Средства многопоточного программирования

Интерфейс Runnable

```
public interface Runnable {
        void run();
}

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

Создать экземпляр класса

```
Runnable r = new MyRunnable();
```

Создать поток

```
Thread t = new Thread ( r );
```

Запустить поток на выполнение

```
t.start();
```

Класс Thread

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

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

Метод run()

- Содержит весь код, который должен быть выполнен в потоке
- Если вызвать метод run() непосредственно, то выполнение будет происходить в том же потоке. Новый поток сформирован не будет
- Для запуска потока предназначен метод start() класса Thread
- Момент начала работы потока не определяется последовательностью команд, среди которых находятся команды start()
- Работа метода run() может быть приостановлена, а затем продолжена

Основные методы класса Thread

- static void sleep(long m)
- void start()
- void run()
- void interrupt()
- boolean isInterrupdet()
- static Thread currentThread()
- boolean isAlive()
- void join() / void join(long m)
- void setPriority (int p)
- static void yield()
- viod setDaemon(boolean d)

Устаревшие методы класса Thread

- void destroy()
- void resume()
- void stop()
- void suspend()

- В документации

For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.

Корректная схема метода run()

```
public void run() {
  try {     . . .
 while (!Thread.currentThread().isInterrupted()  && . . .){
  // код
}
catch (InterruptedException  e) { . . .
     Thread.currentThread().interrupt();
}
finally {
 освобождение ресурсов
}
}
```

- Если выполнение потока заблокировано (sleep() или wait()),поток не может проверить нужно ли завершить работу
- В этом случае блокирующие методы завершаются выбрасывая исключение InterruptedException

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

Пример

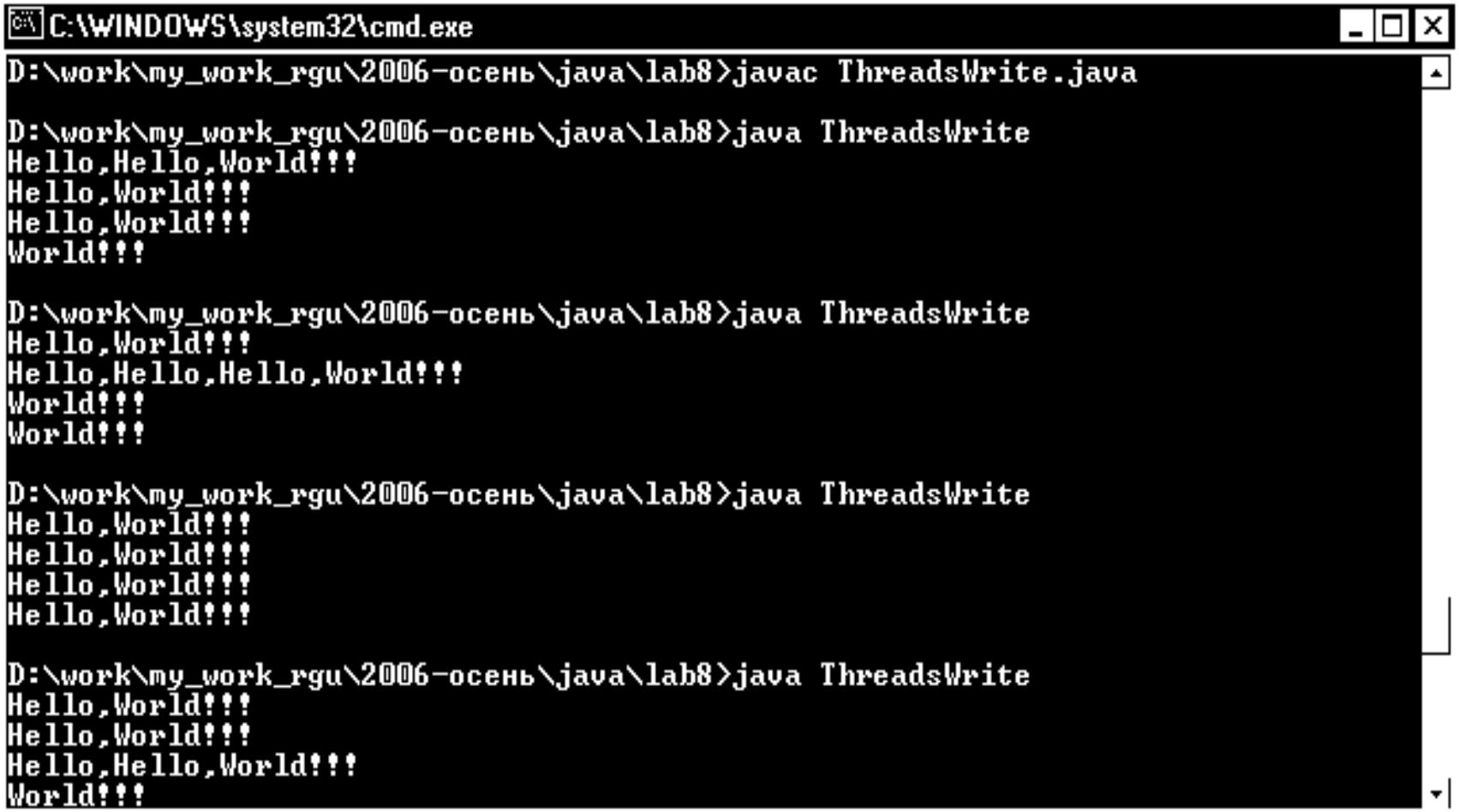
```
class ThreadsWrite implements Runnable{
        public void run() {
//  чтобы добиться синхронизации изменить заголовок метода
// synchronized public void run() {
                System.out.print("Hello,");

                for (int i=0; i<10000; i++)
                        for (int j=0; j<2000; j++) ;
// вместо цикла можно использовать Thread.sleep()  или Thread,yeld();
                System.out.println("World!!!");
}
```

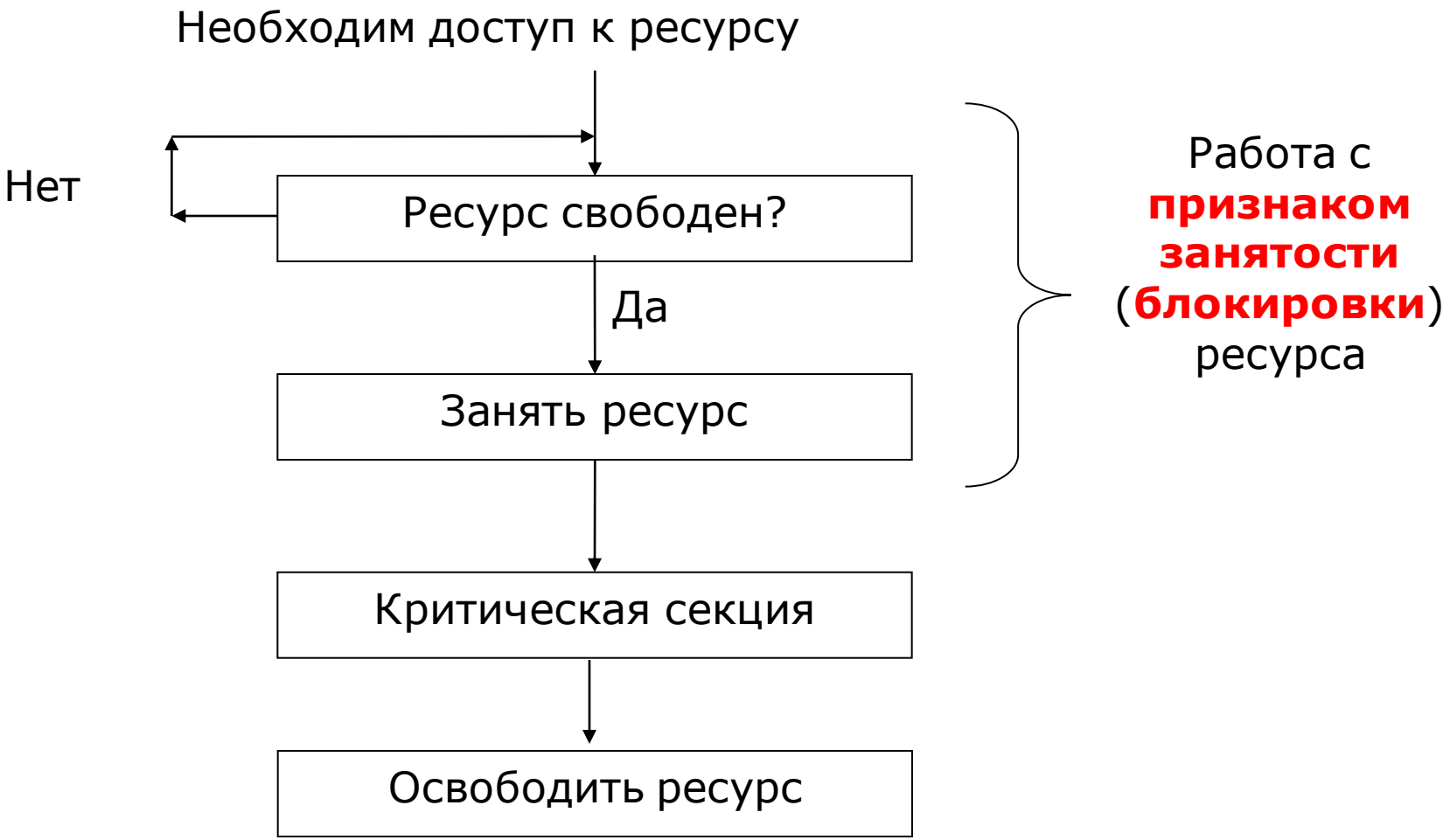
Пример

```
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();
// можно запустить и больше потоков
}}
```

Пример


C:\WINDOWS\system32\cmd.exe
D:\work\my_work_rgu\2006-осень\java\lab8>javac ThreadsWrite.java
D:\work\my_work_rgu\2006-осень\java\lab8>java ThreadsWrite
Hello,Hello,World!!!
Hello,World!!!
Hello,World!!!
World!!!
D:\work\my_work_rgu\2006-осень\java\lab8>java ThreadsWrite
Hello,World!!!
Hello,Hello,Hello,World!!!
World!!!
World!!!
D:\work\my_work_rgu\2006-осень\java\lab8>java ThreadsWrite
Hello,World!!!
Hello,World!!!
Hello,World!!!
Hello,World!!!
D:\work\my_work_rgu\2006-осень\java\lab8>java ThreadsWrite
Hello,World!!!
Hello,World!!!
Hello,Hello,World!!!
World!!!

Синхронизация потоков



Синхронизованные методы

Если в заголовке метода указать ключевое слово synchronized, то метод становится синхронизованным – блокируется объект, выполняющий метод.

Если один поток выполняет синхронизованный метод объекта, то другие потоки не могут обратиться к этому методу, а также ко всем остальным синхронизованным методам этого объекта.

```
class Callme {
synchronized void call(String msg) {
System.out.print("[" + msg);
//Thread.yeld();
for ( int i=0; i<10000; i++)
for ( int j=0; j<10000; j++);
System.out.println("]");
} }
```

Синхронизованные блоки

Еще один способ потоку получить блокировку – выполнить синхронизованный блок.

В синхронизованном блоке поток может заблокировать объект, с которым работает, чтобы другие потоки не могли обратиться к данному объекту, пока блокировка не будет снята.

Синхронизованный блок

```
synchronized (obj) {
   obj.method();
}
```

Согласование работы потоков

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

Согласование работы потоков

- 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

```
public class ThreadGate {
    private boolean isOpen = false;
    public synchronized void close() {
        isOpen = false;
    }
    public synchronized void open() {
        isOpen = true;
        notifyAll();
    }
// BLOCKS-UNTIL
    public synchronized void isOpen()
                                throws InterruptedException {
        while (!isOpen)
            wait();
    }
}
```

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

Объекты блокировки

Пакет `java.util.concurrent.lock`

- интерфейс Lock
- классы ReentrantLock и ReentrantReadWriteLock
- методы
  - lock()
  - unlock()
  - tryLock()

Типичная схема работы

```
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
     }
```

Типичная схема работы

Если поток не может заблокировать объект, он переходит в состояние ожидания и остается в нем до разблокирования объекта.

Можно применить более гибкий механизм, используя метод tryLock()

Типичная схема работы

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

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();
}
```

Наборы для безопасной работы с потоками

Блокирующие очереди

Эффективные очереди

Эффективные хеш-таблицы

Массивы, копируемые при записи

Блокирующие очереди

Приостанавливает работу потока, если он пытается добавить элемент в заполненную очередь или извлечь элемент из пустой очереди

Предоставляют операции трех типов

- Генерирующие исключение при ошибочном использовании
- Возвращающие значение, свидетельствующее об ошибке
- Блокирующие

Операции

• Генерирующие исключение	add() - при переполнении очереди IllegalStateException remove() - если очередь пуста NoSuchElementException element() если очередь пуста NoSuchElementException
• Возвращающие признак ошибки	offer() -false, если очередь заполнилась poll()- null, если очередь пуста peek() - null, если очередь пуста
• Блокирующие	put() take()

Классы блокирующих очередей

- LinkedBlockingQueue <E>
- ArrayBlockingQueue <E>
- DelayQueue <E extends Delayed>
- PriorityBlockingQueue <E >

Эффективные коллекции

- Эффективные наборы поддерживают большое количество читающих потоков и фиксированное число записывающих.
- Наборы данных используют специальные алгоритмы, позволяющие избежать блокирования всей структуры и минимизировать конфликты, допуская одновременный доступ к различным частям структуры
- Эффективные наборы данных представляют «слабо согласованные итераторы»
- Это означает, что итератор не обязательно отражает все изменения, произошедшие после их создания, но никогда не возвращают значение дважды и не вызывают исключение

Классы эффективных коллеций

- ConcurrentLinkedQueue <E>
- ConcurrentHashMap <E>
- CopyOnWriteArrayList <E>
- CopyOnWriteArraySet <E>

Обертки для коллекций Java 2

```
List synchAL =
    Collections.synchronizedList (
                        new ArrayList<…>());
Map synchM =
    Collections.synchronizedMap(
                        new HashMap<…>());
```

Синхронизующие оболочки управляют лишь методами коллекции

Их действие не распространяется на итераторы коллекций

При использовании итераторов и цикла for each необходима явная блокировка

Синхронизаторы

- Классы, которые помогают управлять набором взаимодействующих потоков
- Используются в ряде типов взаимодействия между потоками («рандеву»)

Классы синхронизаторов

- Semaphore
- CyclicBarrier
- CountDownLatch
- Phaser (начиная с Java 7)
- Exchanger
- SynchronousQueue

Семафоры

```
private final Semaphore available =
                    new Semaphore(MAX_AVAILABLE);
public Object getItem() throws InterruptedException {
    available.acquire();
    return getNextAvailableItem();
}
 public void putItem(Object x) {
  if (markAsUnused(x))
   available.release();
}
```

Барьеры

```
Runnable barrierAction = . . . ;
CyclicBarrier barrier =  new CyclicBarrier(nthreads,
                                           barrierAction);

public void run() {
  doWork();
  barrier.await();
 . . . .
}
```

Барьеры

```
class Worker implements Runnable {
 private final CountDownLatch startSignal;
 private final CountDownLatch doneSignal;
 public void run() {
 try {
     startSignal.await();
     doWork();
     doneSignal.countDown();
  } catch (InterruptedException ex) {  return; }
 }
void doWork() { ... }
}
```

Класс SynchronousQueue

- Реализует объединение потока-поставщика и потока-потребителя
- Данные передаются только в одном направлении
- Очередь не используется
- Если поток вызвал метод **`put()`**, его выполнение приостанавливается, пока другой поток не вызовет метод **`take()`**

Класс Exchanger<V>

- Используется при работе двух потоков с двумя экземплярами одной структуры данных.
- Один поток заполняет свой экземпляр, другой считывает из своего экземпляра
- Когда каждый поток выполнит свою задачу, они обмениваются экземплярами