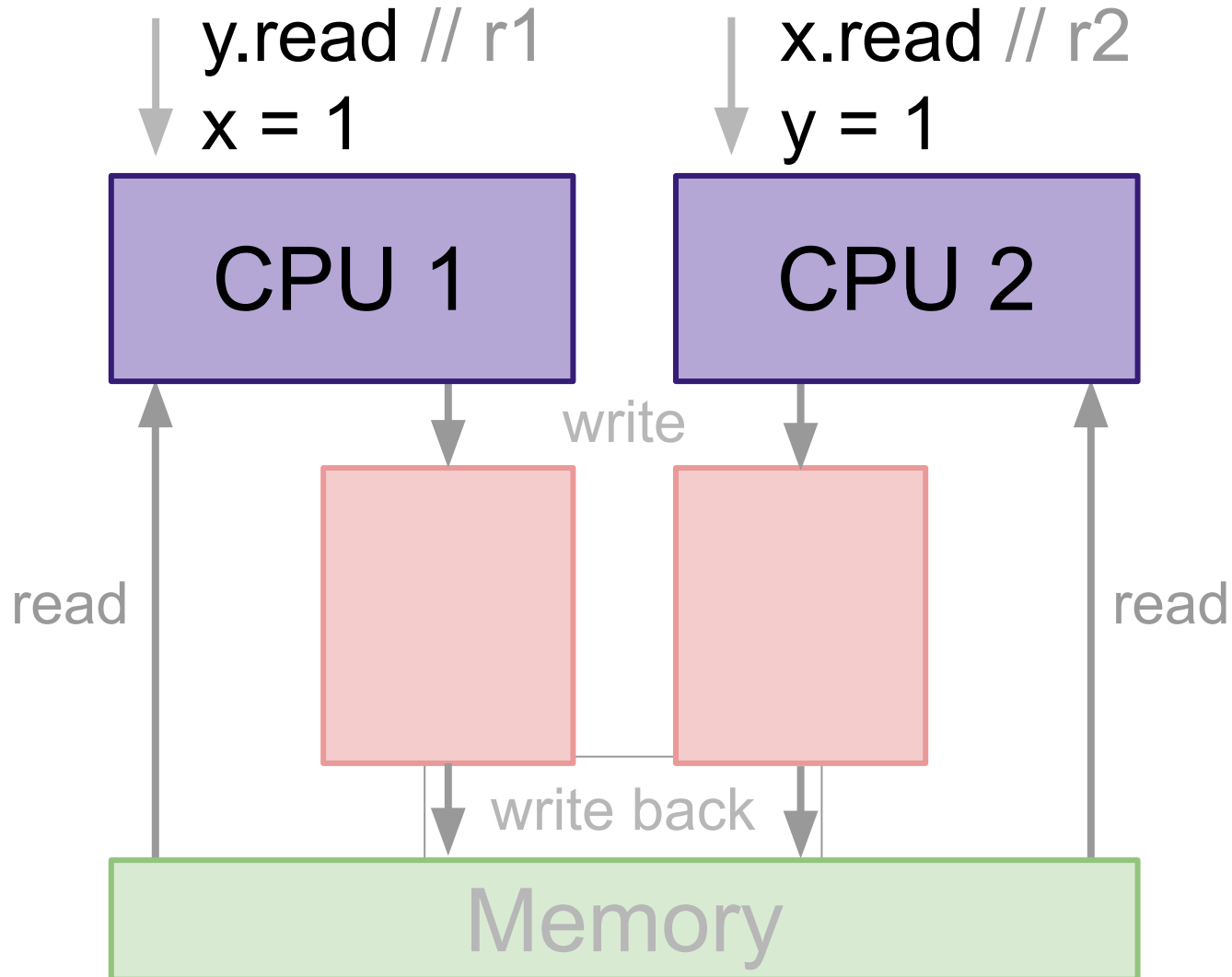
    @Actor
    public void threadQ(IntResult2 r) {
        y = 1;
        r.r2 = x;
    }
}
```

Запустим!

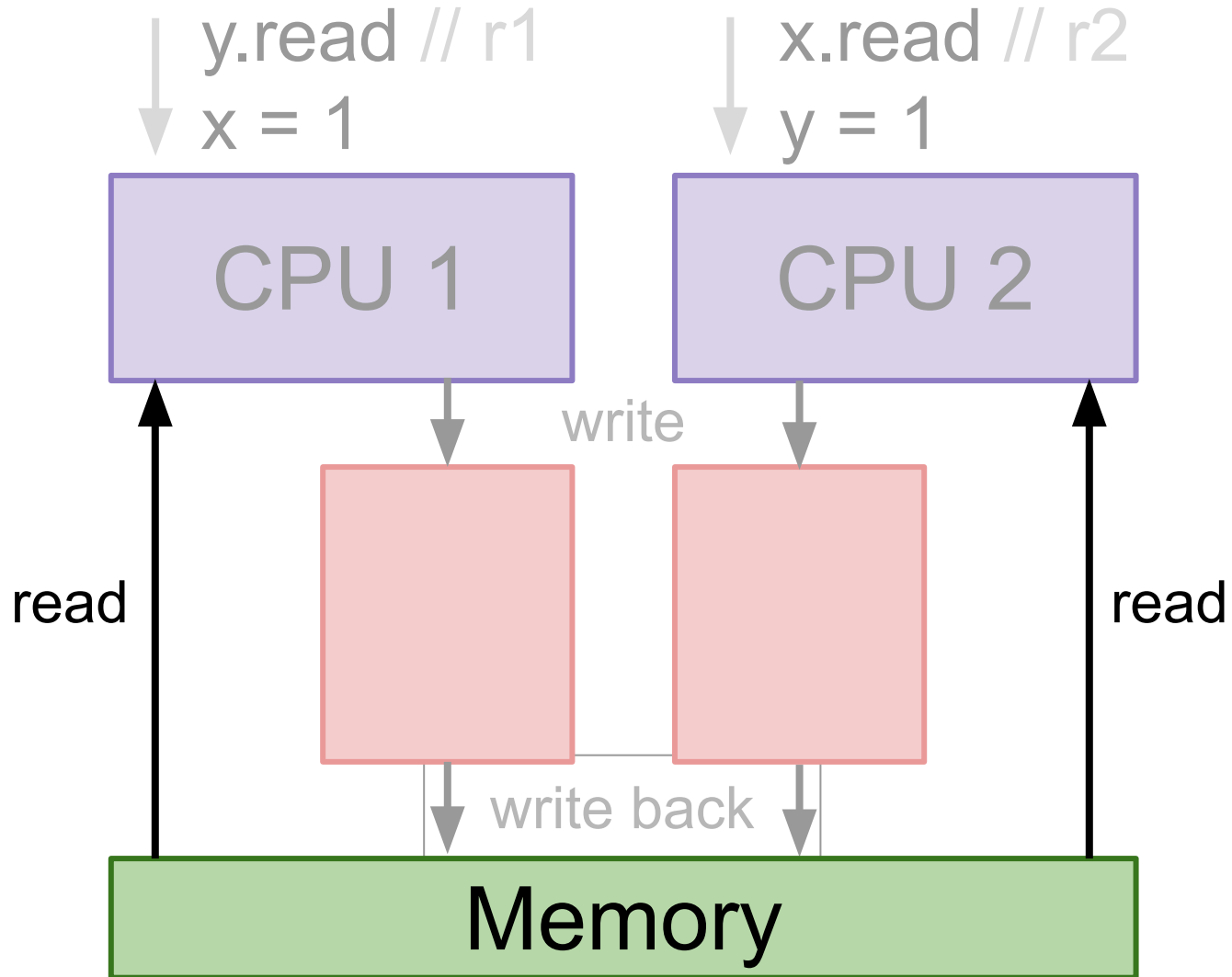
[FAILED]	SimpleTest1
State	Occurrences
0, 0	54,186,890
0, 1	74,036,686
1, 0	53,219,852
1, 1	372

Как же так? Ведь пары $r=0,0$ вообще не должно было быть!

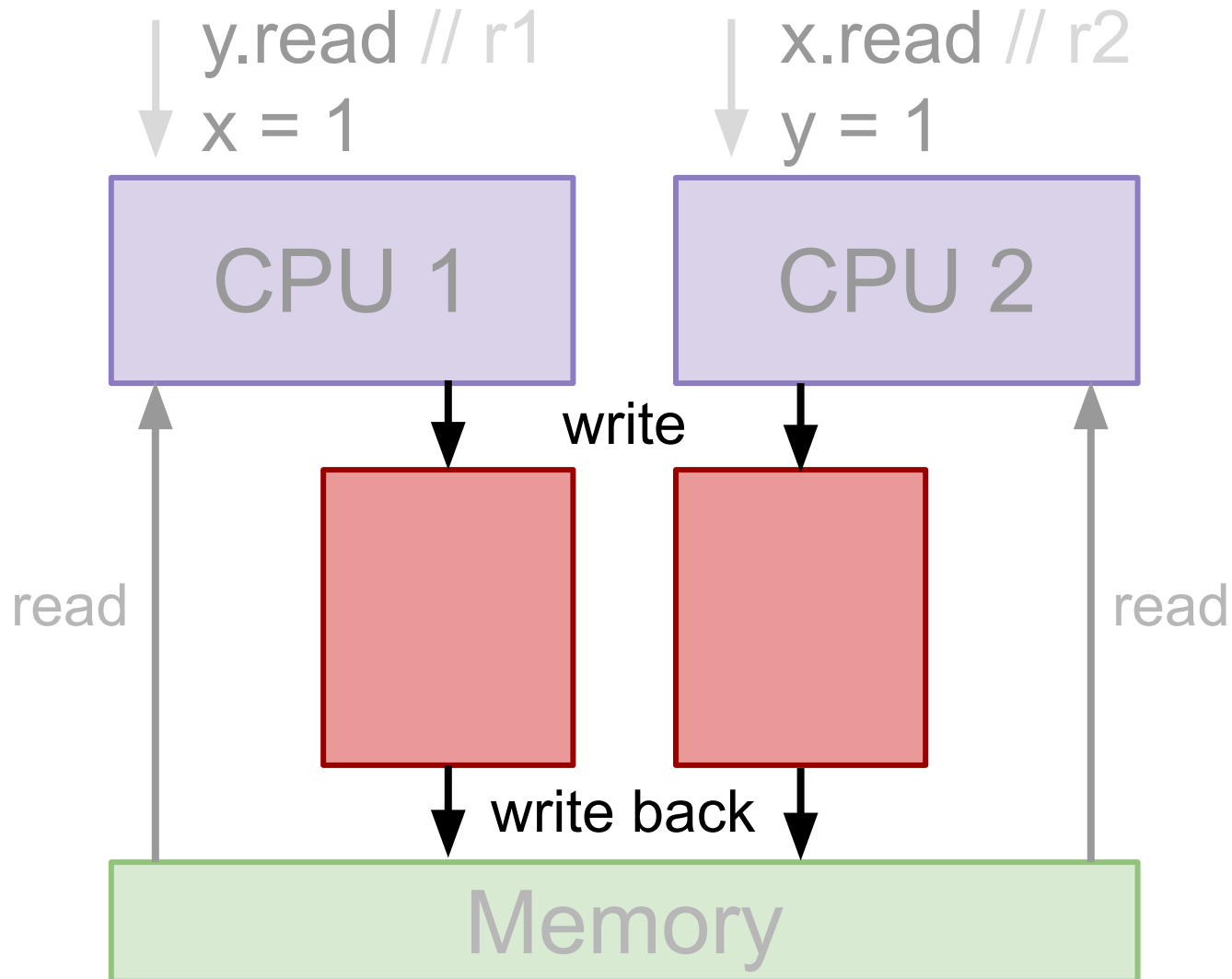
TSO (Total Store Order) модель памяти



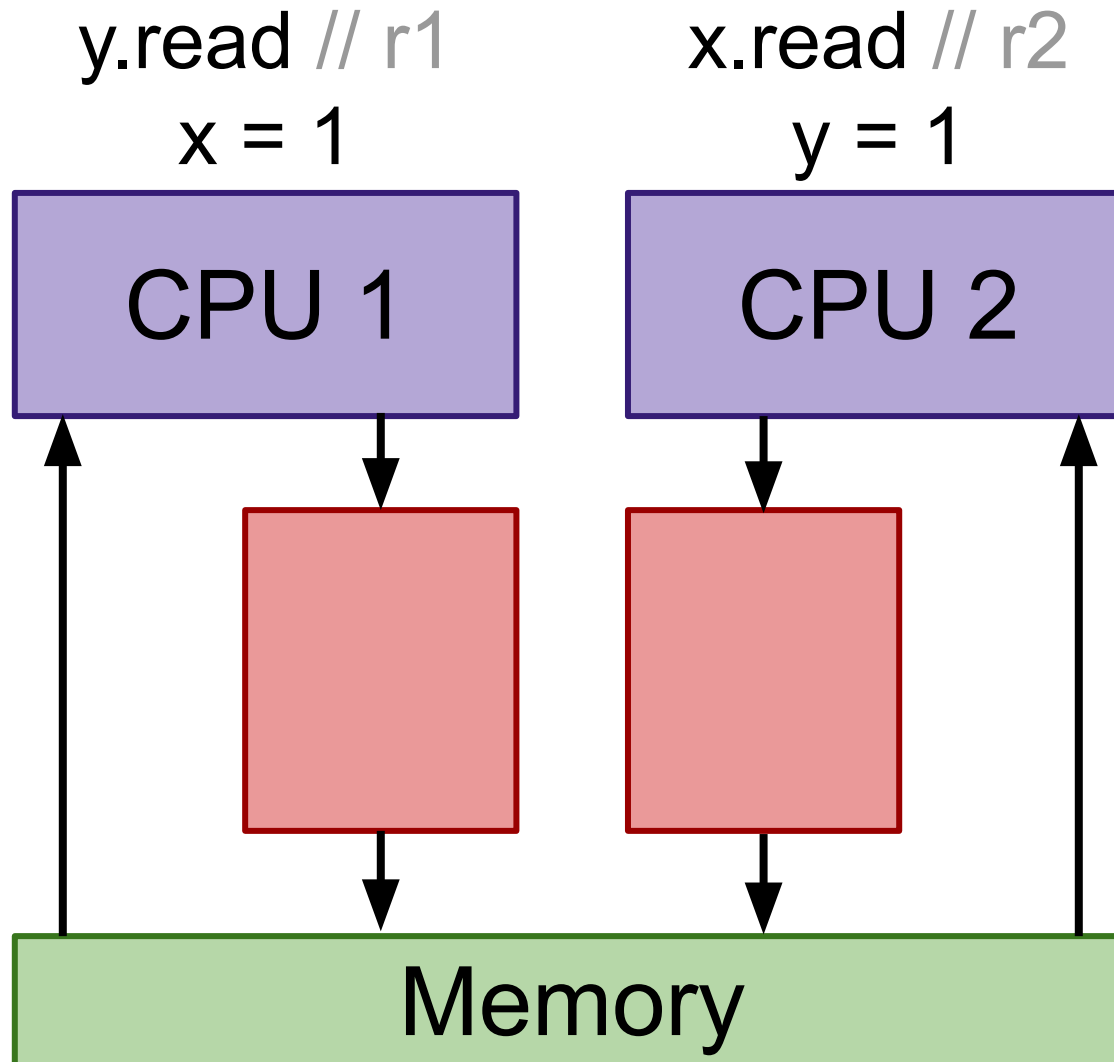
TSO (Total Store Order) модель памяти



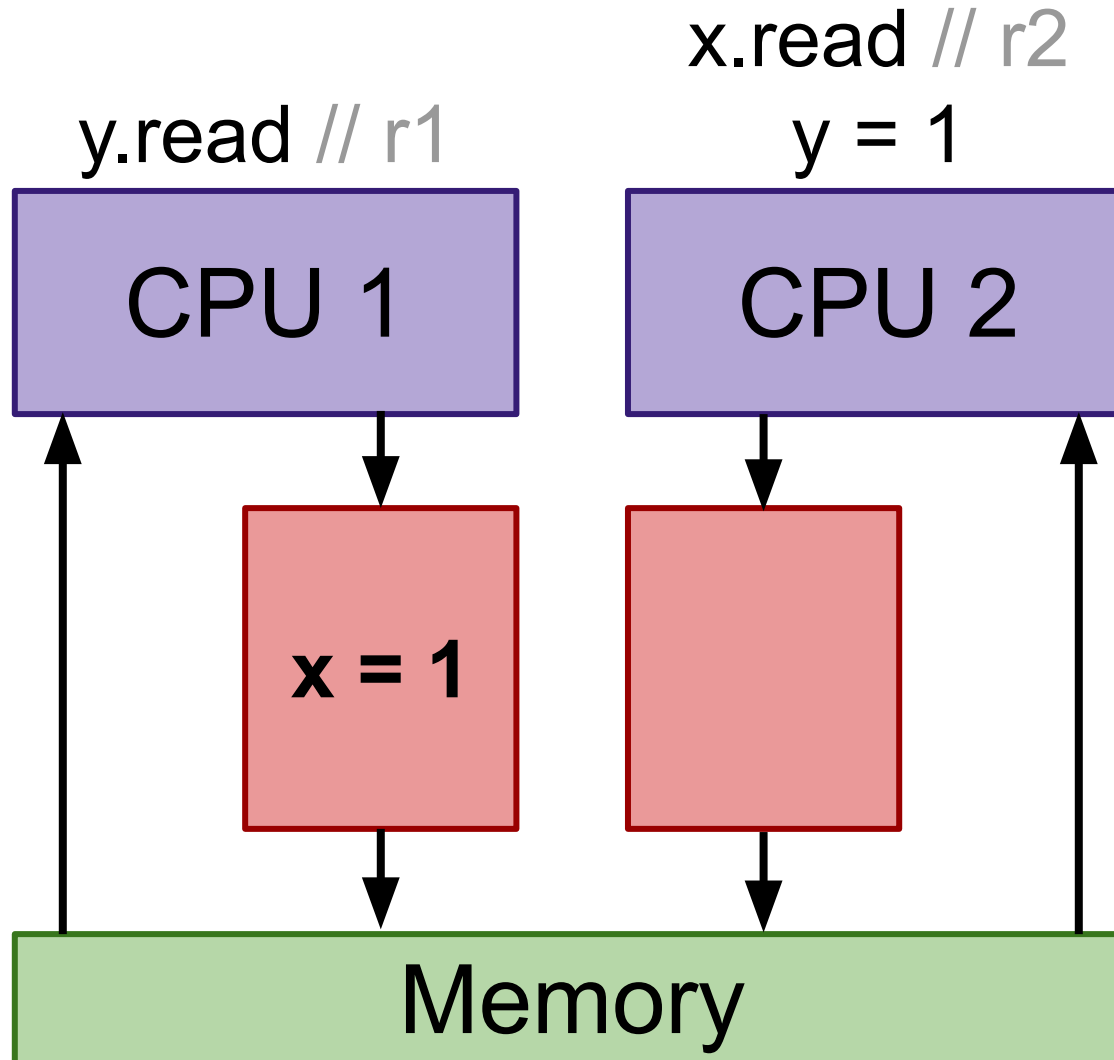
TSO (Total Store Order) модель памяти



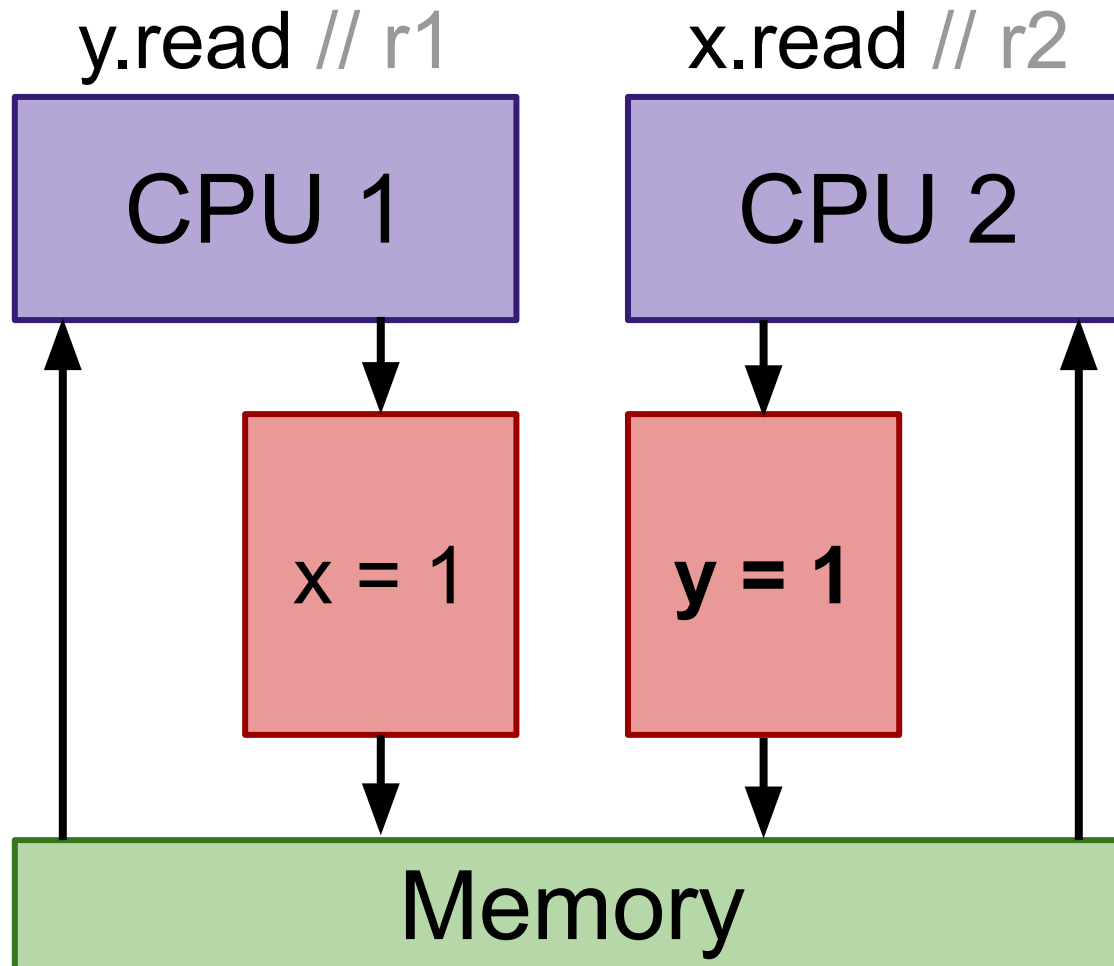
TSO (Total Store Order) модель памяти



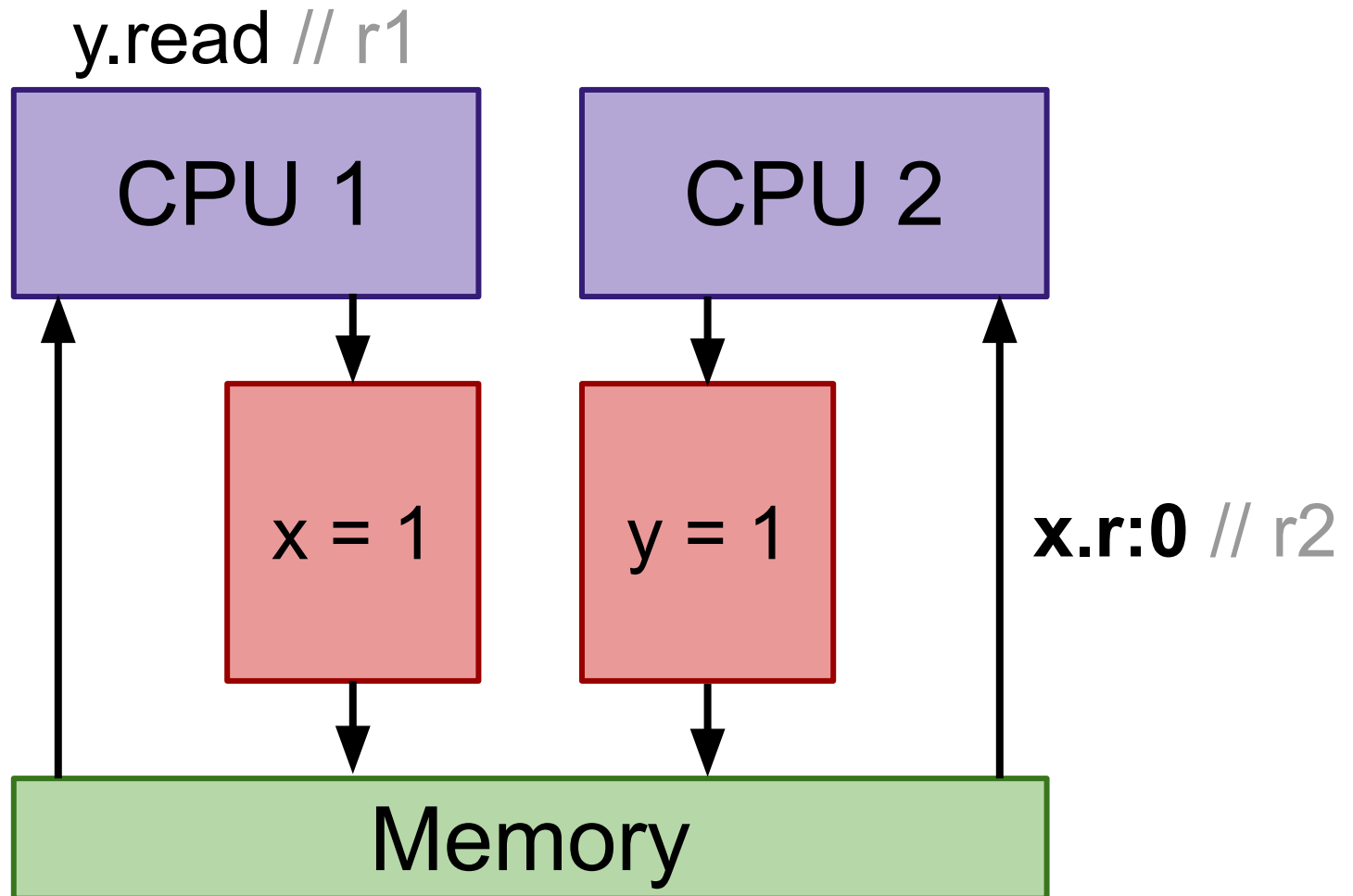
TSO (Total Store Order) модель памяти



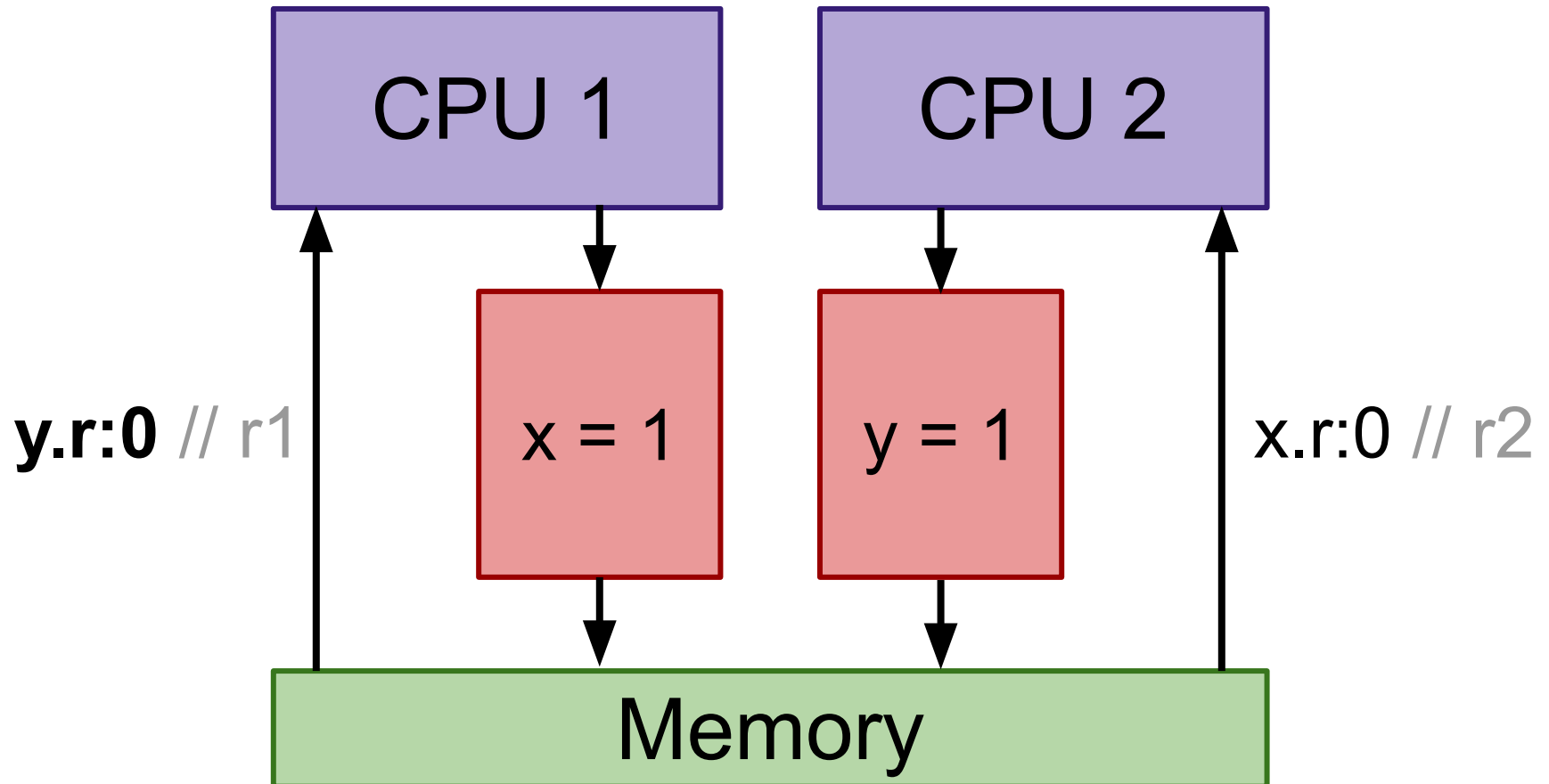
TSO (Total Store Order) модель памяти



TSO (Total Store Order) модель памяти



TSO (Total Store Order) модель памяти

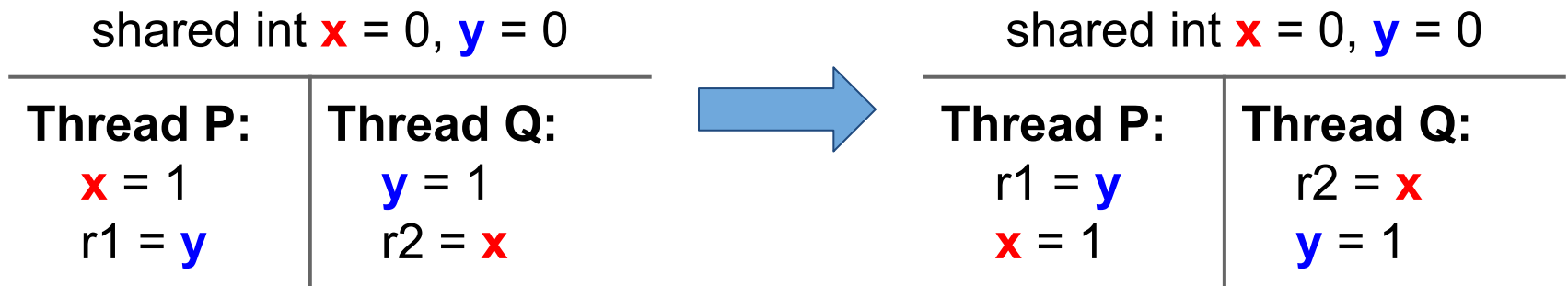


Практическое объяснение

- Запись на современных процессорах не сразу идет в память
 - Буферы записей на x86 (TSO)
 - Еще хуже на ARM
- => Другой поток не увидит эту запись сразу

Практическое объяснение

- Запись на современных процессорах не сразу идет в память
 - Буферы записей на x86 (TSO)
 - Еще хуже на ARM
- => Другой поток не увидит эту запись сразу
- А еще компилятор мог переставить инструкции!



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?

Контрольный вопрос

Какие пары значений x и y возможны после параллельного выполнения следующего кода (P и Q – потоки, x и y общие переменные) в модели чередования

```
shared int x = 0, y = 0
```

thread P:

```
1: if y != 0:  
2:   x = 42  
3: stop
```

thread Q:

```
1: if x != 0:  
2:   y = 42  
3: stop
```

✓ (0, 0)