Многопоточное программирование

Классификация архитектур

- 1966 г. М.Флинн
  - SISD (single instruction stream / single data stream) - одиночный поток команд и одиночный поток данных
  - SIMD (single instruction stream / multiple data stream) - одиночный поток команд и множественный поток данных. Векторные команды
  - MISD (multiple instruction stream / single data stream) - множественный поток команд и одиночный поток данных
  - MIMD (multiple instruction stream / multiple data stream) - множественный поток команд и множественный поток данных

Процессы и потоки

- Объект исполнения прикладной программы операционной системой представляется двумя понятиями
  - *Процесс*
  - *Поток*

Процесс

- *Процесс (process)-* экземпляр программы, загруженной в память
- Процессу ОС выделяет ресурсы, необходимые для выполнения программы, например:
  - адресное пространство процесса содержит его программный код, данные и стек (или стеки)
  - файлы используются процессом для чтения входных данных и записи выходных
  - устройства ввода-вывода используются в соответствии с их назначением
- Процесс – пассивный объект–владелец ресурсов, контейнер для выполнения *потоков (thread)*
- Процесс может иметь несколько потоков. Количество потоков может меняться во время выполнения процесса.

Поток / нить (thread)

- *Поток* - абстракция, представляющая последовательное выполнение команд программы, развертывающееся во времени
- Выполняются не процессы, а именно потоки
- Любой процесс имеет хотя бы один поток

Поток / нить (thread)

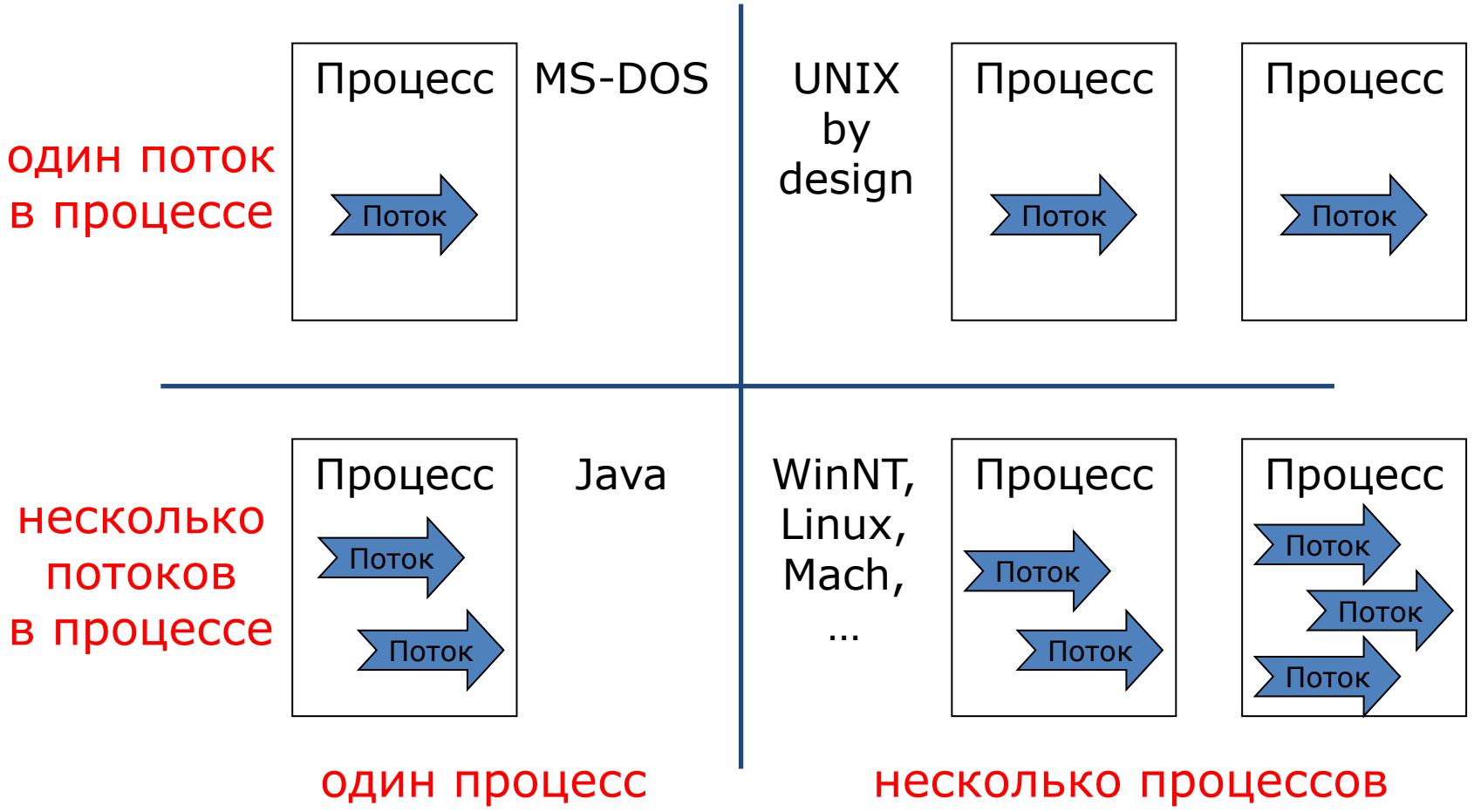
Потоки ***совместно используют***

- глобальные и статические переменные (располагаются в регионе данных)
- динамически распределяемую память (кучу)
- системные ресурсы, выделенные процессу

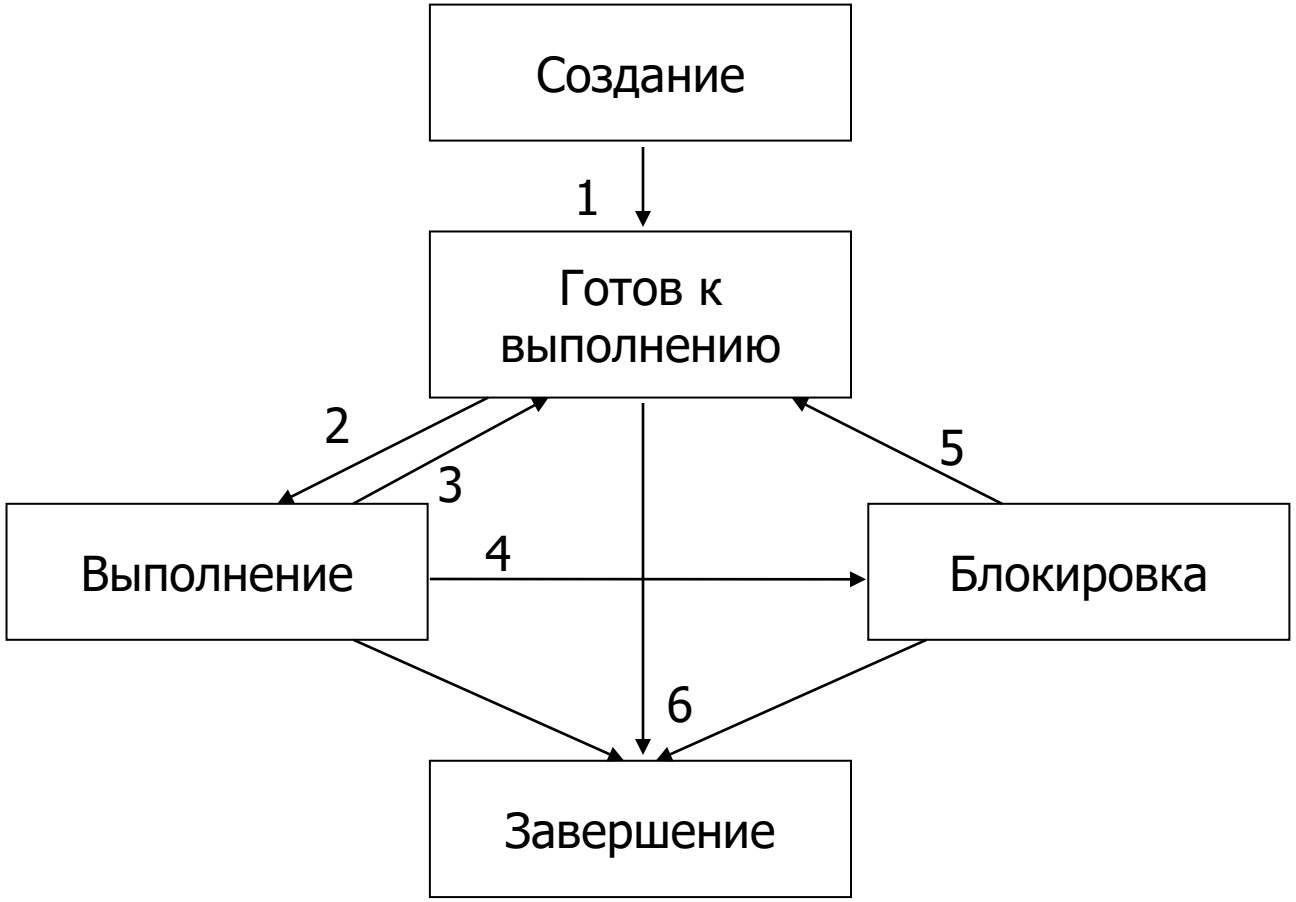
Каждый поток имеет ***свои собственные***

- программный счетчик (IP)
- значения регистров
- локальные переменные (т.е. свой собственный стек)

Процессы и потоки



Состояния потока



Состояния потока

- ***Выполнение*** – состояние работающего потока – обладающего всеми необходимыми ресурсами, в том числе возможностью использования ЦП
- ***Готов к выполнению*** – поток обладает всеми необходимыми для выполнения ресурсами за исключением ресурса «время ЦП»
- ***Ожидание*** (сон, блокировка) – выполнение потока заблокировано до наступления некоторого внешнего события (например, поступления входных данных или освобождения ресурса)

Состояния потока

(1) – главный поток приложения запускается исполняющей системой (***загрузчиком***). Остальные потоки запускаются каким-то из выполняющихся потоков

(2,3) - осуществляются ядром операционной системы (***планировщиком***)

(5) - производится ***планировщиком*** в момент выполнения условия ожидания

(4) продолжение работы невозможно, если

- для продолжения работы требуется наступление какого-либо события
- поток затребовал недоступный в данный момент ресурс
- поток переводится в состояние ожидания ***планировщиком*** во время обработки системного вызова
- поток заблокирован внешним по отношению к нему вызовом

График выполнения потока

Один поток – один процессор

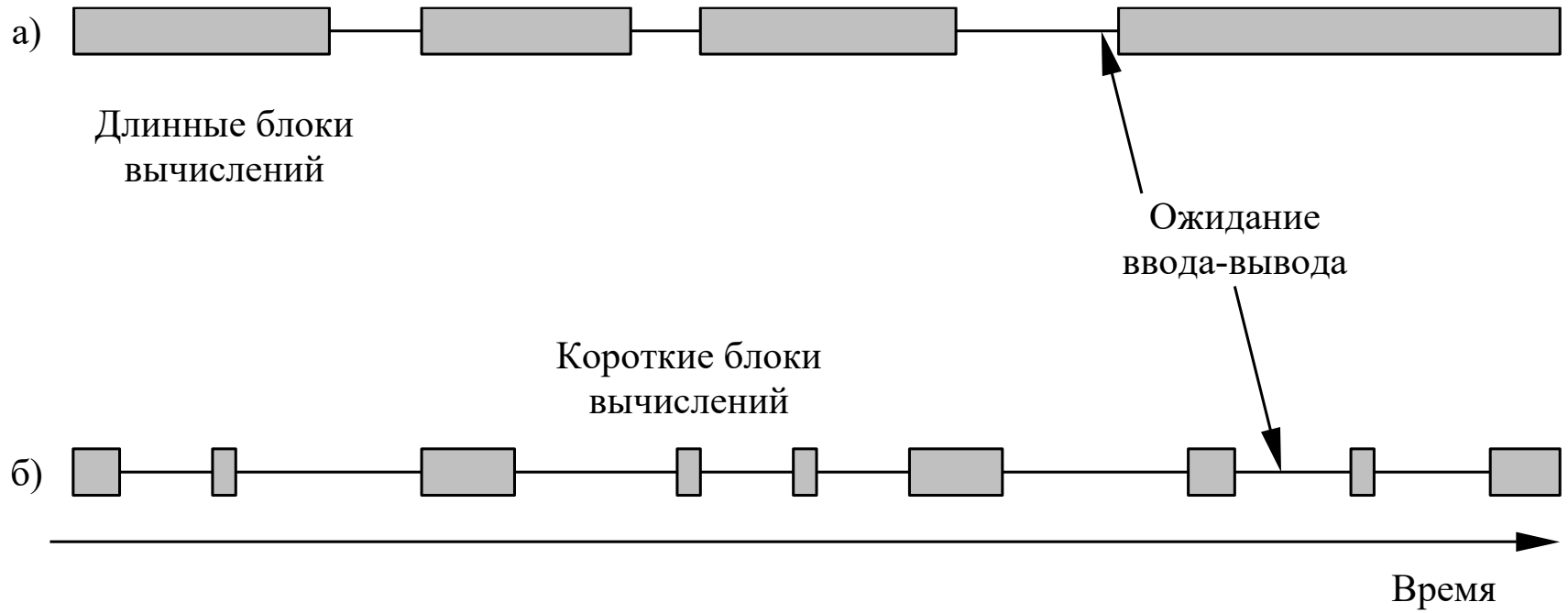


График выполнения потока

Несколько потоков – один процессор

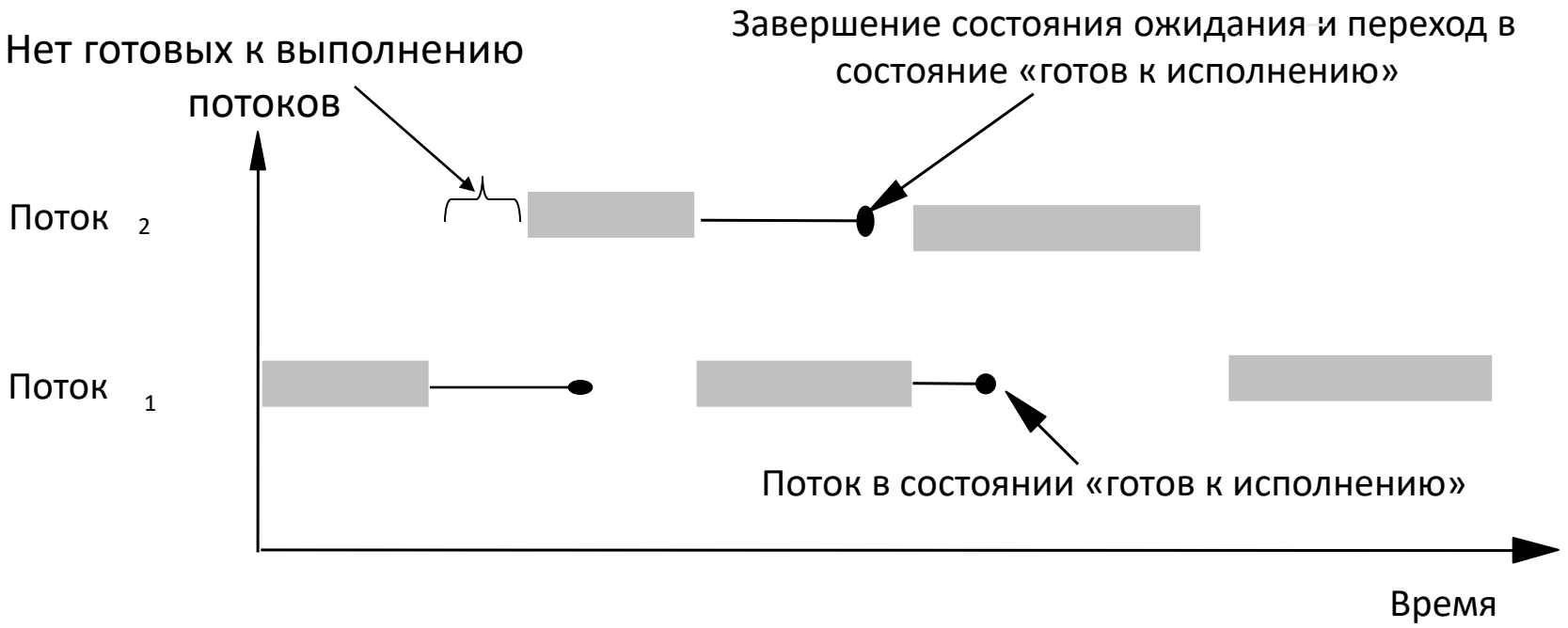
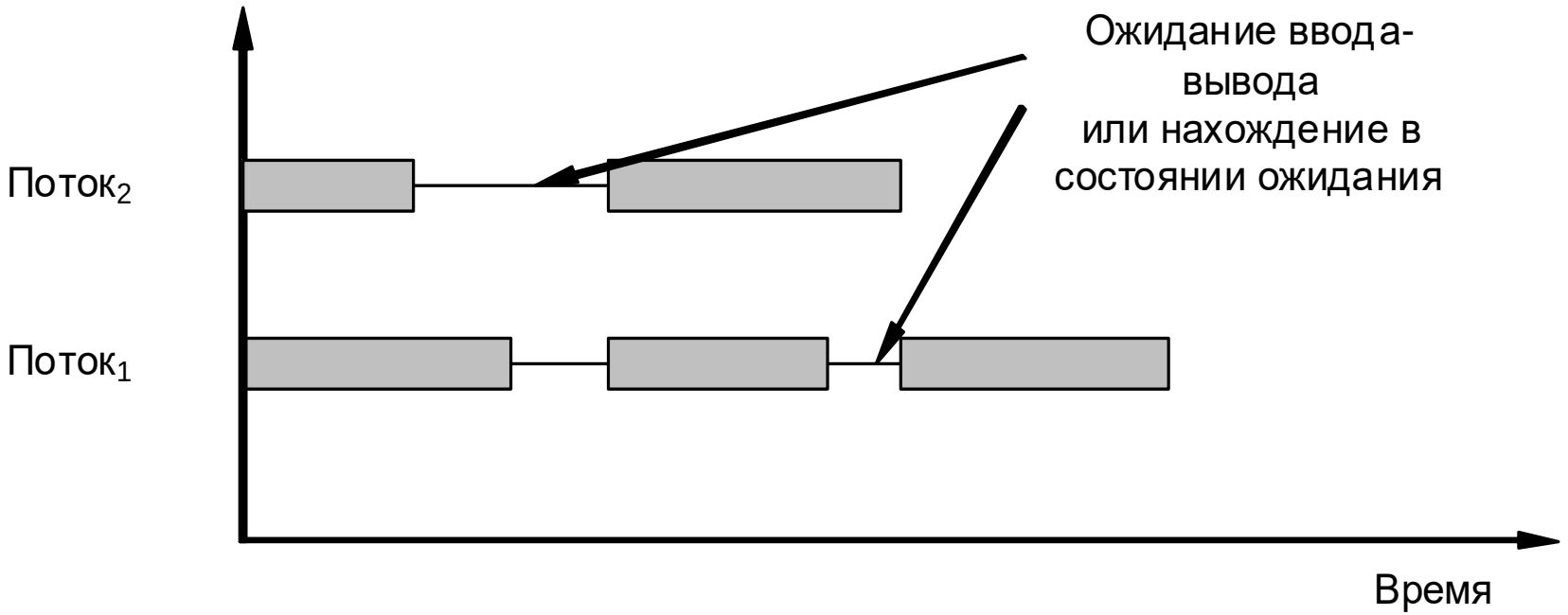


График выполнения потока

Несколько потоков – несколько процессоров



Синхронизация потоков

Все потоки, принадлежащие одному процессу, разделяют некоторые общие ресурсы (адресное пространство ОП, открытые файлы).

Что произойдет, если один поток еще не закончил работать с каким-либо общим ресурсом, а система переключилась на другой поток, использующий тот же ресурс?

Синхронизация потоков

Поток A

x = x+3

Поток B

x = x + 5

---

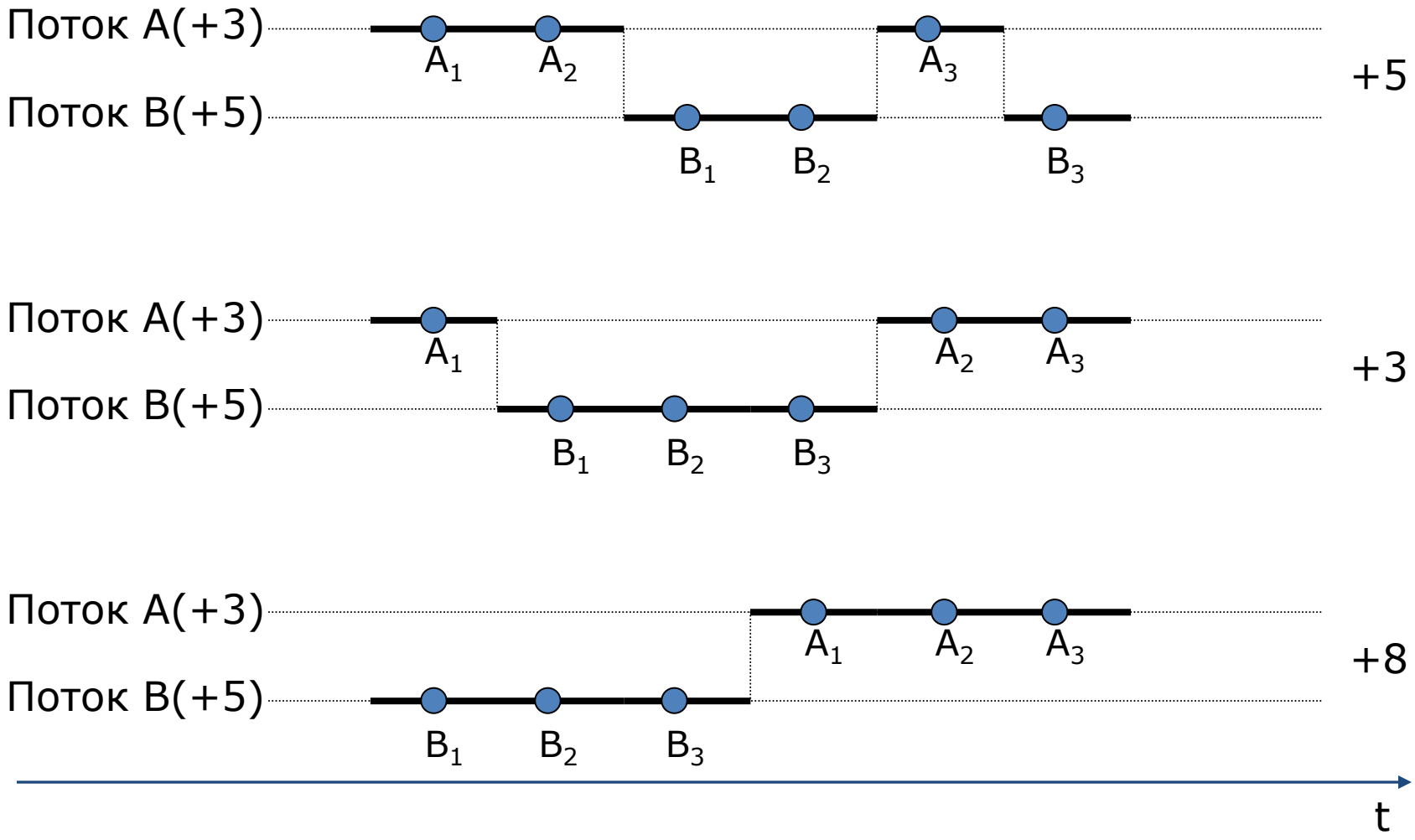
x = x+a 

(1) Прочитать *x*

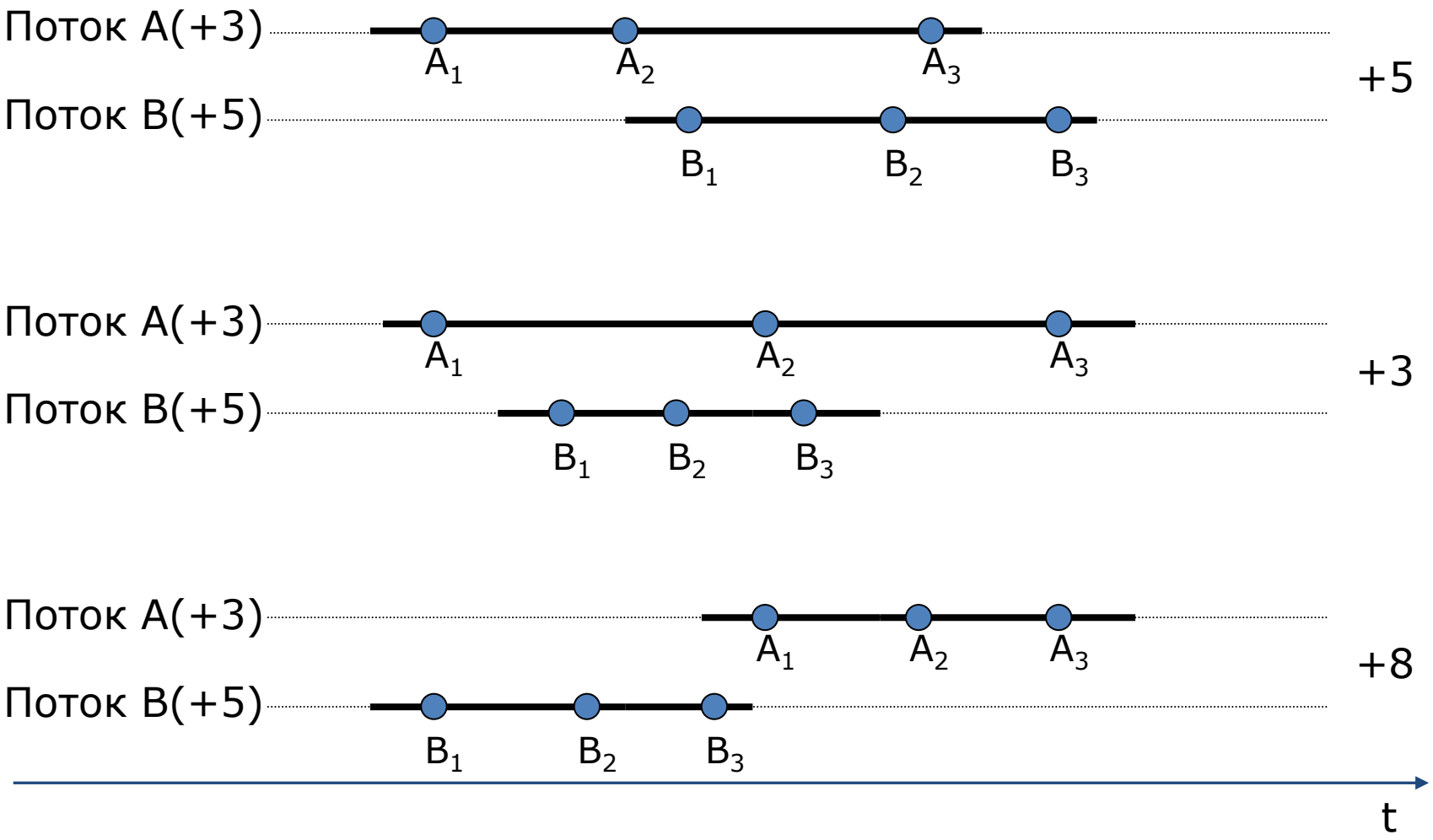
(2) Увеличить на *a*

(3) Записать в *x*

Синхронизация потоков



Синхронизация потоков



Синхронизация потоков

- Сложность проблемы синхронизации кроется в нерегулярности возникающих ситуаций – все определяется взаимными скоростями потоков и моментами их прерываний
- Ситуации, когда два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков, называются **гонками**

Критическая секция

- Критическая секция - это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение данной части еще не завершено

- Критическая секция всегда определяется по отношению к определенным критическим данным, при несогласованном изменении которых могут возникнуть нежелательные эффекты

- Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один поток

Синхронизация потоков

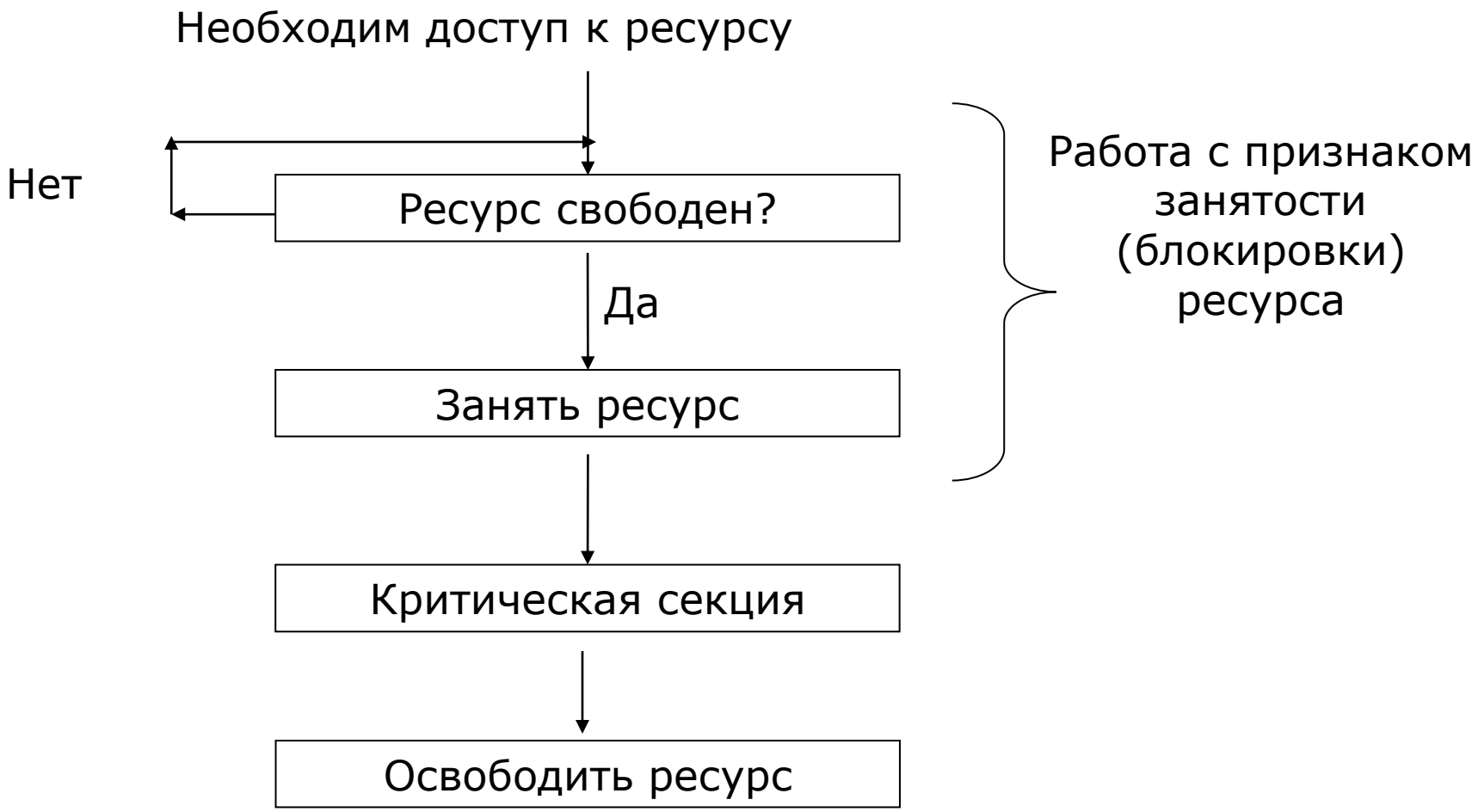
Механизм, позволяющий потокам согласовывать свою работу с общими ресурсами называется ***механизмом синхронизации***

Механизм синхронизации использует набор ***объектов*** операционной системы, которые создаются и управляются программно, являются общими для всех потоков и используются для координирования доступа к ресурсам

Основные объекты синхронизации

- Взаимоисключение (mutex)
- Критическая секция (critical section)
- Событие (event)
- Монитор (monitor)
- Семафор (semaphore)

Синхронизация потоков



Тупиковые ситуации

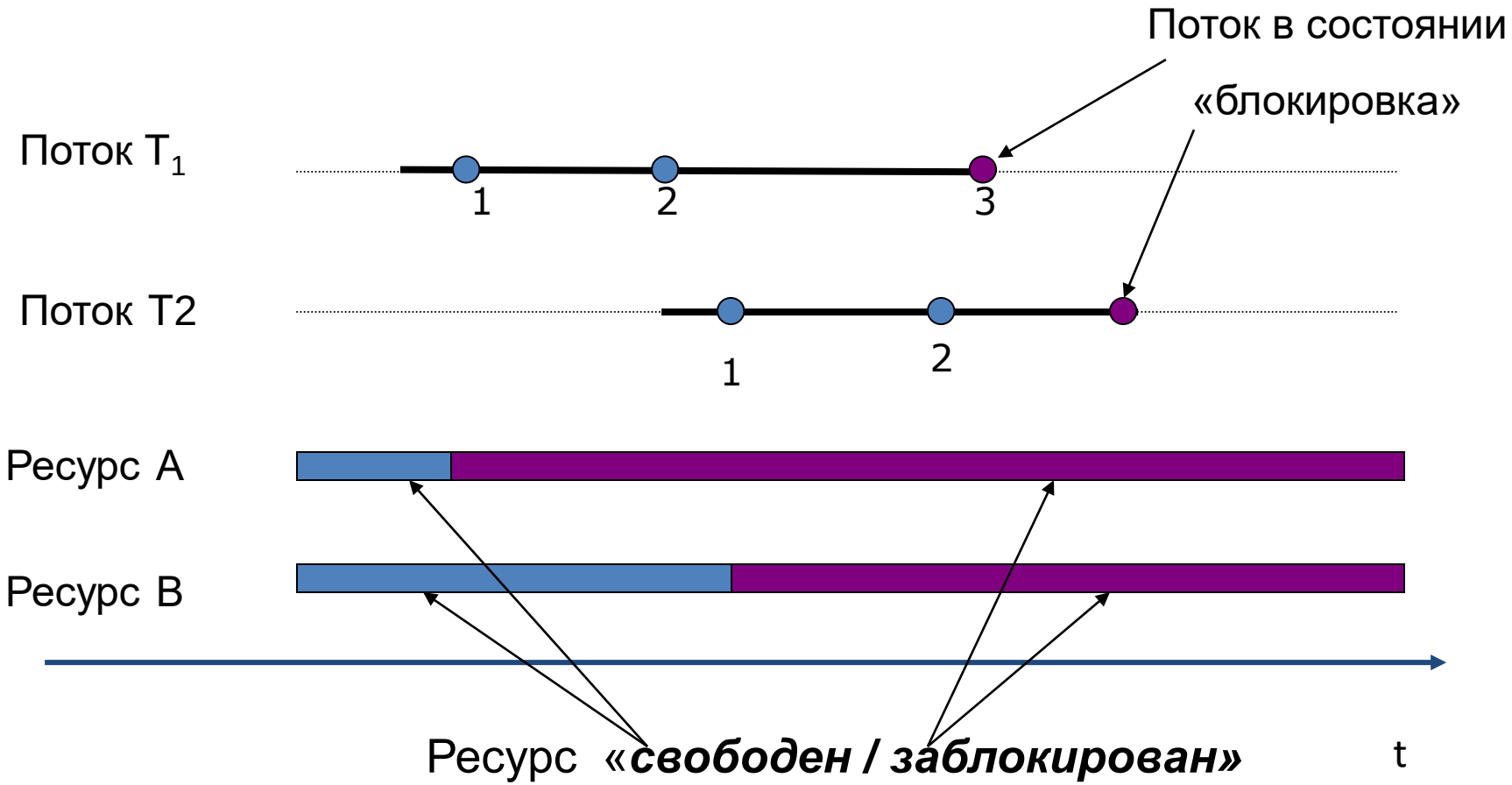
Поток T_1

- заблокировать A
- …
- заблокировать B
- …
- освободить A
- освободить B

Поток T_2

- заблокировать B
- …
- заблокировать A
- …
- освободить B
- освободить A

Тупиковые ситуации



Создание потоков в Java

- **Интерфейс Runnable**

```
class MyRunnable
    implements Runnable {
  public void run(){
    код
   }
}
```

- **Наследовать класс Thread**

```
class MyThread
    extends Thread {
    public void run(){
          код
    }
}
```

Создание собственного потока

- Создать экземпляр класса с интерфейсом Runnable и поместить его в поток, поставить поток в очередь на выполнение

```
Runnable r = new MyRunnable();
Thread t = new Thread ( r );
t.start();
```

- Или создать класс потока и поставить его в очередь

```
Tread t1 = new MyThread();
t1.start();
```

Важно !!!

- Если вызвать метод run() непосредственно, то выполнение будет происходить в том же потоке. Новый поток сформирован не будет
- Для формирования нового потока предназначен метод start() класса Thread

Прерывание потоков

- Метод stop() является запрещенным, т.к. он вызывает немедленное завершение потока и может привести к повреждению объектов, с которыми работал поток
- Вместо остановки следует использовать механизм прерывания потока
- Метод interrupt() устанавливает статус прерывания потока

Прерывание потоков

- Для корректной работы поток в процессе выполнения должен опрашивать статус прерывания

```
while (!Thread.currentThread().isInterrupted()  && . . .)
{
  код
}
```

Примеры

- Threads.java
- SimpleThread.java
- MyRunnable.java

Синхронизация

Гонки потоков

- Два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков
- Сложность проблемы –нерегулярность возникающих ситуаций (все определяется взаимными скоростями потоков и моментами их прерываний)

Критическая секция

- Часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение данной части еще не завершено
- Определяется по отношению к определенным критическим данным, при несогласованном изменении которых могут возникнуть нежелательные эффекты
- Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один поток

На уровне блоков кода

- Синхронизация через блокировку
- Object - представим, что есть скрытое поле (признак блокировки)
- synchronized можно применить к методу или блоку
- synchronized определяет критическую секцию

```
class  MyClass {
synchronized  void method 1() {
. . .}
synchronized  void method 2() {
. . .}
void method3() {. . .}

}
```

```
MyClass ob1;

работаем в потоке T1
ob1. method 1() ;
ob1.method2();
ob1.method3();

работаем в потоке T2
ob1.method1(); // T1 run
ob1.method2(); // T1 run
ob1.method3();
```

Пример

```
class ThreadsWrite implements Runnable{
//synchronized
public void run() {
        System.out.print("Hello,");
        for (int i=0; i<10000; i++)
        for (int j=0; j<2000; j++) ;
// Thread.sleep()  или
// Thread.yeld();
    System.out.println("World!!!");
}
```

```
public static void main (String[] args){
  ThreadsWrite tw = new ThreadsWrite();
  new Thread(tw).start();
  new Thread(tw).start();
  new Thread(tw).start();
 new Thread(tw).start();
}}
```

- Синхронизация блокирует объект
- Если метод не имеет synchronized, он не проверяет блокировку объекта
- Блокировка допускает многократное вхождение (реентерабельность)
- Синхронизация – при входе в блок состояние блокируемого объекта читается из основной памяти в память потока, после завершения блока, измененное состояние блокируемого объекта сбрасывается из памяти потока в основную память
- Блокируется весь объект, а не только его поля

Пример с банком

Bank.java

TransferRunnable.java

UnsynchBankTest.java

Пример с банком

Результаты выполнения

```
Thread[Thread-8,5,main]     286,45 from 8 to 5Total Balance:   10000,00
Thread[Thread-7,5,main]     126,85 from 7 to 1Total Balance:   10000,00
Thread[Thread-9,5,main]     115,62 from 9 to 7Total Balance:   10000,00
Thread[Thread-0,5,main]     283,49 from 0 to 2Total Balance:   10000,00
Thread[Thread-7,5,main]     765,27 from 7 to 6Total Balance:   10000,00
Thread[Thread-2,5,main]     946,71 from 2 to 3Total Balance:    9959,26
Thread[Thread-5,5,main]     469,55 from 5 to 8Total Balance:    9959,26
Thread[Thread-3,5,main]     421,02 from 3 to 5Total Balance:    9959,26
      40,74 from 1 to 6Total Balance:   10000,00
```

Пример с банком

Результаты выполнения

```
begin=1588144990085
Thread[Thread-4,5,main]Thread[Thread-5,5,main]     456,36 from 5 to 9Thread[Thread-1,5,main]      90,77
Thread[Thread-9,5,main]Thread[Thread-8,5,main]     814,80 from 8 to 3Total Balance:    9462,89
```

Добавим сообщение о том, что невозможен перевод

```
Thread[Thread-2,5,main]Thread[Thread-4,5,main]Thread[Thread-5,5,main]
     112,24 from 6 to 3Total Balance:    7642,25
     685,46 from 4 to 7     745,63 from 2 to 1Total Balance:    9991,34
       8,66 from 9 to 4Total Balance:   10000,00
insufficient funds in the account
```

Пример с банком

```
public  synchronized  void transfer (int from,   int to, double amount)
     throws InterruptedException {
. . .
}

public  synchronized  double getTotalBalance() {
. . .
}
```

wait и notify

- Для согласования работы
- Метод wait() ожидает изменения некоторого условия, неподконтрольного для текущего метода. Метод – альтернатива циклу проверки наступления условия
- Метод wait() вызывается в блоке синхронизации, но он снимает блокировку с объекта синхронизации, позволяя другим потокам вызывать другие синхронизованные методы данного объекта

wait и notify

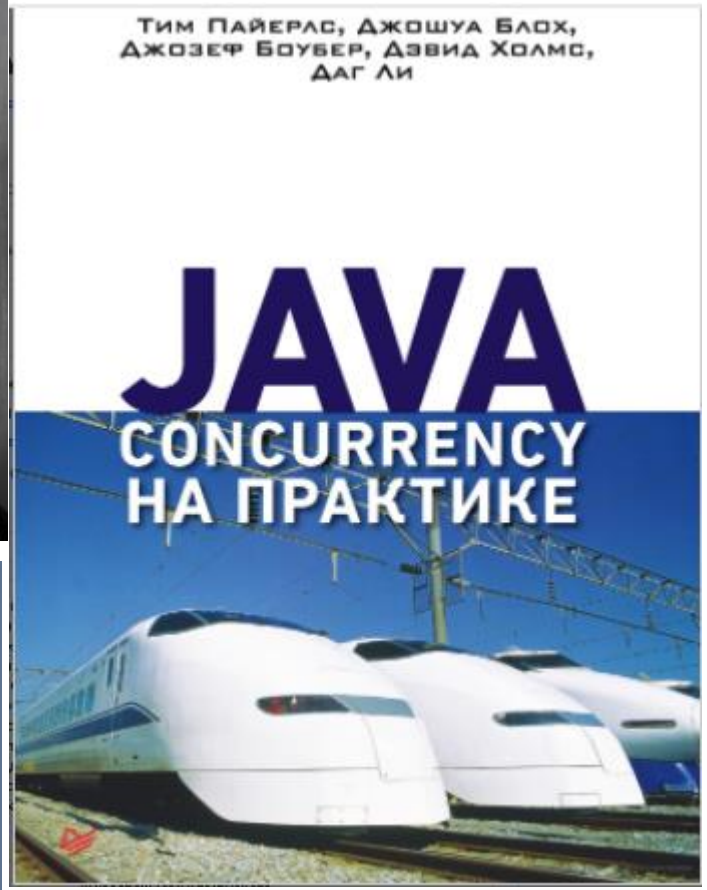
- Выйти из состояния ожидания, установленного методом wait(), можно
  - С помощью уведомления notify() или notifyAll()
  - По истечении срока ожидания, если он был задан
- Метод notify() вызывается в блоке синхронизации

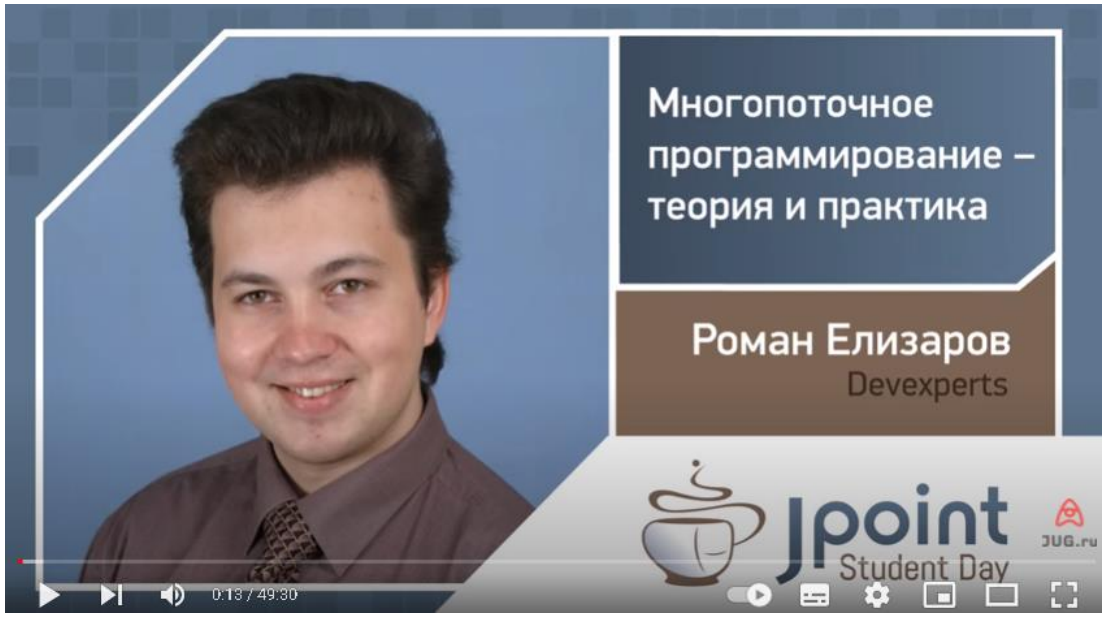
Пример

```
public class ThreadGate { // extends Object
    private boolean isOpen = false;
    public synchronized void open() {
        try {
            tryOpen();
            isOpen = true;

        } catch (InterruptedException ex) {
//???
    }
```

```
public  synchronized void tryOpen() throws
    InterruptedException {
            while (isOpen) wait();        // BLOCKS
}
public synchronized void close() {
            isOpen = false;
            notifyAll();
}

}
```


Joker<?> 2017
Cay Horstmann
San Jose State University
Concurrency
for Humans
JUG.ru


Тим Пайерлс, Джошуа Блох,
Джозеф Боубер, Дэвид Холмс,
Даг Ли
JAVA
CONCURRENCY
НА ПРАКТИКЕ


Многопоточное
программирование –
теория и практика
Роман Елизаров
Devexperts
Jpoint
Student Day
JUG.ru
0:13 / 49:30

`java.util.concurrent`

Lea D. Concurrent Programming in Java

Начиная с Java 1.5

Включает

- классы для явного управления блокировкой
- коллекции для безопасной работы с потоками
- классы для управления синхронизацией
- классы, реализующие интерфейс Callable
- классы для реализации механизма ForkJoin
- классы для атомарных операций с переменными

`java.util.concurrent.atomic`

- Реализует семантику volatile
- Чаще всего используются для реализации последовательностей значений

java.util.concurrent.locks

- Интерфейс Lock
- Классы
  - ReentrantLock
  - ReentrantReadWriteLock
- Методы
  - lock()
  - unlock()
  - tryLock()

Схема, если нет тупиков

```
private Lock myLock = new ReentrantLock();

myLock.lock(); // объект заблокирован, больше никто
                // не может его блокировать
    try {
                // критический фрагмент кода
    }
    finally
    {
      myLock.unlock(); // операцию разблокирования
                       //необходимо выполнить даже
                       //при возникновении исключения
     }
```

Чтобы избежать тупиков

```
if (myLock.tryLock(1,TimeUnit.SECONDS)) {
    try {
        // критический фрагмент кода
    }
    finally
    {
        myLock.unlock();
    }
}
else { // блокировка не удалась
        // выполнение других действий
        // Thread.sleep(waitTime);
}
```

Блокировки чтения-записи

Создать объект ReentrantReadWriteLock

```
private ReentrantReadWriteLock rw =
        new ReentrantReadWriteLock ();
```

Сформировать на его основе отдельные объекты для блокировки чтения и записи

```
private Lock readLock = rw.readLock();
private Lock writeLock = rw.writeLock();
```

Блокировки чтения-записи

Использовать блокировку чтения там, где возможен совместный доступ

```
readLock.lock();
try { . . . }
finally (
readLock.unlock();
}
```

Использовать блокировку записи там, где необходим эксклюзивный доступ

```
writeLock.lock();
try { . . . }
finally {
 writeLock.unlock();
}
```

Объекты условий

Интерфейс Condition

Создается из объекта блокировки

```
Condition myCond = myLock.newCondition();
```

Методы объекта условия, связанного с объектом блокировки

- await()
- signal()
- signalAll()

Пример использования

```
private Lock myLock = new ReentrantLock();
private  Condition myCond = myLock.newCondition();
myLock.lock();
try {
   while(. . .)       //условие когда действие невозможно
         myCond.await();
 // действие
     myCond.signalAll();
}
finally {
   myLock.unlock();
}
```

SynchBankTest.java