7361

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. В.Ф. УТКИНА

МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ НА JAVA. СИНХРОНИЗАТОРЫ

Методические указания к лабораторной работе

Рязань 2022

УДК 681.3

Многопоточное программирование на Java. Синхронизаторы: методические указания к лабораторной работе / Рязан. гос. радиотехн. ун-т.; сост.: А.А. Митрошин, В.Г. Псоянц. Рязань, 2022. 16 с.

Содержат описание лабораторной работы, используемой в курсе «Параллельное программирование».

Предназначены для студентов всех форм обучения направления подготовки «Информатика и вычислительная техника».

Могут использоваться как методические указания к лабораторным и практическим занятиям в других курсах, в которых изучается язык программирования Java.

Табл. 3. Библиогр.: 3 назв.

Java, многопоточное программирование, синхронизаторы

Печатается по решению редакционно-издательского совета Рязанского государственного радиотехнического университета им. В.Ф Уткина.

Рецензент: кафедра САПР вычислительных средств Рязанского государственного радиотехнического университета (зав. кафедрой засл. деят. науки и техники РФ В.П.Корячко)

Многопоточное программирование на Java. Синхронизаторы

Составители: М и т р о ш и н Александр Александрович
П с о я н ц Владимир Грикорович

Редактор М.Е. Цветкова
Корректор И.В. Черникова
Подписано в печать 30.06.22. Формат бумаги 60×84 1/16.
Бумага писчая. Печать трафаретная. Усл. печ. л. 1,0.
Тираж 25 экз. Заказ
Рязанский государственный радиотехнический университет.
390005, Рязань, ул. Гагарина, 59/1.
Редакционно-издательский центр РГРТУ.

Синхронизаторы

Синхронизатор (synchronizer) — это объект, который координирует поток управления в остальных потоках, основываясь на их состоянии. В качестве синхронизаторов могут выступать блокирующие очереди, семафоры, барьеры и защелки. Существует несколько классов синхронизаторов.

Semaphore (семафор) — объект синхронизации, ограничивающий одновременный доступ к общему ресурсу нескольким потокам с использованием счетчика. При запросе разрешения и значении счетчика, большем нуля, доступ предоставляется и значение счетчика уменьшается; если значение счетчика равно нулю, то доступ запрещается. При освобождении ресурса значение счетчика семафора увеличивается. Количество разрешений семафора определяется в конструкторе. Второй конструктор семафора включает дополнительный параметр «справедливости», определяющий порядок предоставления разрешения ожидающим доступа потокам.

CountDownLatch (защелка с обратным отсчетом) — объект синхронизации потоков, блокирующий один или несколько потоков до тех пор, пока не будет выполнено определенное количество условий. Количество условий задается счетчиком, значение которого при выполнении условия уменьшается. При обнулении счетчика, т.е. при выполнении всех условий, блокировки с заблокированных потоков снимаются и они продолжают выполнение. Важно, *что счетчик одноразовый и не может быть инициализирован вновь*.

CyclicBarrier — объект синхронизации типа «Барьер». Барьерная синхронизация останавливает исполняемый поток в определенном месте в ожидании прихода остальных потоков. Как только все потоки достигнут барьера, барьер срабатывает (снимается) и выполнение потоков продолжается. Циклический барьер CyclicBarrier так же, как и CountDownLatch, использует счетчик и похож на него. Отличие заключается в том, что защелку CountDownLatch нельзя использовать повторно после того, как её счётчик обнулится, а CyclicBarrier можно.

Exchanger — объект синхронизации, используемый для двустороннего обмена данными *между двумя потоками*. При обмене данными допускаются null значения, что позволяет использовать класс для односторонней передачи объекта или как синхронизатор двух потоков. Обмен данными выполняется с помощью вызова метода *exchange* и сопровождается блокировкой потока. Как только другой (второй) поток вызовет метод *exchange*, то синхронизатор *Exchanger* выполнит обмен данными между потоками.

Phaser — объект синхронизации типа «Барьер», но в отличие от *CyclicBarrier* может иметь несколько барьеров (фаз), и количество участников на каждой фазе может быть различным.

Семафоры

Класс Semaphore управляет набором разрешений (permits). Начальное число разрешений передается конструктору класса Semaphore. Действия могут приобретать разрешения и освобождать их, когда работа с ними закончена. Если разрешения отсутствуют, то продвижение блокируется до тех пор, пока не появится разрешение. После того, как разрешение больше не нужно, оно возвращается семафору.

Двоичный семафор с начальным счетчиком, равным единице, можно использовать в качестве мьютекса с семантикой повторно невходимой блокировки, и любой владелец единственного разрешения будет владеть мьютексом.

Класс Semaphore

```
public class Semaphore extends Object implements Serializable
```

Конструкторы класса Semaphore

`Semaphore(int permits)` – создает семафор с заданным количеством разрешений и «несправедливой» установкой доступности.

`Semaphore`(int permits, boolean fair) - создает семафор с заданным количеством разрешений и установленным признаком «справедливости» `fair`.

По умолчанию ожидающим потокам предоставляется разрешение в неопределенном (случайном) порядке. Если же использовать второй конструктор и параметру справедливости fair установить значение true, то разрешения будут предоставляться ожидающим исполнения потокам в том порядке, в котором они разрешение запрашивали.

Методы класса Semaphore приведены в табл. 1.

Таблица 1. Методы класса Semaphore

Спецификатор	Метод и его описание
`void`	`acquire()` Получает разрешение этого семафора, блокируется до тех пор, пока разрешение не будет получено или поток не будет прерван

Продолжение таблицы 1

`void`	`acquire(int permits)` Получает количество разрешений этого семафора, передаваемого в качестве параметра. Блокируется до тех пор, пока нужное количество разрешений не будет получено или поток не будет прерван
`void`	`acquireUninterruptibly()` Получает разрешение этого семафора, блокируется до тех пор, пока разрешение не будет получено
`void`	`acquireUninterruptibly(int permits)` Получает количество разрешений этого семафора, передаваемого в качестве параметра. Блокируется до тех пор, пока нужное количество разрешений не будет получено
`int`	`availablePermits()` Возвращает текущее количество разрешений, доступное в этом семафоре
`int`	`drainPermits()` Получает и возвращает количество всех разрешений, которые сейчас доступны
`protected Collection <Thread>`	`getQueuedThreads()` Возвращает коллекцию, содержащую потоки, ожидающие получение разрешения
`int`	`getQueueLength()` Возвращает оценку количества потоков, ожидающих получения разрешения
`boolean`	`hasQueuedThreads()` Возвращает true, если в очереди есть хотя бы один поток, ожидающий получения разрешения
`boolean`	`isFair()` Возвращает `true`, если установлен режим справедливой выдачи разрешений
`protected void`	`reducePermits(int reduction)` Уменьшает количество доступных разрешений на передаваемое значение
`void`	`release()` Освобождает разрешение, возвращая его семафору
`void`	`release(int permits)` Возвращает передаваемое в качестве параметра значение разрешений, возвращая их семафору

Окончание таблицы 1

`String`	`toString()` Возвращает строковое представление семафора
`boolean`	`tryAcquire()` Получает разрешение этого семафора, но только в том случае, если одно разрешение доступно во время вызова
`boolean`	`tryAcquire(int permits)` Получает переданное в качестве параметра количество разрешений этого семафора, но только в том случае, если необходимое количество разрешений доступно во время вызова
`boolean`	`tryAcquire(int permits, long timeout, TimeUnit unit)` Получает переданное в качестве параметра количество разрешений, если такое количество разрешений доступно в этом семафоре и текущий поток не был прерван
`boolean`	`tryAcquire(long timeout, TimeUnit unit)` Получает одно разрешение этого семафора, но только в том случае, если одно разрешение доступно в течение переданного в качестве параметра времени и текущий поток не был прерван

Пример использования семафоров

В примере несколько всадников с лошадьми должны пройти контроль перед скачками. Количество мест контроля меньше количества всадников, поэтому некоторые всадники будут дожидаться, пока одно из мест контроля не освободится.

Общий ресурс CONTROL_PLACES (места контроля), используемый всеми потоками, определен как synchronized. С помощью семафора осуществляется контроль доступа к CONTROL_PLACES только одному потоку в каждый момент времени.

```
import java.util.concurrent.Semaphore;
public class SemaphoreExample
{
//Количество мест контроля
private static final int COUNT_CONTROL_PLACES = 4;
//Количество всадников
private static final int COUNT_RIDERS = 25;
// Флаги мест контроля
private static boolean[] CONTROL_PLACES = null;
```

```
// Семафор
private static Semaphore SEMAPHORE = null;
// Класс, описывающий всадника с лошадью.
//  Класс реализует интерфейс Runnable, чтобы его
экземпляры можно было
// выполнять в отдельных потоках
public static class Rider implements Runnable
{
//Номер всадника
private int riderNum;
//Конструктор
public Rider(int riderNum)
{this.riderNum = riderNum;}
@Override
public void run() {
System.out.printf("Всадник %d подошел к зоне
контроля\n", riderNum);
try {
// Запрос разрешения занять место осмотра.
SEMAPHORE.acquire();
System.out.printf("\tвсадник %d проверяет, есть
свободное место осмотра\n",
riderNum);
// Переменной controlNum, в которой будет храниться
номер найденного
// места осмотра, присваиваем значение несуществующего
места контроля.
int controlNum = -1;
// Ищем свободное место и подходим к одному из мест
контроля.
// Поиск осуществляем в синхронизованном блоке.
synchronized (CONTROL_PLACES){
for (int i = 0; i < COUNT_CONTROL_PLACES; i++)
//Есть ли свободное место?У i свободного места
CONTROL_PLACES[i]=true
if (CONTROL_PLACES[i]) {
// Занимаем свободное место контроля.
CONTROL_PLACES[i] = false;
//Сохраняем в переменной controlNum и выводим номер
занятого места контроля
controlNum = i;
System.out.printf("\t\tвсадник %d подошел к контроллеру
%d.\n", riderNum, i);
break;
}
}
// Моделируем время проверки лошади и всадника
Thread.sleep((int)(Math.random()*10+1)*1000);
```

```
// Освобождение места контроля. Освобождаем в
синхронизованном блоке
synchronized (CONTROL_PLACES) {
CONTROL_PLACES[controlNum] = true;
}
// Возвращаем одно разрешение семафору
SEMAPHORE.release();
System.out.printf("Всадник %d завершил проверку\n",
riderNum);
} catch (InterruptedException e) {}
}
}
public static void main(String[] args) throws
InterruptedException
{
// Определяем количество мест контроля
CONTROL_PLACES = new boolean[COUNT_CONTROL_PLACES];
// Флаги мест контроля [true-свободно, false-занято]
// В начальный момент все места свободны и все
CONTROL_PLACES[i] = true
for (int i = 0; i < COUNT_CONTROL_PLACES; i++)
CONTROL_PLACES[i] = true;
/*Определяем семафор со следующими параметрами:
*     - количество разрешений равно количеству мест
контроля
*     - флаг справедливости очередности установлен:
fair=true
*   (очередь first_in-first_out – первым пришел, первым
получил разрешение)
*/
SEMAPHORE = new Semaphore(CONTROL_PLACES.length,true);
// Создаем и стартуем потоки для всех всадников
for (int i = 1; i <= COUNT_RIDERS; i++) {
new Thread(new Rider(i)).start();
Thread.sleep(400);
}}}
```

Защелка с обратным отсчетом

Иногда требуется, чтобы поток исполнения находился в режиме ожидания до тех пор, пока не наступит некоторое количество событий (одно или более). Для этих целей используется класс CountDownLatch, реализующий защелку с обратным отсчетом. Объект этого класса изначально создается с количеством событий, которые должны произойти до того момента, как будет снята блокировка. Всякий раз, когда происходит событие, значение счетчика уменьшается. Как только значение счетчика достигнет нуля, блокировка будет снята.

Класс *CountDownLatch*

```
public class CountDownLatch extends Object
```

Класс CountDownLatch имеет единственный конструктор, параметром которого является количество событий, которые должны произойти до того момента, когда блокировка будет снята:
`CountDownLatch(int count).`

Методы класса CountDownLatch описаны в табл. 2.

Таблица 2. Методы класса CountDownLatch

Модификатор и тип	Метод и его описание
`void`	`await()` Блокирует текущий поток до тех пор, пока счетчик с обратным отсчетом защелки не станет равным 0
`boolean`	`await(long timeout, TimeUnit unit)` Блокирует текущий поток до тех пор, пока счетчик с обратным отсчетом защелки не станет равным 0 или не истечет время ожидания
`void`	`countDown()` Уменьшает на 1 значение счетчика защелки
`long`	`getCount()` Возвращает текущее значение счетчика
`String`	`toString()` Возвращает строковое представление защелки

Пример использования CountDownLatch

Несколько всадников должны подъехать к стартовому барьеру. Как только все всадники выстроятся перед стартовым барьером, будут даны команды «На старт», «Внимание», «Марш». После этого начнутся скачки. Объект синхронизации CountDownLatch выполняет роль счетчика количества всадников и команд.

```
import java.util.concurrent.CountDownLatch;
public class CountDownLatchExample
{
// Количество всадников
private static final int RIDERS_COUNT = 5;
// Объект синхронизации
private static CountDownLatch LATCH;
// Условная длина трассы
private static final int trackLength = 3000;
public static class Rider implements Runnable
{// номер всадника
 private int riderNumber;
```

```
 // скорость всадника постоянная
 private int riderSpeed ;
 //Конструктор
 public Rider(int riderNumber, int riderSpeed)
        {this.riderNumber = riderNumber;
         this.riderSpeed  = riderSpeed;}
 @Override
 public void run() {
 try {
 System.out.printf("Всадник   %d   вышел   на   старт\n",
riderNumber);
 //  Уменьшаем счетчик на 1
 LATCH.countDown();
 // Метод await блокирует поток до тех пор,
 // пока счетчик CountDownLatch не обнулится
 LATCH.await();
 // Ожидание, пока всадник не преодолеет трассу
 Thread.sleep(trackLength / riderSpeed * 10);
 System.out.printf("Всадник      %d      финишировал\n",
riderNumber);
 } catch (InterruptedException e) {}
 }}
 public static Rider createRider(final int number)
    {return new Rider(number, (int)(Math.random() * 10 +
+5));}
 public   static   void   main(String[]   args)   throws
InterruptedException
{// Определение объекта синхронизации. Нам нужно, чтобы
старт произошел
// после того, как:
// - сначала на старт выйдут все RIDERS_COUNT всадников;
//   -   затем   прозвучат   три   команды   ("На   старт",
"Внимание", "Марш")
// То есть произойдет только после того, как произойдут
эти
// RIDERS_COUNT + 3 события
LATCH = new CountDownLatch(RIDERS_COUNT + 3);
// Создание потоков всадников
for (int i = 1; i <= RIDERS_COUNT; i++)
{new Thread(createRider(i)).start();
Thread.sleep(1000);}
//  Проверка   наличия   всех   всадников   на   старте.   Если
значение счетчика больше
//  трех,  значит  прибыли  еще  не  все  всадники  и  надо
немного подождать
while (LATCH.getCount() > 3)
Thread.sleep(100);
// Все всадники на месте. Ждем еще немножко …
```

```
Thread.sleep(1000);
// На старт!
System.out.println("На старт!");
LATCH.countDown();  // Уменьшаем счетчик на 1
Thread.sleep(1000);
// Внимание!!
System.out.println("Внимание!");
LATCH.countDown(); // Уменьшаем счетчик на 1
Thread.sleep(1000);
// Марш!!!
System.out.println("Марш!");
LATCH.countDown(); // Уменьшаем счетчик на 1
// Счетчик обнулен, и все ожидающие этого события потоки
разблокированы
}}
```

Циклический барьер *CyclicBarrier*

```
public class CyclicBarrier extends Object
```

Объект синхронизации *CyclicBarrier* представляет собой барьерную синхронизацию, часто используемую в вычислениях. Особенно эффективно использование барьеров при циклических расчетах. При барьерной синхронизации алгоритм расчета делится на несколько потоков. С помощью барьера организуют точку сбора частичных результатов вычислений, в которой подводится итог этапа вычислений.

Барьер для потоков означает, что каждый поток должен остановиться в определенном месте (у барьера) и ожидать прихода остальных потоков. Как только все потоки достигнут барьера, их выполнение продолжится.

Конструкторы класса *CyclicBarrier*

Класс CyclicBarrier имеет два конструктора.

`CyclicBarrier(int parties)` – создает новый объект класса CyclicBarrier, который сработает, когда заданное в качестве параметра конструктора количество потоков (parties) будет ожидать у барьера; после этого все ожидающие потоки получат возможность выполняться.

`CyclicBarrier(int parties, Runnable barrierAction)` – в этом конструкторе дополнительно определяется класс, реализующий интерфейс Runnable, который должен быть запущен после прихода к барьеру parties потоков. Поток запускать в программе не следует, *CyclicBarrier это делает автоматически*.

Методы класса *CyclicBarrier*

Методы класса CyclicBarrier описаны в табл. 3.

Таблица 3. Методы класса CyclicBarrier

Модификатор и тип	Метод и его описание
`int`	`await()` Ждать, пока необходимое количество потоков (задается в конструкторе) не вызовет метод `await` этого циклического барьера. Возвращает индекс поступления текущего потока; возвращаемое значение getParties() – 1 соответствует прибытию первого потока, а 0 – последнего из необходимых для срабатывания барьера
`int`	`await(long timeout, TimeUnit unit)` Ждать, пока необходимое количество потоков не вызовет метод `await` этого циклического барьера или не истечет время тайм-аута `timeout`. Время ожидания указывается в единицах измерения времени unit объекта перечисления TimeUnit. В TimeUnit определены следующие значения: TimeUnit.DAYS – дни, TimeUnit.HOURS – часы, TimeUnit.MINUTES – минуты, TimeUnit.SECONDS – секунды, TimeUnit.MICROSECONDS – микросекунды, TimeUnit.MILLISECONDS – миллисекунды. Возвращает индекс поступления текущего потока; возвращаемое значение getParties() – 1 соответствует прибытию первого потока, а 0 – последнего из необходимых для срабатывания барьера
`int`	`getNumberWaiting()` Возвращает текущее количество потоков, ожидающих у барьера
`int`	`getParties()` Возвращает количество потоков, необходимых для срабатывания барьера
`boolean`	`isBroken()` Возвращает true, если барьер находится в поврежденном состоянии
`void`	`reset()` Сбрасывает барьер в исходное состояние

Пример использования CyclicBarrier

Пример моделирует работу паромной переправы. Паром может вместить только FerryBoat_size автомобилей. Количество автомобилей в примере - 9. Роль парома выполняет объект синхронизации FerryBarrier, которому в качестве второго параметра конструктора передается реализующий интерфейс Runnable класса FerryBoat. Как только Cars (в примере - 3) автомобилей (потоков) достигнут парома (барьера), автоматически будет запущен на выполнение экземпляр класса FerryBoat (паром отчалит), после завершения работы которого потоки (автомобили) продолжают свою работу.

```
import java.util.concurrent.CyclicBarrier;
public class CyclicBarrierExample
{
  private static CyclicBarrier FerryBarrier;
  private static final int FerryBoat_size = 3;
  private static final int Cars = 9;

  // Паром, переправляющий автомобили.
  // Экземпляр этого класса будет выполнен в отдельном
  //  потоке  барьером  после  того,  как  FerryBoat_size
потоков,
  // моделирующих автомобили, подойдет к барьеру

  public static class FerryBoat implements Runnable
    {
      @Override
      public void run() {
          try {
             // Задержка на переправе
             System.out.println("\nЗагрузка
автомобилей");
             Thread.sleep(500);
             System.out.println("Паром переправил
автомобили\n");
             } catch (InterruptedException e) {}
                  }
    }
    // Класс автомобиля
    public static class Car implements Runnable
    {
        private int carNumber;
        public Car(int carNumber) {
            this.carNumber = carNumber;
        }
        @Override
```

```
        public void run() {
            try {
        System.out.printf("К переправе подъехал
автомобиль %d\n", carNumber);
        // Вызов метода await при подходе к
        // барьеру; поток блокируется в ожидании
        // прихода остальных потоков
        FerryBarrier.await();
        // После того, как барьер сработает
        System.out.printf("Автомобиль    %d    продолжил
движение\n",

carNumber);
            } catch (Exception e) {}
        }
    }
    public  static  void  main(String[]  args)  throws
InterruptedException {
        FerryBarrier = new CyclicBarrier(FerryBoat_size,
new FerryBoat());
        for (int i = 1; i < Cars+1; i++) {
            new Thread(new Car(i)).start();
            Thread.sleep(400);
        }}}
```

Phaser

Phaser, как и CyclicBarrier, является реализацией шаблона синхронизации «Барьер». Этот класс позволяет синхронизировать потоки, представляющие отдельную фазу выполнения общего действия. Как и CyclicBarrier, Phaser является точкой синхронизации, в которой встречаются потоки. Когда все потоки на текущей фазе прибыли, Phaser переходит к следующей фазе и снова ожидает ее завершения.

Особенности Phaser по сравнению с CyclicBarrier:

- каждая фаза (цикл синхронизации) имеет номер;
- количество сторон-участников жестко не задано и может меняться: поток может регистрироваться в качестве участника и отменять свое участие;
- поток-участник не обязан ожидать, пока все остальные участники соберутся на барьере; чтобы продолжить работу, достаточно сообщить о своем прибытии;
- поток может и не быть участником барьера, чтобы ожидать его преодоления.

Объект Phaser чаще всего создается с помощью одного из конструкторов:

```
Phaser()
Phaser(int parties)
```

Параметр parties указывает на количество потоков-участников, которые будут выполнять фазы действия. Первый конструктор создает объект Phaser без каких-либо зарегистрированных потоков, при этом барьер закрыт. Второй конструктор регистрирует передаваемое в конструктор количество потоков. Барьер открывается, когда количество прибывших потоков совпадает с количеством зарегистрированных.

Основные методы:

- int register() — регистрирует поток, который выполняет фазы. Вызывается из потока, который хочет зарегистрироваться. Возвращает номер текущей фазы;
- int getPhase() — возвращает номер текущей фазы;
- int arriveAndAwaitAdvance() — указывает, что поток завершил выполнение фазы (прибыл). Поток приостанавливается до момента, пока все остальные потоки не закончат выполнять данную фазу. Возвращает номер текущей фазы;
- int arrive() — сообщает, что поток завершил фазу (прибыл), и возвращает номер фазы. При вызове данного метода поток не приостанавливается, а продолжает выполняться;
- int arriveAndDeregister() — сообщает о завершении всех фаз потоком и снимает его с регистрации. Возвращает номер текущей фазы;
- awaitAdvance(int phase) — если значение phase равно номеру текущей фазы, приостанавливает вызвавший его поток до его окончания.

Пример использования *Phaser*

```
import java.util.concurrent.Phaser;
public class Program {
    public static void main(String[] args) {
   // Создаем фазер. В параметре конструктора - 1, 
следовательно, для участия
   // в нулевой фазе зарегистрирован один поток.
        Phaser phaser = new Phaser(1);
   // Создаем два потока с именами PhaseThread-1 и 
PhaseThread-2
        new Thread(new PhaseThread(phaser, "PhaseThread-
1")).start();
        new Thread(new PhaseThread(phaser, "PhaseThread-
2")).start();
   // Получаем номер текущей фазы
```

```
        System.out.println("Основной поток: Текущая Фаза
фазер " + phaser.getPhase());
        System.out.println("Основной поток: число
зарегистрированных в фазере потоков " +
phaser.getRegisteredParties());
        System.out.println();
        try {
             Thread.sleep(2000);
             } catch (InterruptedException e) {
             e.printStackTrace();
             }
        System.out.println("Основной поток: сейчас
прибыло к фазеру потоков " +
phaser.getArrivedParties());
        System.out.println();
        // К фазеру прибыл основной поток приложения.
        // Поток будет задержан до тех пор, пока не
завершится нулевая фаза
        phaser.arriveAndAwaitAdvance();
        // Нулевая фаза завершена. Выводим сообщение и
продолжаем
        System.out.println("Основной поток: Фаза " +
(phaser.getPhase()-1) + " фазер завершен");
        System.out.println();
        // Получаем номер текущей фазы
        System.out.println("Основной поток: Текущая Фаза
фазера " + phaser.getPhase());
        System.out.println();
        // Основной поток прибыл к фазеру и ждет
завершения первой фазы
        phaser.arriveAndAwaitAdvance();
        System.out.println("Основной поток: Фаза " +
(phaser.getPhase()-1) + " фазер завершен");
        System.out.println();
        // Основной поток прибыл к фазеру и ждет
завершения второй фазы
        phaser.arriveAndAwaitAdvance();
        System.out.println("Основной поток: Фаза " +
(phaser.getPhase()-1) + " фазер завершен");
        System.out.println();

        // Отмена регистрации основного потока в фазере
        phaser.arriveAndDeregister();
        System.out.println("Основной поток: Вычисления
завершены");
    }
}
//-------------------------------------------------------
```

```
class PhaseThread implements Runnable{
    Phaser phaser;
    String name;
    PhaseThread(Phaser p, String n){
        this.phaser=p;
        this.name=n;
        // Регистрация потока в фазере
        phaser.register();
    }
    @SuppressWarnings("deprecation")
      public void run()
    {
      // Поток что-то делает какое-то время
      try {
            Thread.sleep((int)Math.random()*1000+500);
       } catch (InterruptedException e) {
            e.printStackTrace();
      }
       System.out.println(name + " выполняет фазу " +
phaser.getPhase());
     // Поток прибыл и ожидает завершения нулевой фазы
на фазере
       phaser.arriveAndAwaitAdvance();
     // Нулевая фаза завершена - поток может продолжить
выполнение на фазе 1
     // Поток что-то делает какое-то время
      try {
         Thread.sleep((int)Math.random()*1000+500);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
       System.out.println(name + " выполняет фазу " +
phaser.getPhase());
    // Поток прибыл и ожидает завершения первой фазы на
фазере
       phaser.arriveAndAwaitAdvance();
    // Первая фаза завершена - поток может продолжить
выполнение на фазе 2
    // Поток что-то делает какое-то время
    // Если имя потока "PhaseThread-2", то ко второй
фазе переходить не надо.
    // Снимаем регистрацию в фазере и останавливаем
поток
       if (name.equals("PhaseThread-2"))
       {
            phaser.arriveAndDeregister();
```

```
System.out.println("Поток PhaseThread-2 отменил свою
регистрацию в фазере. Число зарегистрированных потоков -
" + phaser.getRegisteredParties());
            Thread.currentThread().stop(); ;
        };
       try {
          Thread.sleep((int)Math.random()*1000+500);
       } catch (InterruptedException e) {
          e.printStackTrace();
       }
        System.out.println(name + " выполняет фазу " +
phaser.getPhase());
//Поток завершил все фазы и отменил свою регистрацию в
фазере (дерегистрация)
        phaser.arriveAndDeregister();
    }}
```

Порядок выполнения работы

1. Изучите теоретический материал.
2. Изучите работу примеров использования синхронизаторов, выполнив их. Выполните дополнительные задания преподавателя.
3. Ответьте на контрольные вопросы.

Контрольные вопросы

1. Для чего используются синхронизаторы?
2. Опишите назначение семафора, конструкторы и методы класса Semaphore.
3. Объсните работу примера использования семафора.
4. Опишите назначение защелки с обратным отсчетом, конструкторы и методы класса CountDownLatch.
5. Объсните работу примера использования защелки с обратным отсчетом.
6. Опишите назначение циклического барьера, конструкторы и методы класса CyclicBarrier.
7. Объясните принцип работы фазера.
8. Опишите конструкторы и методы класса Phaser.
9. Прокомментируйте примеры использования Phaser.

Библиографический список

1. Хорстманн К., Корнелл Г. Java2. Библиотека профессионала, том 1. Основы. – М.: ООО «И.Д. Вильямс», 2008.
2. Эванс Б., Вербург М. Java. Новое поколение разработки. – СПб.: Питер, 2014.
3. Гетц Б., Пайерлс Т., Блох Д., Боубер Д., Холмс Д., Ли Д. Java Concurrency на практике. — СПб.: Питер, 2020.