# Синхронизация потоков. Оператор synchronized

При работе потоки нередко обращаются к каким-то общим ресурсам, которые определены вне потока, например, обращение к какому-то файлу. Если одновременно несколько потоков обратятся к общему ресурсу, то результаты выполнения программы могут быть неожиданными и даже непредсказуемыми.

### Пример 1. Одновременный доступ к ресурсу

```
package syn_001;

// При работе потоки нередко обращаются к каким-то общим ресурсам,
// которые определены вне потока, например, обращение к какому-то
файлу.
// Если одновременно несколько потоков обратятся к общему ресурсу,
// то результаты выполнения программы могут быть неожиданными
// и даже непредсказуемыми.

class Program {
//    В главном классе программы запускается три потока.
//    То есть мы ожидаем, что каждый поток будет увеличивать res.x с
1 до 4
//    и так три раза.
//    Но если мы посмотрим на результат работы программы, то он
будет иным.
    public static void main(String[] args) {

        CommonResource commonResource= new CommonResource();
        for (int i = 1; i < 4; i++){
            Thread t = new Thread(new CountThread(commonResource));
            t.setName("Thread "+ i);
            t.start();
        }
    }
}

// Здесь определен класс CommonResource,
// который представляет общий ресурс и
// в котором определено одно целочисленное поле x.
class CommonResource{

    int x=0;
}

//    Этот ресурс используется классом потока CountThread.
```

```
//    Этот класс просто увеличивает в цикле значение x на единицу.
//    Причем при входе в поток значение x=1:
class CountThread implements Runnable{
    CommonResource res;

    CountThread(CommonResource res){
        this.res=res;
    }
// То есть в итоге мы ожидаем, что после выполнения цикла res.x
будет равно 4.
    public void run(){
        res.x=1;
        for (int i = 1; i < 5; i++){
            System.out.printf("%s %d\n",
Thread.currentThread().getName(), res.x);
            res.x++;
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }}
```

То есть пока один поток не окончил работу с полем res.x, с ним начинает работать другой поток.

Чтобы избежать подобной ситуации, надо синхронизировать потоки. Одним из способов синхронизации является использование ключевого слова synchronized. Этот оператор предваряет блок кода или метод, который подлежит синхронизации. Для его применения изменим класс CountThread:

**Пример 2. Разделенный доступ к ресурсу**

```
package syn_002;

class Program {

    public static void main(String[] args) {

        CommonResource commonResource = new CommonResource();
        for (int i = 1; i < 4; i++) {
            Thread t = new Thread(new CountThread(commonResource));
            t.setName("Thread " + i);
            t.start();
        }
```

```
    }
}

class CommonResource {

    int x = 0;
}

class CountThread implements Runnable {
    CommonResource res;

    CountThread(CommonResource res) {
        this.res = res;
    }

    public void run() {
// При создании синхронизированного блока кода после оператора
synchronized
// идет объект-заглушка: synchronized(res).
// Причем в качестве объекта может использоваться только объект
какого-нибудь класса,
// но не примитивного типа.
        synchronized (res) {
            res.x = 1;
            for (int i = 1; i < 5; i++) {
                System.out.printf("%s %d\n",
Thread.currentThread().getName(), res.x);
                res.x++;
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                }}}}}
```

**Каждый объект в Java имеет ассоциированный с ним монитор**. Монитор представляет своего рода инструмент для управления доступа к объекту. Когда выполнение кода доходит до оператора synchronized, монитор объекта res блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку. После окончания работы блока кода, монитор объекта res освобождается и становится доступным для других потоков.

После освобождения монитора его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.

*Монитор* - это объект, используемый в качестве взаимоисключающей блокировки. Когда поток исполнения запрашивает блокировку, то говорят, что он входит в монитор.

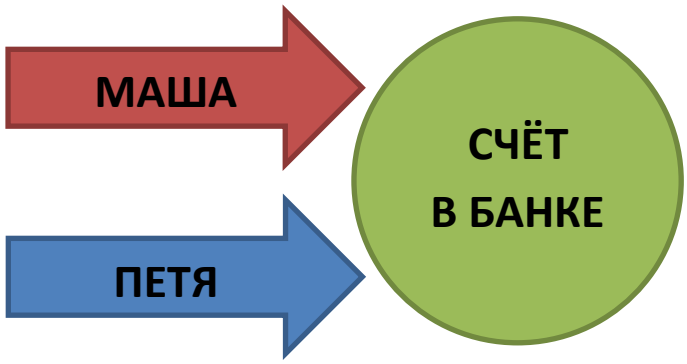
Только один поток исполнения может в одно, и то же время владеть монитором. Все другие потоки исполнения, пытающиеся войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не выйдет из монитора. Говорят, что они ожидают монитор.

Поток, владеющий монитором, может, если пожелает, повторно войти в него.

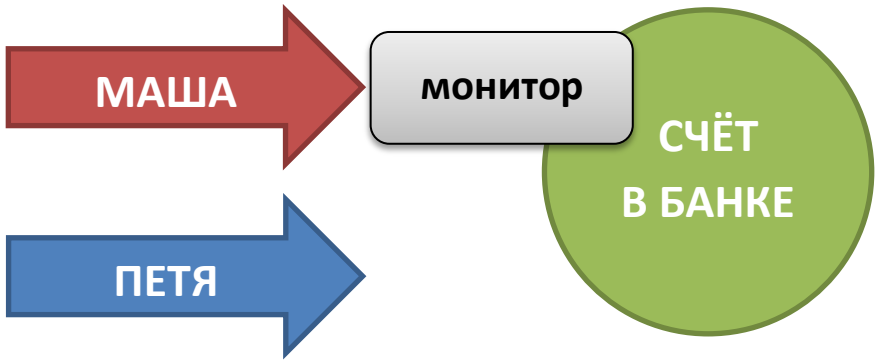
Если поток засыпает, то он удерживает монитор.

Поток может захватить сразу несколько мониторов.

Рассмотрим разницу между доступом к объекту без синхронизации и из синхронизированного кода. Доступ к банковскому счету без синхронизации:




И с синхронизацией:



## Методы wait и notify

Иногда при взаимодействии потоков встает вопрос **о извещении одних потоков о действиях других**. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса Object определено ряд методов:

- **wait**(): освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()
- **notify**(): продолжает работу потока, у которого ранее был вызван метод wait()
- **notifyAll**(): возобновляет работу всех потоков, у которых ранее был вызван метод wait()

**Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода**.

Возьмем стандартную задачу - "Производитель-Потребитель" ("Producer-Consumer"): пока производитель не произвел продукт, потребитель не может его купить. Пусть производитель должен произвести 5 товаров, соответственно потребитель должен их все купить. Но при этом одновременно на складе может находиться не более 3 товаров. Для решения этой задачи задействуем методы wait() и notify():

> **Пример 3. Методы wait и notify**

```
package syn_003;
class Program {

    public static void main(String[] args) {
        Store store=new Store();
```

```
        Producer producer = new Producer(store);
        Consumer consumer = new Consumer(store);
        new Thread(producer).start();
        new Thread(consumer).start();
    }
}


// Класс Магазин, хранящий произведенные товары
class Store{
// Для отслеживания наличия товаров в классе Store проверяем
значение переменной product.
// По умолчанию товара нет, поэтому переменная равна 0.
    private int product=0;


    public synchronized void get() {
// Метод get() - получение товара должен срабатывать только при
наличии
// хотя бы одного товара. Поэтому в методе get проверяем,
отсутствует ли товар:
        while (product<1) {
            try {
                // Если товар отсутсвует, вызывается метод wait().
                wait();
                // Этот метод освобождает монитор объекта Store
                // и блокирует выполнение метода get,
                // пока для этого же монитора не будет вызван метод
notify().
            }
            catch (InterruptedException e) {
            }
        }

        // Затем имитируется получение покупателем товара.
        // Для этого выводится сообщение, и уменьшается значение
product:
        // product--.
        product--;
        System.out.println("Покупатель купил 1 товар");
        System.out.println("Товаров на складе: " + product);
        notify();
    }


    public synchronized void put() {
        while (product>=3) {
            try {
                // А в методе put() с помощью wait() мы ожидаем
освобождения места на складе.
```

```
                // После того, как место освободится, добавляем
товар и через notify()
                // уведомляем покупателя о том, что он может
забирать товар.
                wait();
            }
            catch (InterruptedException e) {
            }
        }
        // Когда в методе put() добавляется товар и вызывается
notify(),
        // то метод get() получает монитор и выходит из конструкции
while (product<1),
        // так как товар добавлен.
        product++;
        System.out.println("Производитель добавил 1 товар");
        System.out.println("Товаров на складе: " + product);
        // И в конце вызов метода notify() дает сигнал методу put()
продолжить работу.
        notify();
    }
}

// класс Производитель
class Producer implements Runnable{

    Store store;
    Producer(Store store){
        this.store=store;
    }
    public void run(){
        for (int i = 1; i < 6; i++) {
            store.put();
        }
    }
}


// Класс Потребитель
class Consumer implements Runnable{

    Store store;
    Consumer(Store store){
        this.store=store;
    }

    public void run(){
        for (int i = 1; i < 6; i++) {
            store.get();
        }}}
```

## Семафоры

**Семафоры представляют еще одно средство синхронизации для доступа к ресурсу**. В Java семафоры представлены классом **Semaphore**, который располагается в пакете java.util.concurrent.

Для управления доступом к ресурсу **семафор** использует **счетчик**, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора.

Установить количество разрешений для доступа к ресурсу можно с помощью конструкторов класса Semaphore:

```
Semaphore(int permits)
```

```
Semaphore(int permits, boolean fair)
```

Параметр permits указывает на количество допустимых разрешений для доступа к ресурсу. Параметр fair во втором конструкторе позволяет установить очередность получения доступа. Если он равен true, то разрешения будут предоставляться ожидающим потокам в том порядке, в каком они запрашивали доступ. Если же он равен false, то разрешения будут предоставляться в неопределенном порядке.

Для получения разрешения у семафора надо вызвать метод acquire(), который имеет две формы:

```
void acquire() throws InterruptedException
```

```
void acquire(int permits) throws InterruptedBxception
```

Для получения одного разрешения применяется первый вариант, а для получения нескольких разрешений - второй вариант.

После вызова этого метода пока поток не получит разрешение, он блокируется.

**После окончания работы с ресурсом полученное ранее разрешение надо освободить с помощью метода release**():

```
void release()

void release(int permits)
```

Первый вариант метода освобождает одно разрешение, а второй вариант - количество разрешений, указанных в permits.

> **Пример 4. Семафор**
>
> ```
> package syn_004;
>
> import java.util.concurrent.Semaphore;
>
> class Program {
>
>     public static void main(String[] args) {
>
>         Semaphore sem = new Semaphore(1); // 1 разрешение
>         CommonResource res = new CommonResource();
>         new Thread(new CountThread(res, sem, "CountThread
> 1")).start();
>         new Thread(new CountThread(res, sem, "CountThread
> 2")).start();
>         new Thread(new CountThread(res, sem, "CountThread
> 3")).start();
>     }
> }
> class CommonResource{
>
>     int x=0;
> }
> ```

```
class CountThread implements Runnable{

    CommonResource res;
    Semaphore sem;
    String name;
    CountThread(CommonResource res, Semaphore sem, String
name){
        this.res=res;
        this.sem=sem;
        this.name=name;
    }

    public void run(){

        try{
            System.out.println(name + " ожидает
разрешение");
            sem.acquire();
            res.x=1;
            for (int i = 1; i < 4; i++){
                System.out.println(this.name + ": " +
res.x);
                res.x++;
                Thread.sleep(100);
            }
        }
        catch(InterruptedException
e){System.out.println(e.getMessage());}
        System.out.println(name + " освобождает
разрешение");
        sem.release();}}
```

Итак, здесь есть общий ресурс CommonResource с полем x, которое изменяется каждым потоком. Потоки представлены классом CountThread, который получает семафор и выполняет некоторые действия над ресурсом. В основном классе программы эти потоки запускаются. В итоге мы получим следующий вывод:

```
CountThread 1 ожидает разрешение
CountThread 2 ожидает разрешение
CountThread 3 ожидает разрешение
CountThread 1: 1
```

```
CountThread 1: 2
CountThread 1: 3
CountThread 1 освобождает разрешение
CountThread 2: 1
CountThread 2: 2
CountThread 2: 3
CountThread 2 освобождает разрешение
CountThread 3: 1
CountThread 3: 2
CountThread 3: 3
CountThread 3 освобождает разрешение
```

Семафоры отлично подходят для решения задач, где надо ограничивать доступ. Например, **классическая задача про обедающих философов**. Ее суть: есть несколько философов, допустим, пять, но одновременно за столом могут сидеть не более двух. И надо, чтобы все философы пообедали, но при этом не возникло взаимоблокировки философами друг друга в борьбе за тарелку и вилку:

**Пример 5. Философы и приемы пищи**

```
package syn_005;
import java.util.concurrent.Semaphore;

class Program {

    public static void main(String[] args) {

        Semaphore sem = new Semaphore(2);
        for(int i=1;i<6;i++)
            new Philosopher(sem,i).start();
    }
}
// класс философа
class Philosopher extends Thread
{
    Semaphore sem; // семафор. ограничивающий число философов
    // кол-во приемов пищи
    int num = 0;
    // условный номер философа
    int id;
    // в качестве параметров конструктора передаем идентификатор
```

```
философа и семафор
    Philosopher(Semaphore sem, int id)
    {
        this.sem=sem;
        this.id=id;
    }

    public void run()
    {
        try
        {
            while(num<3)// пока количество приемов пищи не достигнет 3
            {
                //Запрашиваем у семафора разрешение на выполнение
                sem.acquire();
                System.out.println ("Философ " + id+" садится за
стол");
                // философ ест
                sleep(500);
                num++;

                System.out.println ("Философ " + id+" выходит из-за
стола");
                sem.release();

                // философ гуляет
                sleep(500);
            }
        }
        catch(InterruptedException e)
        {
            System.out.println ("у философа " + id + " проблемы со
здоровьем");}}}
```

В итоге только два философа смогут одновременно находиться за столом, а другие будут ждать.

## Обмен между потоками. Класс Exchanger

Класс **Exchanger** предназначен для обмена данными между потоками. Он является типизированным и типизируется типом данных, которыми потоки должны обмениваться.

Обмен данными производится с помощью единственного метода этого класса **exchange**():

```
V exchange(V x) throws InterruptedException

V exchange(V x, long timeout, TimeUnit unit) throws
InterruptedException, TimeoutException
```

Параметр x представляет буфер данных для обмена. Вторая форма метода также определяет параметр timeout - время ожидания и unit - тип временных единиц, применяемых для параметра timeout.

Данный класс очень просто использовать:

**Пример 6. Класс Exchanger**

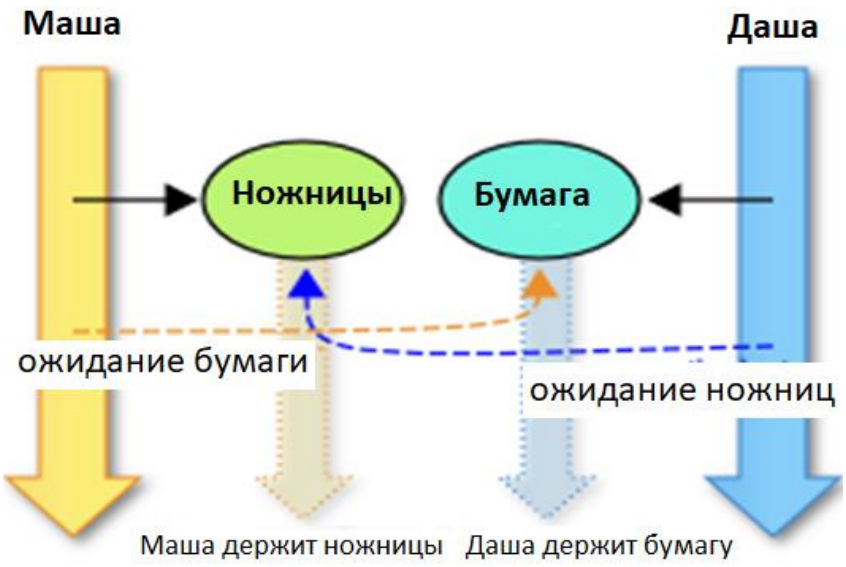
```
package syn_006;

import java.util.concurrent.Exchanger;

class Program {

    public static void main(String[] args) {

        Exchanger<String> ex = new Exchanger<String>();
        new Thread(new PutThread(ex)).start();
        new Thread(new GetThread(ex)).start();
    }
}

class PutThread implements Runnable{

    Exchanger<String> exchanger;
    String message;

    PutThread(Exchanger<String> ex){

        this.exchanger=ex;
        message = "Hello Java!";
    }
    public void run(){
```

```
        try{
            message=exchanger.exchange(message);
            System.out.println("PutThread has received: " + message);
        }
        catch(InterruptedException ex){
            System.out.println(ex.getMessage());
        }
    }
}
class GetThread implements Runnable{

    Exchanger<String> exchanger;
    String message;

    GetThread(Exchanger<String> ex){

        this.exchanger=ex;
        message = "Hello World!";
    }
    public void run(){

        try{
            message=exchanger.exchange(message);
            System.out.println("GetThread has received: " + message);
        }
        catch(InterruptedException ex){
            System.out.println(ex.getMessage());
        }}}
```

## Взаимная блокировка

Особый тип ошибок, которого следует избегать, имеющий отношение к многозадачности это *взаимная блокировка* (deadlock). Она происходит, когда потоки имеют циклическую зависимость от пары синхронизированных объектов.

Например, две девочки Маша и Даша в детском саду делают аппликацию. Для работы каждой нужны ножницы и цветная бумага. Предположим Маша взяла ножницы (поток Маша вошла в монитор объекта ножницы), а Даша бумагу (поток Даша вошла в монитор объекта бумага). Каждая из них ждет другой предмет и не хочет делиться тем, что взяла. Они не могут продолжить свою работу и будут ждать вечно (пока воспитательница не поможет им).




### Пример. Взаимная блокировка

```
package syn_006;

class DeadlockRisk implements Runnable {
    private static class Resource {
    }

    private final Resource scissors = new Resource();
    private final Resource paper = new Resource();

    public void doSun() {
        synchronized (scissors) { // May deadlock here
            System.out.println(Thread.currentThread().getName()
                    + " взяла ножницы для вырезания солнышка");
            synchronized (paper) {
                System.out.println(Thread.currentThread().getName()
                        + " взяла бумагу для вырезания солнышка");
                System.out.println(Thread.currentThread().getName()
                        + " вырезает солнышко");
            }
        }
    }

    public void doCloud() {
        synchronized (paper) { // May deadlock here
            System.out.println(Thread.currentThread().getName()
                    + " взяла бумагу для вырезания облачка");
            synchronized (scissors) {
                System.out.println(Thread.currentThread().getName()
                        + " взяла ножницы для вырезания облачка");
                System.out.println(Thread.currentThread().getName()
                        + " вырезает облачко");
```

```
            }
        }
    }

    public void run() {
        doSun();
        doCloud();
    }

    public static void main(String[] args) {
        DeadlockRisk job = new DeadlockRisk();
        Thread masha = new Thread(job, "Маша");
        Thread dasha = new Thread(job, "Даша");
        masha.start();
        dasha.start();
    }}
```

Если поток пытается зайти в синхронизированный метод, а монитор уже захвачен, то поток блокируется по монитору объекта.

Поток попадает в специальный пул для этого конкретного объекта и должен находиться там пока монитор не будет освобожден. После этого поток возвращается в состояние runnable.

**Варианты блокировки:**

1. Потоки, вызывающие нестатические синхронизированные методы одного и того же класса, будут блокировать друг друга только если они вызваны для одного объекта.
2. Потоки, вызывающие статические синхронизированные методы одного класса, будут всегда блокировать друг друга. Они блокируются по монитору Class объекта. Статические синхронизированные и нестатические синхронизированные методы не будут блокировать друг друга никогда.
3. Для синхронизированных блоков нужно смотреть - какой объект используется для синхронизации.
4. Блоки, синхронизированные по одному объекту, будут блокировать друг друга.

**Методы и состояние блокировки**

| Освобождают монитор | Удерживают монитор | Класс, определяющий метод |
|---|---|---|
| wait() | notify() | java.lang.Object |
| | join() | java.lang.Thread |
| | sleep() | java.lang.Thread |
| | yield() | java.lang.Thread |

## Межпотоковые коммуникации

В Java внедрен изящный механизм взаимодействия потоков исполнения с помощью методов *wait(), notify()* и *notifyAll()*. Эти методы реализованы как *final* в классе *Object*, поэтому они доступны всем классам.

Все три метода могут быть вызваны только из *synchronized* контекста.

Метод *wait()* принуждает вызывающий поток отдать монитор и приостановить выполнение до тех пор, пока какой-нибудь другой поток не войдет в тот же монитор и не вызовет *notify()*.

Метод *notify()* возобновляет работу потока, который вызвал *wait()* на том же самом объекте.

Метод *notifyAll()* возобновляет работу всех потоков, который вызвали *wait()* на том же самом объекте. Одному из потоков дается доступ.

> **Пример. Взаимодействия потоков**

```
package syn_007;

class Producer implements Runnable {
    private MyQueue myQueue;

    public Producer(MyQueue myQueue) {
        this.myQueue = myQueue;
    }
```

```
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            myQueue.put(i);
        }
    }
}
class Consumer implements Runnable {
    private MyQueue myQueue;

    public Consumer(MyQueue myQueue) {
        this.myQueue = myQueue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            myQueue.get();
        }
    }
}
class MyQueue {
    private int n;
    boolean valueSet = false;

    public synchronized int get() {
        while (!valueSet) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Получено: " + n);
        valueSet = false;
        notify();
        return n;
    }

    public synchronized void put(int n) {
        while (valueSet) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        valueSet = true;
        this.n = n;
        System.out.println("Отправлено: " + n);
        notify();
```

```
    }
}
class ProducerDemo {
    public static void main(String[] args) {
        MyQueue myQueue = new MyQueue();

        Consumer consumer = new Consumer(myQueue);
        Producer producer = new Producer(myQueue);

        Thread t1 = new Thread(consumer);
        Thread t2 = new Thread(producer);

        t1.start();
        t2.start();
    }}
```

**Модификатор** *volatile*

Поток создается с чистой рабочей памятью, и должен перед использованием загрузить все необходимые переменные из основного хранилища (можно сказать что он имеет некий кэш).
Любая переменная сначала создается в основном хранилище и лишь затем копируется в рабочую память потоков, которые будут ее применять.
Если переменная объявлена, как ***volatile***, то ее чтение и запись будет производиться из\в основное хранилище.
Чтение ***volatile*** переменных синхронизировано и запись в ***volatile*** переменные синхронизирована, а неатомарные операции – нет.