Министерство Просвещения Республики Молдова
Технический Университет Молдовы

ОТЧЕТ

по лабораторной работе nr. 2
по предмету PR по теме:
“Программирование Multi-Threading”

Выполнил: ст. гр. TI-145 Ялтыченко А.

Проверил: преп. Остапенко С.

Кишинев 2017 г.

Цель работы: Реализация потоков на языке Java
Тема работы: Свойства потоков. Состояния потока исполнения. Запуск, остановка, завершение потока. Группы потоков. Механизмы для реализации коммуникации и синхронизации потоков.
Задание:

- В соответствии с вариантом задания реализовать представленную схему потоков исполнения

Краткая теория

CountDownLatch

CountDownLatch (замок с обратным отсчетом) предоставляет возможность любому количеству потоков в блоке кода ожидать до тех пор, пока не завершится определенное количество операций, выполняющихся в других потоках, перед тем как они будут «отпущены», чтобы продолжить свою деятельность. В конструктор CountDownLatch (CountDownLatch(int count)) обязательно передается количество операций, которое должно быть выполнено, чтобы замок «отпустил» заблокированные потоки.

Блокировка потоков снимается с помощью счётчика: любой действующий поток, при выполнении определенной операции уменьшает значение счётчика. Когда счётчик достигает 0, все ожидающие потоки разблокируются и продолжают выполняться (примером CountDownLatch из жизни может служить сбор экскурсионной группы: пока не наберется определенное количество человек, экскурсия не начнется).

CyclicBarrier

CyclicBarrier реализует шаблон синхронизации «барьер». Циклический барьер является точкой синхронизации, в которой указанное количество параллельных потоков встречается и блокируется. Как только все потоки прибыли, выполняется опционное действие (или не выполняется, если барьер был инициализирован без него), и, после того, как оно выполнено, барьер ломается и ожидающие потоки «освобождаются». В конструктор барьера (CyclicBarrier(int parties) и CyclicBarrier(int parties, Runnable barrierAction)) обязательно передается количество сторон, которые должны «встретиться», и, опционально, действие, которое должно произойти, когда стороны встретились, но перед тем когда они будут «отпущены».

Барьер похож на CountDownLatch, но главное различие между ними в том, что вы не можете заново использовать «замок» после того, как его счётчик достигнет нуля, а барьер вы можете использовать снова, даже после того, как он сломается. CyclicBarrier является альтернативой метода join(), который «собирает» потоки только после того, как они выполнились.

Методы wait()/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.'

Ход работы

Анализ варианта задания

Вариант журнала, соответствующий номеру по журналу представлен ниже на рис. 1:

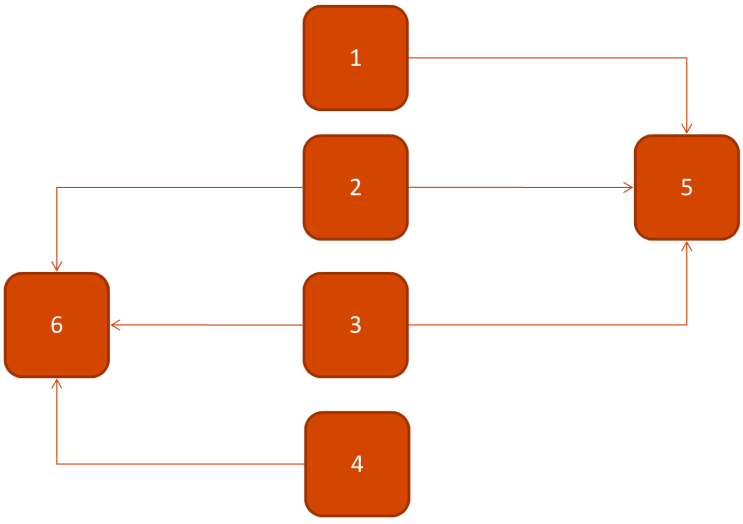

Рис. 1 – Вариант задания

Можно заметить, что поток 5 начинает работать при осуществлении необходимых действий потоками 1,2,3, а поток 6 – потоками 2,3,4. Для реализации подобного рода синхронизации, можно использовать механизмы CyclicBarrier, CountDownLatch, wait/notify.

Реализация синхронизации посредством барьера и механизма wait/notify

Для максимальной наглядности синхронизации все шесть потоков запускаются одновременно, однако, потоки 5 и 6 будут пребывать в ожидании реализации необходимых условий для выполнения своего набора команд (в данном случае вывода на экран сообщения).

Потоки созданы в разных классах, каждый из которых наследует класс Thread. Инициализация и запуск потоков происходят при помощи кода, представленного ниже:

```
public void runLab(){
        final CyclicBarrier barrier = new CyclicBarrier(3,() -> {
                System.out.println("BARRIER!");
                synchronized (thread5Waiter){
                    thread5Waiter.notify();
                }
            }
        );
        CountDownLatch latch = new CountDownLatch(3);
        Thread1 thread1 = new Thread1(barrier);
        Thread2 thread2 = new Thread2(barrier,latch);
        Thread3 thread3 = new Thread3(barrier, latch);
        Thread4 thread4 = new Thread4(latch);
        Thread5 thread5 = new Thread5(thread5Waiter);
        Thread6 thread6 = new Thread6(latch);
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
        thread5.start();
        thread6.start();
    }
```

В конструкторы классов потоков передаются экземпляры классов-шаблонов синхронизации во избежание использования static атрибутов класса Main.
Для CyclicBarrier определено действие, выполняемое после преодоления барьера: отображение сообщения и вызов метода notify для thread5Waiter, где thread5Waiter - экземпляр класса Object, для которого при старте экзепляра класса Thread5 был выполнен метод .wait(). В то же время, в потоках 1,2,3 для barrier вызывается метод barrier.await(). Таким образом, преодоление барьера приводит к выполнению заданного действия и разблокировки 5-го потока.

Реализация синхронизации посредством механизма CountDownLatch

CountDownLatch передается в конструкторы потоков 2,3,4 и 6. Поток 6 при старте обращается к latch с методом await(), в то время, как потоки 2,3,4 по истечению некоторого времени и выполнению своих задач вызывают метод latch.countDown(). После того, как все три потока (именно тройка передана в конструктор CountDownLatch) вызовут countDown(), внутренний счетчик синхронизатора достигнет нуля и поток 6 сможет продолжить свое выполнение.

Результаты работы

Старт потоков, несмотря на последовательность вызовов start() происходит в случайном порядке. Далее в работу включаются механизмы синхронизации и спустя некоторое время происходит преодоление барьера, ведущее к активации 5 потока, а затем и обнуление счетчика CountDownLatch, результатом чего становится активация 6 потока. Результат работы приложения представлен ниже на рисунке 2:

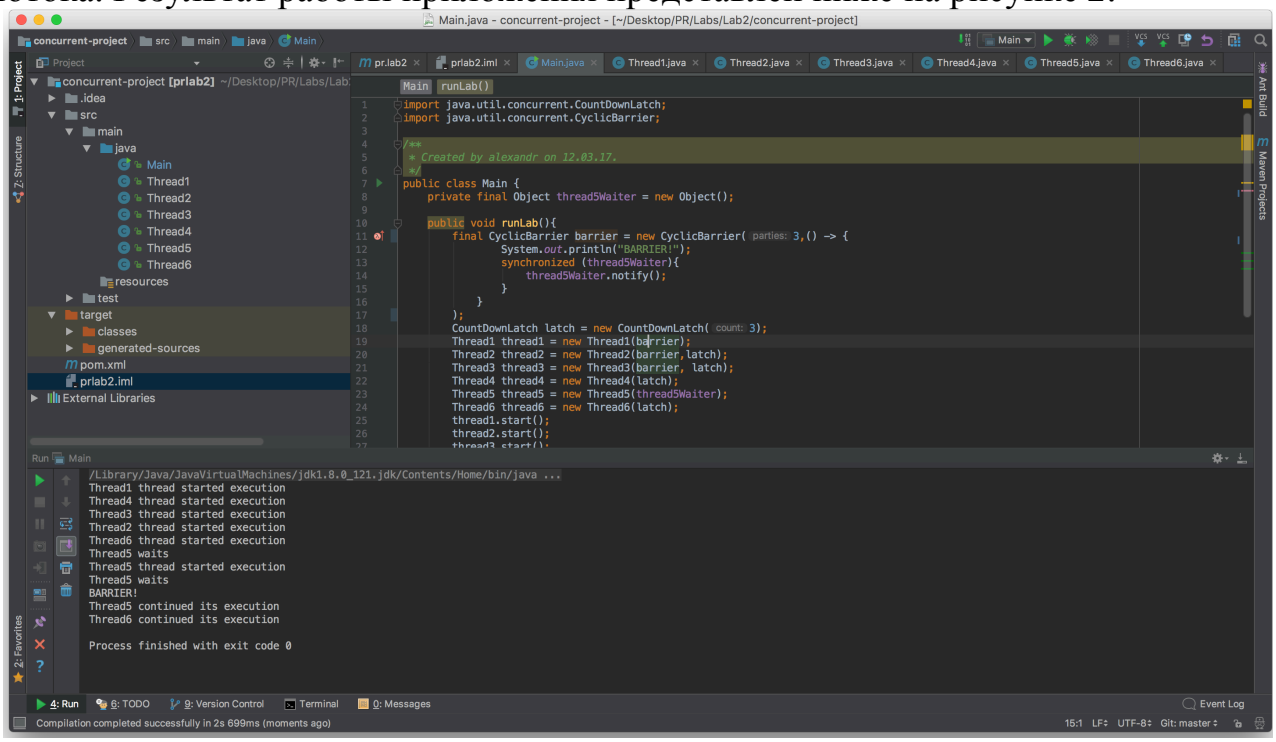

Рис. 2 – Результат работы приложения

Для большей наглядности в методах run() потоков вызывается метод Thread.sleep(). Весь код каждого из классов представлен в приложении А.

Выводы

В рамках данной лабораторной работы были изучены базовые принципы функционирования и использования механизмов создания и запуска потоков, а также их синхронизации (такие как CountDownLatch, CyclicBarrier, wait/notify) в языке Java. На базе полученных знаний было реализовано multi-threading-приложение, осуществляющее пуск и синхронизацию потоков в соответствии со схемой варианта задания. Замечено, что запуск потоков, как и выполнение методов run() каждого из них происходит независимо и параллельно, таким образом, что повторный запуск того же приложения может привести к получению другого вывода в консоль. Именно поэтому были применены механизмы синхронизации, обеспечившие активацию 5 и 6 потоков в соответствии с представленным условием.

ПРИЛОЖЕНИЕ А

Исходный код проекта

```
//Main.java

public class Main {
    private final Object thread5Waiter = new Object();

    public void runLab(){
        final CyclicBarrier barrier = new CyclicBarrier(3,() -> {
                System.out.println("BARRIER!");
                synchronized (thread5Waiter){
                    thread5Waiter.notify();
                }
            }
        );
        CountDownLatch latch = new CountDownLatch(3);
        Thread1 thread1 = new Thread1(barrier);
        Thread2 thread2 = new Thread2(barrier,latch);
        Thread3 thread3 = new Thread3(barrier, latch);
        Thread4 thread4 = new Thread4(latch);
        Thread5 thread5 = new Thread5(thread5Waiter);
        Thread6 thread6 = new Thread6(latch);
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
        thread5.start();
        thread6.start();
    }
    public static void main(String[] args) {
        Main main = new Main();
        main.runLab();
    }

}

//Thread1.java

public class Thread1 extends Thread {
    private final CyclicBarrier barrier;
    public Thread1(CyclicBarrier barrier){
        this.barrier = barrier;
    }
    @Override
    public void run(){
        System.out.println("Thread1 thread started execution ");
        try {
            Thread.sleep(500);
            barrier.await();
        } catch (InterruptedException ex) {
            return;
        } catch (BrokenBarrierException ex) {
            return;
        }
    }
}

//Thread2.java

public class Thread2 extends Thread{
```

```
    private final CyclicBarrier barrier;
    private final CountDownLatch latch;
    public Thread2(CyclicBarrier barrier, CountDownLatch latch){
        this.barrier = barrier;
        this.latch = latch;
    }
    @Override
    public void run(){
        System.out.println("Thread2 thread started execution ");
        try {
            Thread.sleep(300);
            barrier.await();
            latch.countDown();
        } catch (InterruptedException ex) {
            return;
        } catch (BrokenBarrierException ex) {
            return;
        }
    }
}

//Thread3.java

public class Thread3 extends Thread {
    private final CyclicBarrier barrier;
    private final CountDownLatch latch;
    public Thread3(CyclicBarrier barrier, CountDownLatch latch){
        this.barrier = barrier;
        this.latch = latch;
    }
    @Override
    public void run(){
        System.out.println("Thread3 thread started execution ");
        try {
            Thread.sleep(400);
            barrier.await();
            latch.countDown();
        } catch (InterruptedException ex) {
            return;
        } catch (BrokenBarrierException ex) {
            return;
        }
    }
}

//Thread4.java

public class Thread4 extends Thread {
    private final CountDownLatch latch;
    public Thread4(CountDownLatch latch){
        this.latch = latch;
    }
    @Override
    public void run(){
        System.out.println("Thread4 thread started execution ");
        try {
            Thread.sleep(1250);
            latch.countDown();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

//Thread5.java
```

```
public class Thread5 extends Thread{
    private final Object waiter;
    public Thread5(Object waiter){
        this.waiter = waiter;
    }
    @Override
    public void run(){
        System.out.println("Thread5 thread started execution ");
        synchronized (waiter) {
            try {
                System.out.println("Thread5 waits");
                waiter.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Thread5 continued its execution ");
    }
}

//Thread6.java

public class Thread6 extends Thread {
    private final CountDownLatch latch;
    public Thread6(CountDownLatch latch){
        this.latch = latch;
    }
    @Override
    public void run(){
        System.out.println("Thread6 thread started execution ");
        try {
            System.out.println("Thread5 waits");
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread6 continued its execution ");

    }
}
```