Архитектура клиент-серверных приложений

Практическая работа №1 Многопоточные приложения, как основа распределённых клиент-серверных вычислений

Цель: Ознакомится с моделью многопоточного программирования в Java.

Теоретические сведения:

В отличие от многих других языков программирования, java предлагает встроенную поддержку многопоточного программирования. Многопоточная программа содержит две или более частей, которые могут выполняться одновременно. Каждая часть такой программы называется потоком (thread), и каждый поток задает отдельный путь выполнения. То есть, многопоточность - это специализированная форма многозадачности.

Существуют два отдельных типа многозадачности: многозадачность, основанная на процессах, и многозадачность, основанная на потоках. Важно понимать разницу между ними. Большинству читателей многозадачность, основанная на процессах, является более знакомой формой.

Процесс по сути своей - это выполняющаяся программа. То есть многозадачность, основанная на процессах, представляет собой средство, которое позволяет вашему компьютеру одновременно выполнять две или более программ. Так, например, процессная многозадачность позволяет запускать компилятор java в то самое время, когда вы используете текстовый редактор. В многозадачности, основанной на процессах, программа представляет собой наименьший элемент кода, которым может управлять планировщик операционной системы.

В среде поточной многозадачности наименьшим элементом управляемого кода явля-ется поток это означает, что одна программа может выполнять две или более задач од-новременно. Например, текстовый редактор может форматировать текст в то же время, когда выполняется его печать - до тех пор, пока эти два действия выполняются двумя отдельными потоками. То есть многозадачность на основе процессов имеет дело с "карти-ной в целом", а потоковая многозадачность справляется с деталями.

Многозадачные потоки требуют меньше накладных расходов, чем многозадачные процессы. Процессы - это тяжеловесные задачи, каждая из которых требует своего соб-ственного адресного пространства. Межпроцессные коммуникации дорогостоящи и ог-раничены. Переключение контекста от одного процесса к другому также обходится до-рого. С другой стороны, потоки являются облегченными. Они разделяют одно и тоже адресное пространство и совместно используют один и тот же тяжеловесный процесс.

Коммуникации между потоками являются экономными, а переключения контекста меж-ду потоками характеризуется низкой стоимостью. Хотя java-программы используются в средах процессной многозадачности, многозадачность, основанная на процессах, средствами java не управляется. А вот многопоточная многозадачность средствами java управ-ляется.

Многопоточность позволяет вам писать очень эффективные программы, которые по максимуму используют центральный процессор, поскольку время ожидания может быть сведено к минимуму. Это особенно важно для интерактивных сетевых сред, в которых работает java, так как в них наличие ожидания и простоев - обычное явление. Например, скорость передачи данных по сети намного ниже, чем скорость, с которой компьютер может их обрабатывать. Даже ресурсы локальной файловой системы читаются и пишутся намного медленнее, чем темп их обработки в процессоре. И, конечно, ввод пользователя намного медленнее, чем компьютер. В однопоточных средах ваша программа вынуждена ожидать окончания таких задач, прежде чем переходить к следующей, - даже если центральный процессор большую часть времени простаивает. Многопоточность позволяет получить доступ к этому времени ожидания и использовать его рациональным образом.

Если вы программировали для таких операционных систем, как Windows, это значит, что вы уже знакомы с многопоточным программированием. Однако тот факт, что java управляет потоками, делает многопоточность особенно удобной, поскольку многие детали подконтрольны вам как программисту.

Класс Thread

В Java функциональность отдельного потока заключается в классе Thread. И чтобы создать новый поток, нам надо создать объект этого класса. Но все потоки не создаются сами по себе. Когда запускается программа, начинает работать главный поток этой программы. От этого главного потока порождаются все остальные дочерние потоки.

С помощью статического метода Thread.currentThread() мы можем получить текущий поток выполнения:

```
public static void main(String[] args) {

    Thread t = Thread.currentThread(); // получаем главный поток
    System.out.println(t.getName()); // main
}
```

По умолчанию именем главного потока будет main.

Для управления потоком класс Thread предоставляет еще ряд методов. Наиболее используемые из них:

- getName(): возвращает имя потока
- setName(String name): устанавливает имя потока
- getPriority(): возвращает приоритет потока
- **setPriority(int proirity):** устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из кучи потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется средний приоритет - 5.
- isAlive(): возвращает true, если поток активен
- isInterrupted(): возвращает true, если поток был прерван
- join(): ожидает завершение потока
- run(): определяет точку входа в поток
- sleep(): приостанавливает поток на заданное количество миллисекунд
- start(): запускает поток, вызывая его метод run()

Мы можем вывести всю информацию о потоке:

```
public static void main(String[] args) {

    Thread t = Thread.currentThread(); // получаем главный поток
    System.out.println(t); // main
}
```

Первое main будет представлять имя потока (что можно получить через t.getName()), второе значение 5 предоставляет приоритет потока (также можно получить через t.getPriority()), и последнее main представляет имя группы потоков, к которому относится текущий - по умолчанию также main (также можно получить через t.getThreadGroup().getName())

Недостатки при использовании потоков

Далее мы рассмотрим, как создавать и использовать потоки. Это довольно легко. Однако при создании многопоточного приложения нам следует учитывать ряд обстоятельств, которые негативно могут сказаться на работе приложения.

На некоторых платформах запуск новых потоков может замедлить работу приложения. Что может иметь большое значение, если нам критичная производительность приложения.

Для каждого потока создается свой собственный стек в памяти, куда помещаются все локальные переменные и ряд других данных, связанных с выполнением потока. Соответственно, чем больше потоков создается, тем больше памяти используется. При этом надо помнить, в любой системе размеры используемой памяти ограничены. Кроме того, во многих системах может быть ограничение на количество потоков. Но даже если такого ограничения нет, то в любом случае имеется естественное ограничение в виде максимальной скорости процессора.

Создание и выполнение потоков

Для создания нового потока мы можем создать новый класс, либо наследуя его от класса Thread, либо реализуя в классе интерфейс Runnable.

Наследование от класса Thread

Создадим свой класс на основе Thread:

```
class JThread extends Thread {

    JThread(String name){
        super(name);
    }

    public void run(){

        System.out.printf("%s started... \n",
Thread.currentThread().getName());
        try{
            Thread.sleep(500);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.printf("%s fiished... \n",
Thread.currentThread().getName());
    }
}

public class Program {
```

Класс потока называется JThread. Предполагается, что в конструктор класса передается имя потока, которое затем передается в конструктор базового класса. В конструктор своего класса потока мы можем передать

различные данные, но главное, чтобы в нем вызывался конструктор базового класса Thread, в который передается имя потока.

И также в JThread переопределяется метод run(), код которого собственно и будет представлять весь тот код, который выполняется в потоке.

В методе main для запуска потока JThread у него вызывается метод start(), после чего начинается выполнение того кода, который определен в методе run:

```
new JThread("JThread").start();
```

Здесь в методе main в конструктор JThread передается произвольное название потока, и затем вызывается метод start(). По сути этот метод как раз и вызывает переопределенный метод run() класса JThread.

Обратите внимание, что главный поток завершает работу раньше, чем порожденный им дочерний поток JThread.

Аналогично созданию одного потока мы можем запускать сразу несколько потоков:

```
public static void main(String[] args) {

    System.out.println("Main thread started...");
    for(int i=1; i < 6; i++)
        new JThread("JThread " + i).start();
    System.out.println("Main thread finished...");
}
```

Ожидание завершения потока

При запуске потоков в примерах выше Main thread завершался до дочернего потока. Как правило, более распространенной ситуацией является случай, когда Main thread завершается самым последним. Для этого надо применить метод join(). В этом случае текущий поток будет ожидать завершения потока, для которого вызван метод join:

```
public static void main(String[] args) {

 System.out.println("Main thread started...");
 JThread t= new JThread("JThread ");
 t.start();
 try{
     t.join();
 }
 catch(InterruptedException e){
```

```
            System.out.printf("%s has been interrupted", t.getName());
        }
        System.out.println("Main thread finished...");
    }
```

Метод join() заставляет вызвавший поток (в данном случае Main thread) ожидать завершения вызываемого потока, для которого и применяется метод join (в данном случае JThread).

Если в программе используется несколько дочерних потоков, и надо, чтобы Main thread завершался после дочерних, то для каждого дочернего потока надо вызвать метод join.

Реализация интерфейса Runnable

Другой способ определения потока представляет реализация интерфейса Runnable. Этот интерфейс имеет один метод run:

```
interface Runnable{

    void run();
}
```

В методе run() собственно определяется весь тот код, который выполняется при запуске потока.

После определения объекта Runnable он передается в один из конструкторов класса Thread:

```
Thread(Runnable runnable, String threadName)
```

Для реализации интерфейса определим следующий класс MyThread:

```
class MyThread implements Runnable {


    public void run(){

        System.out.printf("%s started... \n",
Thread.currentThread().getName());
        try{
            Thread.sleep(500);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.printf("%s finished... \n",
Thread.currentThread().getName());
    }
```

```
}

public class Program {

    public static void main(String[] args) {

        System.out.println("Main thread started...");
        Thread myThread = new Thread(new MyThread(),"MyThread");
        myThread.start();
        System.out.println("Main thread finished...");
    }
}
```

Завершение и прерывание потока

Примеры потоков ранее представляли поток как последовательный набор операций. После выполнения последней операции завершался и поток. Однако нередко имеет место и другая организация потока в виде бесконечного цикла. Например, поток сервера в бесконечном цикле прослушивает определенный порт на предмет получения данных. И в этом случае мы также можем предусмотреть механизм завершения потока.

Завершение потока

Распространенный способ завершения потока представляет опрос логической переменной. И если она равна, например, false, то поток завершает бесконечный цикл и заканчивает свое выполнение.

Определим следующий класс потока:

```
class MyThread implements Runnable {

    private boolean isActive;

    void disable(){
        isActive=false;
    }

    MyThread(){
       isActive = true;
    }

    public void run(){

        System.out.printf("%s started... \n", Thread.currentThread().getName());
        int counter=1; // счетчик циклов
        while(isActive){
            System.out.println("Loop " + counter++);
            try{
                Thread.sleep(400);
            }
```

```
            catch(InterruptedException e){
                System.out.println("Thread has been interrupted");
            }
        }
        System.out.printf("%s finished... \n", Thread.currentThread().getName());
    }
}
```

Переменная isActive указывает на активность потока. С помощью метода disable() мы можем сбросить состояние этой переменной.

Теперь используем этот класс:

```
public static void main(String[] args) {

    System.out.println("Main thread started...");
    MyThread myThread = new MyThread();
    new Thread(myThread,"MyThread").start();

    try{
        Thread.sleep(1100);

        myThread.disable();

        Thread.sleep(1000);
    }
    catch(InterruptedException e){
        System.out.println("Thread has been interrupted");
    }
    System.out.println("Main thread finished...");
}
```

Итак, вначале запускается дочерний поток: `new Thread(myThread,"MyThread").start()`. Затем на 1100 миллисекунд останавливаем Main thread и потом вызываем метод myThread.disable(), который переключает в потоке флаг isActive. И дочерний поток завершается.

Метод interrupt

Еще один способ вызова завершения или прерывания потока представляет метод interrupt(). Вызов этого метода устанавливает у потока статус, что он прерван. Сам метод возвращает true, если поток может быть прерван, в ином случае возвращается false.

При этом сам вызов этого метода НЕ завершает поток, он только устанавливает статус: в частности, метод isInterrupted() класса Thread будет возвращать значение true. Мы можем проверить значение возвращаемое данным методом и прозвести некоторые действия. Например:

```
class JThread extends Thread {

    JThread(String name){
        super(name);
    }
    public void run(){

        System.out.printf("%s           started...           \n",
Thread.currentThread().getName());
        int counter=1; // счетчик циклов
        while(!isInterrupted()){

            System.out.println("Loop " + counter++);
        }
        System.out.printf("%s           finished...          \n",
Thread.currentThread().getName());
    }
}
public class Program {

    public static void main(String[] args) {

        System.out.println("Main thread started...");
        JThread t = new JThread("JThread");
        t.start();
        try{
            Thread.sleep(150);
            t.interrupt();

            Thread.sleep(150);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.println("Main thread finished...");
    }
}
```

В классе, который унаследован от Thread, мы можем получить статус текущего потока с помощью метода isInterrupted(). И пока этот метод возвращает false, мы можем выполнять цикл. А после того, как будет вызван метод interrupt, isInterrupted() возвратит true, и соответственно произойдет выход из цикла.

Если основная функциональность заключена в классе, который реализует интерфейс Runnable, то там можно проверять статус потока с помощью метода Thread.currentThread().isInterrupted():

```
class MyThread implements Runnable {
```

```
        public void run(){

            System.out.printf("%s            started...            \n",
Thread.currentThread().getName());
            int counter=1; // счетчик циклов
            while(!Thread.currentThread().isInterrupted()){

                System.out.println("Loop " + counter++);
            }
            System.out.printf("%s           finished...           \n",
Thread.currentThread().getName());
        }
    }
    public class Program {

        public static void main(String[] args) {

            System.out.println("Main thread started...");
            MyThread myThread = new MyThread();
            Thread t = new Thread(myThread,"MyThread");
            t.start();
            try{
                Thread.sleep(150);
                t.interrupt();

                Thread.sleep(150);
            }
            catch(InterruptedException e){
                System.out.println("Thread has been interrupted");
            }
            System.out.println("Main thread finished...");
        }
    }
```

Однако при получении статуса потока с помощью метода isInterrupted() следует учитывать, что если мы обрабатываем в цикле исключение InterruptedException в блоке catch, то при перехвате исключения статус потока автоматически сбрасывается, и после этого isInterrupted будет возвращать false.

Например, добавим в цикл потока задержку с помощью метода sleep:

```
    public void run(){

        System.out.printf("%s             started...             \n",
Thread.currentThread().getName());
        int counter=1; // счетчик циклов
        while(!isInterrupted()){

            System.out.println("Loop " + counter++);
```

```
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){
                System.out.println(getName() + " has been interrupted");
                System.out.println(isInterrupted());    // false
                interrupt();    // повторно сбрасываем состояние
            }
        }
        System.out.printf("%s             finished...             \n",
Thread.currentThread().getName());
    }
```

Когда поток вызовет метод interrupt, метод sleep сгенерирует исключение InterruptedException, и управление перейдет к блоку catch. Но если мы проверим статус потока, то увидим, что метод isInterrupted возвращает false. Как вариант, в этом случае мы можем повторно прервать текущий поток, опять же вызвав метод interrupt(). Тогда при новой итерации цикла while метода isInterrupted возвратит true, и поизойдет выход из цикла.

Либо мы можем сразу же в блоке catch выйти из цикла с помощью break:

```
    while(!isInterrupted()){

        System.out.println("Loop " + counter++);
        try{
            Thread.sleep(100);
        }
        catch(InterruptedException e){
            System.out.println(getName() + " has been interrupted");

            break;  // выход из цикла
        }
    }
```

Если бесконечный цикл помещен в конструкцию try...catch, то достаточно обработать InterruptedException:

```
    public void run(){

        System.out.printf("%s             started...             \n",
Thread.currentThread().getName());
        int counter=1; // счетчик циклов
        try{
            while(!isInterrupted()){
                System.out.println("Loop " + counter++);
                Thread.sleep(100);
            }
        }
```

```
        catch(InterruptedException e){
            System.out.println(getName() + " has been interrupted");
        }

        System.out.printf("%s             finished...             \n",
Thread.currentThread().getName());
    }
```

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. Например, определим следующий код:

```
public class Program {

    public static void main(String[] args) {

        CommonResource commonResource= new CommonResource();
        for (int i = 1; i < 6; i++){

            Thread t = new Thread(new CountThread(commonResource));
            t.setName("Thread "+ i);
            t.start();
        }
    }
}

class CommonResource{

    int x=0;
}

class CountThread implements Runnable{
```

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

Этот ресурс используется классом потока CountThread. Этот класс просто увеличивает в цикле значение x на единицу. Причем при входе в поток значение x=1:

То есть в итоге мы ожидаем, что после выполнения цикла res.x будет равно 4.

В главном классе программы запускается пять потоков. То есть мы ожидаем, что каждый поток будет увеличивать res.x с 1 до 4 и так пять раз. Но если мы посмотрим на результат работы программы, то он будет иным

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:

```
class CountThread implements Runnable{

    CommonResource res;
    CountThread(CommonResource res){
        this.res=res;
    }
    public void run(){
        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){}
            }
        }
    }
}
```

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.

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.

При применении оператора synchronized к методу пока этот метод не завершит выполнение, монопольный доступ имеет только один поток - первый, который начал его выполнение. Для применения synchronized к методу, изменим классы программы:

```
public class Program {
```

```
    public static void main(String[] args) {

        CommonResource commonResource= new CommonResource();
        for (int i = 1; i < 6; i++){

            Thread t = new Thread(new CountThread(commonResource));
            t.setName("Thread "+ i);
            t.start();
        }
    }
}

class CommonResource{

    int x;
    synchronized void increment(){
        x=1;
        for (int i = 1; i < 5; i++){
            System.out.printf("%s %d \n", Thread.currentThread().getName(), x);
            x++;
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
}

class CountThread implements Runnable{

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

    public void run(){
        res.increment();
    }
}
```

Результат работы в данном случае будет аналогичен примеру выше с блоком synchronized. Здесь опять в дело вступает монитор объекта CommonResource - общего объекта для всех потоков. Поэтому синхронизированным объявляется не метод run() в классе CountThread, а метод increment класса CommonResource. Когда первый поток начинает выполнение метода increment, он захватывает монитор объекта CommonResource. А все потоки также продолжают ожидать его освобождения.

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.

Рассмотрим, как мы можем использовать эти методы. Возьмем стандартную задачу из прошлой темы - "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():

```
public 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{
   private int product=0;
   public synchronized void get() {
      while (product<1) {
         try {
            wait();
         }
         catch (InterruptedException e) {
         }
      }
      product--;
      System.out.println("Покупатель купил 1 товар");
```

```
      System.out.println("Товаров на складе: " + product);
      notify();
   }
   public synchronized void put() {
       while (product>=3) {
         try {
            wait();
         }
         catch (InterruptedException e) {
         }
      }
      product++;
      System.out.println("Производитель добавил 1 товар");
      System.out.println("Товаров на складе: " + product);
      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();
        }
    }
}
```

Итак, здесь определен класс магазина, потребителя и покупателя. Производитель в методе run() добавляет в объект Store с помощью его метода put() 6 товаров. Потребитель в методе run() в цикле обращается к методу get объекта Store для получения этих товаров. Оба метода Store - put и get являются синхронизированными.

Для отслеживания наличия товаров в классе Store проверяем значение переменной product. По умолчанию товара нет, поэтому переменная равна 0. Метод get() - получение товара должен срабатывать только при наличии хотя бы одного товара. Поэтому в методе get проверяем, отсутствует ли товар.

Если товар отсутсвует, вызывается метод wait(). Этот метод освобождает монитор объекта Store и блокирует выполнение метода get, пока для этого же монитора не будет вызван метод notify().

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. Затем имитируется получение покупателем товара. Для этого выводится сообщение, и уменьшается значение product: product--. И в конце вызов метода notify() дает сигнал методу put() продолжить работу.

В методе put() работает похожая логика, только теперь метод put() должен срабатывать, если в магазине не более трех товаров. Поэтому в цикле проверяется наличие товара, и если товар уже есть, то освобождаем монитор с помощью wait() и ждем вызова notify() в методе get().

Таким образом, с помощью wait() в методе get() мы ожидаем, когда производитель добавит новый продукт. А после добавления вызываем notify(), как бы говоря, что на складе освободилось одно место, и можно еще добавлять.

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.

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.

Используем семафор в простом примере:

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

public 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 < 5; 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.

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:

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

public 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");
        }
    }
}
```

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:

```
import java.util.concurrent.Exchanger;

public 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());
        }
    }
}
```

В классе PutThread отправляет в буфер сообщение "Hello Java!":

message=exchanger.exchange(message);

Причем в ответ метод exchange возвращает данные, которые отправил в буфер другой поток. То есть происходит обмен данными. Хотя нам необязательно получать данные, мы можем просто их отправить:

exchanger.exchange(message);

Логика класса GetThread аналогична - также отправляется сообщение.

Класс Phaser

Класс Phaser позволяет синхронизировать потоки, представляющие отдельную фазу или стадию выполнения общего действия. Phaser определяет объект синхронизации, который ждет, пока не завершится определенная фаза. Затем Phaser переходит к следующей стадии или фазе и снова ожидает ее завершения.

Для создания объекта Phaser используется один из конструкторов:

```
Phaser()
Phaser(int parties)
Phaser(Phaser parent)
Phaser(Phaser parent, int parties)
```

Параметр parties указывает на количество сторон (грубо говоря, потоков), которые должны выполнять все фазы действия. Первый конструктор создает объект Phaser без каких-либо сторон. Второй конструктор регистрирует передаваемое в конструктор количество сторон. Третий и четвертый конструкторы также устанавливают родительский объект Phaser.

Основные методы класса Phaser:

- int register(): регистрирует сторону, которая выполняет фазы, и возвращает номер текущей фазы - обычно фаза 0
- int arrive(): сообщает, что сторона завершила фазу и возвращает номер текущей фазы

- int arriveAndAwaitAdvance(): аналогичен методу arrive, только при этом заставляет phaser ожидать завершения фазы всеми остальными сторонами

- int arriveAndDeregister(): сообщает о завершении всех фаз стороной и снимает ее с регистрации. Возвращает номер текущей фазы или отрицательное число, если синхронизатор Phaser завершил свою работу
- int getPhase(): возвращает номер текущей фазы

При работае с классом Phaser обычно сначала создается его объект. Далее нам надо зарегистрировать все участвующие стороны. Для регистрации в каждой стороне вызывается метод register(), либо можно обойтись и без этого метода, передав нужное количество сторон в конструктор Phaser.

Затем каждая сторона выполняет некоторый набор действий, составляющих фазу. А синхронизатор Phaser ждет, пока все стороны не завершат выполнение фазы. Чтобы сообщить синхронизатору, что фаза завершена, сторона должна вызвать метод arrive() или arriveAndAwaitAdvance(). После этого синхронизатор переходит к следующей фазе.

```
import java.util.concurrent.Phaser;

public class Program {

    public static void main(String[] args) {

        Phaser phaser = new Phaser(1);
        new Thread(new PhaseThread(phaser, "PhaseThread 1")).start();
        new Thread(new PhaseThread(phaser, "PhaseThread 2")).start();

        // ждем завершения фазы 0
        int phase = phaser.getPhase();
        phaser.arriveAndAwaitAdvance();
        System.out.println("Фаза " + phase + " завершена");
        // ждем завершения фазы 1
        phase = phaser.getPhase();
        phaser.arriveAndAwaitAdvance();
        System.out.println("Фаза " + phase + " завершена");

        // ждем завершения фазы 2
        phase = phaser.getPhase();
        phaser.arriveAndAwaitAdvance();
        System.out.println("Фаза " + phase + " завершена");

        phaser.arriveAndDeregister();
    }
}
```

```
class PhaseThread implements Runnable{

    Phaser phaser;
    String name;

    PhaseThread(Phaser p, String n){

        this.phaser=p;
        this.name=n;
        phaser.register();
    }
    public void run(){

        System.out.println(name + " выполняет фазу " + phaser.getPhase());
        phaser.arriveAndAwaitAdvance(); // сообщаем, что первая фаза достигнута

        System.out.println(name + " выполняет фазу " + phaser.getPhase());
        phaser.arriveAndAwaitAdvance(); // сообщаем, что вторая фаза достигнута

        System.out.println(name + " выполняет фазу " + phaser.getPhase());
        phaser.arriveAndDeregister(); // сообщаем о завершении фаз и удаляем с регистрации
объекты
    }
}
```

Итак, здесь у нас фазы выполняются тремя сторонами - главным потоком и двумя потоками PhaseThread. Поэтому при создании объекта Phaser ему передается число 1 - главный поток, а в конструкторе PhaseThread вызывается метод register(). Мы в принципе могли бы не использовать метод register, но тогда нам надо было бы указать Phaser phaser = new Phaser(3), так как у нас три стороны.

Фаза в каждой стороне представляет минимальный примитивный набор действий: для потоков PhaseThread это вывод сообщения, а для главного потока - подсчет текущей фазы с помощью метода getPhase(). При этом отсчет фаз начинается с нуля. Каждая сторона завершает выполнение фазы вызовом метода phaser.arriveAndAwaitAdvance(). При вызове этого метода пока последняя сторона не завершит выполнение текущей фазы, все остальные стороны блокируются.

После завершения выполнения последней фазы происходит отмена регистрации всех сторон с помощью метода arriveAndDeregister().

В данном случае получается немного путанный вывод. Так, сообщения о выполнении фазы 1 выводится после сообщения об окончании фазы 0. Что связано с многопоточностью - фазы завершились, но в одном потоке еще не выведено сообщение о завершении, тогда как другие потоки уже начали выполнение следующей фазы. В любом случае все это происходит уже после завершения фазы.

Но чтобы было более наглядно, мы можем использовать sleep в потоках:

```
public void run(){

        System.out.println(name + " выполняет фазу " + phaser.getPhase());
        phaser.arriveAndAwaitAdvance(); // сообщаем, что первая фаза достигнута
        try{
            Thread.sleep(200);
        }
        catch(InterruptedException ex){
            System.out.println(ex.getMessage());
        }

        System.out.println(name + " выполняет фазу " + phaser.getPhase());
        phaser.arriveAndAwaitAdvance(); // сообщаем, что вторая фаза достигнута
        try{
            Thread.sleep(200);
        }
        catch(InterruptedException ex){
            System.out.println(ex.getMessage());
        }
        System.out.println(name + " выполняет фазу " + phaser.getPhase());
        phaser.arriveAndDeregister(); // сообщаем о завершении фаз и удаляем с регистрации
объекты
    }
```

Блокировки. ReentrantLock

Для управления доступом к общему ресурсу в качестве альтернативы оператору synchronized мы можем использовать блокировки. Функциональность блокировок заключена в пакете java.util.concurrent.locks.

Вначале поток пытается получить доступ к общему ресурсу. Если он свободен, то на него накладывает блокировку. После завершения работы блокировка с общего ресурса снимается. Если же ресурс не свободен и на него уже наложена блокировка, то поток ожидает, пока эта блокировка не будет снята.

Классы блокировок реализуют интерфейс Lock, который определяет следующие методы:

1. void lock(): ожидает, пока не будет получена блокировка

2. void lockInterruptibly() throws InterruptedException: ожидает, пока не будет получена блокировка, если поток не прерван

3. boolean tryLock(): пытается получить блокировку, если блокировка получена, то возвращает true. Если блокировка не получена, то

возвращает false. В отличие от метода lock() не ожидает получения блокировки, если она недоступна

4. void unlock(): снимает блокировку

5. Condition newCondition(): возвращает объект Condition, который связан с текущей блокировкой

Организация блокировки в общем случае довольно проста: для получения блокировки вызывается метод lock(), а после окончания работы с общими ресурсами вызывается метод unlock(), который снимает блокировку.

Объект Condition позволяет управлять блокировкой.

Как правило, для работы с блокировками используется класс ReentrantLock из пакета java.util.concurrent.locks. Данный класс реализует интерфейс Lock.

Для примера возьмем код из темы про оператор synchronized и перепишем данный код с использованием заглушки ReentrantLock:

```
import java.util.concurrent.locks.ReentrantLock;

public class Program {

    public static void main(String[] args) {

        CommonResource commonResource= new CommonResource();
        ReentrantLock locker = new ReentrantLock(); // создаем заглушку
        for (int i = 1; i < 6; i++){

            Thread t = new Thread(new CountThread(commonResource, locker));
            t.setName("Thread "+ i);
            t.start();
        }
    }
}

class CommonResource{

    int x=0;
}

class CountThread implements Runnable{

    CommonResource res;
    ReentrantLock locker;
    CountThread(CommonResource res, ReentrantLock lock){
        this.res=res;
```

```
        locker = lock;
    }
    public void run(){

        locker.lock(); // устанавливаем блокировку
        try{
            res.x=1;
            for (int i = 1; i < 5; i++){
                System.out.printf("%s %d \n", Thread.currentThread().getName(), res.x);
                res.x++;
                Thread.sleep(100);
            }
        }
        catch(InterruptedException e){
            System.out.println(e.getMessage());
        }
        finally{
            locker.unlock(); // снимаем блокировку
        }
    }
}
```

Здесь также используется общий ресурс CommonResource, для управления которым создается пять потоков. На входе в критическую секцию устанавливается заглушка:

После этого только один поток имеет доступ к критической секции, а остальные потоки ожидают снятия блокировки. В блоке finally после всей окончания основной работы потока эта блокировка снимается. Причем делается это обязательно в блоке finally, так как в случае возникновения ошибки все остальные потоки окажутся заблокированными.

Условия в блокировках

Применение условий в блокировках позволяет добиться контроля над управлением доступом к потокам. Условие блокировки представлет собой объект интерфейса Condition из пакета java.util.concurrent.locks.

Применение объектов Condition во многом аналогично использованию методов wait/notify/notifyAll класса Object, которые были рассмотрены в одной из прошлых тем. В частности, мы можем использовать следующие методы интерфейса Condition:

1. await: поток ожидает, пока не будет выполнено некоторое условие и пока другой поток не вызовет методы signal/signalAll. Во многом аналогичен методу wait класса Object

2. signal: сигнализирует, что поток, у которого ранее был вызван метод await(), может продолжить работу. Применение аналогично использованию методу notify класса Object

3. signalAll: сигнализирует всем потокам, у которых ранее был вызван метод await(), что они могут продолжить работу. Аналогичен методу notifyAll() класса Object

Эти методы вызываются из блока кода, который попадает под действие блокировки ReentrantLock. Сначала, используя эту блокировку, нам надо получить объект Condition:

```
ReentrantLock locker = new ReentrantLock();
Condition condition = locker.newCondition();
```

Как правило, сначала проверяется условие доступа. Если соблюдается условие, то поток ожидает, пока условие не изменится:

```
while (условие)
    condition.await();
```

После выполнения всех действий другим потокам подается сигнал об изменении условия:

```
    condition.signalAll();
```

Важно в конце вызвать метод signal/signalAll, чтобы избежать возможности взаимоблокировки потоков.

Для примера возьмем задачу из темы про методы wait/notify и изменим ее, применяя объект Condition.

Итак, у нас есть склад, где могут одновременно быть размещено не более 3 товаров. И производитель должен произвести 5 товаров, а покупатель должен эти товары купить. В то же время покупатель не может купить товар, если на складе нет никаких товаров:

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public 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{
   private int product=0;
   ReentrantLock locker;
   Condition condition;

   Store(){
```

```
        locker = new ReentrantLock(); // создаем блокировку
        condition = locker.newCondition(); // получаем условие, связанное с блокировкой
    }

    public void get() {

        locker.lock();
        try{
            // пока нет доступных товаров на складе, ожидаем
            while (product<1)
                condition.await();

            product--;
            System.out.println("Покупатель купил 1 товар");
            System.out.println("Товаров на складе: " + product);

            // сигнализируем
            condition.signalAll();
        }
        catch (InterruptedException e){
            System.out.println(e.getMessage());
        }
        finally{
            locker.unlock();
        }
    }
    public void put() {

        locker.lock();
        try{
            // пока на складе 3 товара, ждем освобождения места
            while (product>=3)
                condition.await();

            product++;
            System.out.println("Производитель добавил 1 товар");
            System.out.println("Товаров на складе: " + product);
            // сигнализируем
            condition.signalAll();
        }
        catch (InterruptedException e){
            System.out.println(e.getMessage());
        }
        finally{
            locker.unlock();
        }
    }
}
// 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();
        }
    }
}
```

Задание на практическую работу

Используя материалы данной практической работы необходимо написать многопоточную программу, в которой два потока записывают строку в стандартный вывод, по образцу PING PONG PING PONG PING PONG. Программа должна работать следующим образом:

- 1-й поток печатает «Ping» и переходит в состояние ожидания.
- 2-й поток выходит из состояния ожидания, печатает «Pong», уведомляет 1-й поток, возвращается в состояние ожидания.
- 1-й поток выходит из состояния ожидания, печатает «Pшng», уведомляет 2-й поток, возвращается в состояние ожидания.
- Шаги 2 и 3 повторяются и печатают «Ping Pong».
-

Программа должна быть реализована только с использованием Wait Notify, либо ReentrantLock.