Lab 2

Java explicit locking

Besides the already mentioned mechanism using *synchronized* Java also offers an explicit way of locking through the `Lock` interface (from the `java.util.concurrent.locks` package) that is implemented by several classes which can be used for instantiating locks. One of these is `ReentrantLock`.

The ReentrantLock class

The `ReentrantLock` class can be used for instantiating locks that can be used to achieve synchronized access to a portion of code. The main methods used in performing this are *lock* and *unlock*.

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

public class SharedWork {

        ReentrantLock lock;
        SomeObject sharedObject;

        SharedWork() {
                lock = new ReentrantLock();
                sharedObject = new SomeObject();
        }

        public void accessObject() {
                MyThread thread1 = new MyThread();
                MyThread thread2 = new MyThread();
                thread1.start();
                thread2.start();
        }

        class MyThread extends Thread {

         public void run() {
          while (true) {
            lock.lock();                // the thread acquires the lock
   shareObject.someMethod();   // executes a method of the object
// that requires synchronization
  lock.unlock();               // the thread releases the lock
         }
        }
      }
}
```

What if the method in the above example will throw an exception? Unlike using *synchronized* that releases the internal lock taken when the execution of the program exits the synchronized block, the explicit locks remain taken until *unlock* is called. It is usually recommended to ensure that the *unlock* call is executed:

```
...
while (true) {
 lock.lock();                 //the thread acquires the lock
 try
  {
   shareObject.someMethod(); //executes a method of the
                             //object that requires sync
  }
 finally
  {
   lock.unlock();            //the thread releases the lock
  }
}
...
```

The reentrance property

One of the properties of the ReentrantLock is that *lock* can be called successfully more than once by a thread already holding the lock. The effect is an internal increment of the lock instance. In case of calling *unlock* this counter is decremented, and the lock isn't actually released until the counter reaches 0. If you want to use the lock in a non-reentrant way, by not allowing to increment more than one the internal counter, this can be achieved by using *isHeldByCurrentThread* method:

```
 // sample taken from Java API Documentation
 class X {
   ReentrantLock lock = new ReentrantLock();
   // ...
   public void m() {
    assert !lock.isHeldByCurrentThread(); // checking if the thread
                                          // doesn't already hold the lock
    lock.lock();
     try {
           // ... method body
       } finally {
           lock.unlock();
       }
   }
 }
```

The current value of the internal lock counter can be obtained by calling *getHoldCount*

Synchronizers

The purpose of most (if not all) synchronizers is to guard some area of the code (critical section) from concurrent access by threads. They provide a higher level of abstraction then when using locks.

Semaphore

A Semaphore is a thread synchronization construct that can be used either to send signals between threads to avoid missed signals, or to guard a critical section like you would with a lock. A semaphore maintains a count between zero and some maximum value, limiting the number of threads that are simultaneously accessing a shared resource (critical section). Java 8 comes with a semaphore implementation in the `java.util.concurrent` package.

A semaphore is like an integer, with three differences:

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.

2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.

3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

Examples

1. The lock example revisited

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

public class SharedWork {

 Semaphore sem;
 SomeObject sharedObject;

 SharedWork() {
  sem = new Semaphore(1);//upper counter limit; 1 would create a binary
  //semaphore with almost the same functionality as a Lock
  sharedObject = new SomeObject();
 }

 public void accessObject() {
  MyThread thread1 = new MyThread();
  MyThread thread2 = new MyThread();
  thread1.start();
  thread2.start();
}

class MyThread extends Thread {

 public void run() {
  while (true) {
   sem.acquire();
   // 1. the thread acquires the semaphore by decrementing
   //    the counter with one and blocking the rest of the threads to enter
   //    a critical section
   // 2. try/catch omitted for brevity
   shareObject.someMethod();
   //executes a method of the object that requires sync
```

```
   sem.release(); //the thread releases the semaphore
  }
 }
}
}
```

2. Parking places

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

public class Parking {
 // Parking place is taken - true, is free - false
 private static final boolean[] PARKING_PLACES = new boolean[5];
  // We set the fairness flag to true, this way acquire() will be giving
  // permits according to thread arrival order
  private static final Semaphore SEMAPHORE = new Semaphore(5, true);

  public static void main(String[] args) throws InterruptedException {
   for (int i = 1; i <= 7; i++) {
    new Thread(new Car(i)).start();
    Thread.sleep(400);
   }
  }

   public static class Car implements Runnable {
    private int carNumber;

     public Car(int carNumber) {
      this.carNumber = carNumber;
     }

     @Override
     public void run() {
      System.out.printf("Car#%d attempts to park.\n", carNumber);
       try {
        // acquire() requests to run the code that follows;
        // if access was denied, the thread requesting access will be
        // blocked, until the semaphore permits access
        SEMAPHORE.acquire();
        int parkingNumber = -1;
        // trying to find a free spot
         synchronized (PARKING_PLACES){
          for (int i = 0; i < 5; i++)
           if (!PARKING_PLACES[i]) { // when a spot is free
            PARKING_PLACES[i] = true; // we occupy it
            parkingNumber = i; // semaphore guarantees space availability
            System.out.printf("Car#%d parked at spot %d.\n", carNumber, i);
            break;
           }
          }

         Thread.sleep(5000); // Leave the car, go shopping :)

         synchronized (PARKING_PLACES) {
          PARKING_PLACES[parkingNumber] = false;// Freeing up the space
         }

         //release(), frees the resource
         SEMAPHORE.release();
         System.out.printf("Car#%d left the parking spot.\n", carNumber);
```

```
            } catch (InterruptedException e) {}
          }
      }
}
```

A semaphore works in the same way as a lock, but allows ***x*** number of threads to enter. In addition, one of the main differences between a lock and a semaphore is that the lock, once taken, can be released only by the thread which got it. For the semaphore any thread can call `release()`.

Further info:

- An animation explaining how Semaphores work can be found here
- A wonderful introduction to Semaphores can be also found here
- The Little Book of Semaphores can be consulted as well, which covers practical use cases for using Semaphores

CountDownLatch

`CountDownLatch` in Java is a type of synchronizer which allows one `Thread` to wait for one or more `Thread`s before it starts processing. `CountDownLatch` works on latch principle, thread will wait until gate is open. One thread waits for `n` number of threads specified while creating `CountDownLatch`.

e.g. `final CountDownLatch latch = new CountDownLatch(3);`

Here we set the counter to 3.

Any thread (usually the main application thread), which calls `CountDownLatch.await()` will wait until count reaches zero or it's interrupted by another `Thread`. To decrement the count, other threads need to call the `CountDownLatch.countDown()` method. As soon as count reaches zero, the threads which invoked the `await()` method will resume (unblock).

The disadvantage of `CountDownLatch` is that it's not reusable: once the count become zero it is no longer usable.

Examples

1. A simple master-worker example

```
import java.util.concurrent.*;

public class CountDownLatchExample {
```

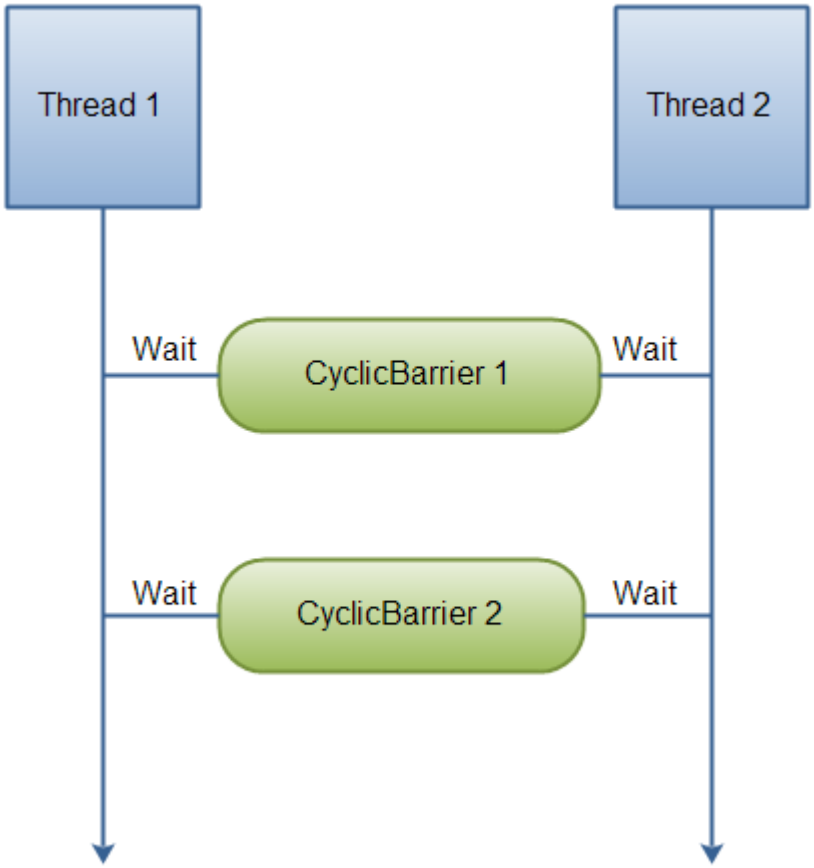
```
 public static class ProcessThread implements Runnable {

  CountDownLatch latch;
  long workDuration;
  String name;

  public ProcessThread(String name, CountDownLatch latch, long duration){
   this.name= name;
   this.latch = latch;
   this.workDuration = duration;
  }

  public void run() {
   try
    {
     System.out.println(name +" processing something for "+
     workDuration/1000 + " seconds");
     Thread.sleep(workDuration);
    }
   catch (InterruptedException e)
    {
     e.printStackTrace();
    }
   System.out.println(name+ " completed all tasks");
   // when task finishes
   // count down the latch count...
   latch.countDown();
  }
 }

 public static void main(String[] args) {
  // Parent thread creating a latch object
  CountDownLatch latch = new CountDownLatch(3);

  // time in millis
  new Thread(new ProcessThread("Worker1",latch, 2000)).start();
  new Thread(new ProcessThread("Worker2",latch, 6000)).start();
  new Thread(new ProcessThread("Worker3",latch, 4000)).start();

  System.out.println("Waiting for children processes to complete....");

  try {
       // current thread will get notified if all chidren's are done
       // and thread will resume from wait() mode.
       latch.await();
      } catch (InterruptedException e) {
       e.printStackTrace();
      }

       System.out.println("All process completed....");
       System.out.println("Parent Thread resuming work....");
    }
}
```

Further info:

- An animation explaining how latches work can be found here

CyclicBarrier

The `java.util.concurrent.CyclicBarrier` class is a synchronization mechanism that can synchronize threads progressing through some algorithm. In other words, it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue. Here is a diagram illustrating that:



Two threads waiting for each other at CyclicBarriers.

The threads wait for each other by calling the `await()` method on the `CyclicBarrier`. Once N threads are waiting at the `CyclicBarrier`, all threads are released and can continue running.

Creating a CyclicBarrier

When you create a `CyclicBarrier` you specify how many threads are to wait at it, before releasing them. Here is how you create a `CyclicBarrier`:

```
CyclicBarrier barrier = new CyclicBarrier(2);
```

Waiting at a CyclicBarrier

Here is how a thread waits at a `CyclicBarrier`:

```
barrier.await();
```

You can also specify a timeout for the waiting thread. When the timeout has passed the thread is also released, even if not all N threads are waiting at the `CyclicBarrier`. Here is how you specify a timeout:

```
barrier.await(10, TimeUnit.SECONDS);
```

The waiting threads waits at the `CyclicBarrier` until either:

- The last thread arrives (calls `await()`)
- The thread is interrupted by another thread (another thread calls its `interrupt()` method)
- Another waiting thread is interrupted
- Another waiting thread times out while waiting at the `CyclicBarrier`
- The `CyclicBarrier.reset()` method is called by some external thread.

CyclicBarrier Action

The `CyclicBarrier` supports a barrier action, which is a `Runnable` that is executed once the last thread arrives. You pass the `Runnable` barrier action to the `CyclicBarrier` in its constructor, like this:

```
Runnable      barrierAction = ... ;
CyclicBarrier barrier       = new CyclicBarrier(2, barrierAction);
```

CyclicBarrier Example

Here is a code example that shows you how to use a `CyclicBarrier`:

```
Runnable barrier1Action = new Runnable() {
    public void run() {
        System.out.println("BarrierAction 1 executed ");
    }
};
Runnable barrier2Action = new Runnable() {
    public void run() {
        System.out.println("BarrierAction 2 executed ");
    }
};

CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);
CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);

CyclicBarrierRunnable barrierRunnable1 =
        new CyclicBarrierRunnable(barrier1, barrier2);

CyclicBarrierRunnable barrierRunnable2 =
        new CyclicBarrierRunnable(barrier1, barrier2);
```

```
new Thread(barrierRunnable1).start();
new Thread(barrierRunnable2).start();
```

Here is the `CyclicBarrierRunnable` class:

```
public class CyclicBarrierRunnable implements Runnable{

    CyclicBarrier barrier1 = null;
    CyclicBarrier barrier2 = null;

    public CyclicBarrierRunnable(
            CyclicBarrier barrier1,
            CyclicBarrier barrier2) {

        this.barrier1 = barrier1;
        this.barrier2 = barrier2;
    }

    public void run() {
        try {
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName() +
                               " waiting at barrier 1");
            this.barrier1.await();

            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName() +
                               " waiting at barrier 2");
            this.barrier2.await();

            System.out.println(Thread.currentThread().getName() +
                               " done!");

        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

Here is the console output for an execution of the above code. Note that the sequence in which the threads gets to write to the console may vary from execution to execution. Sometimes `Thread-0` prints first, sometimes `Thread-1` prints first etc.

```
Thread-0 waiting at barrier 1
Thread-1 waiting at barrier 1
BarrierAction 1 executed
Thread-1 waiting at barrier 2
Thread-0 waiting at barrier 2
BarrierAction 2 executed
Thread-0 done!
Thread-1 done!
```

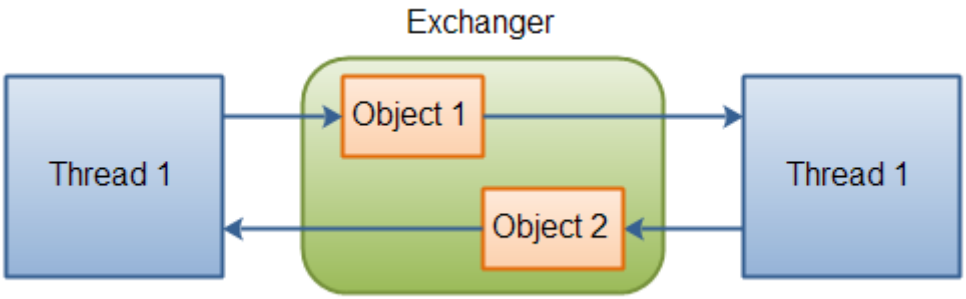
Further info:

- An animation explaining how the `CyclicBarrier` works can be found here

Exchanger<V>

Another form of barrier is the `java.util.concurrent.Exchanger`, a two-party barrier in which the parties exchange data at the barrier point. `Exchanger`s are useful when the parties perform asymmetric activities, for example when one thread fills a buffer with data and the other thread consumes the data from the buffer; these threads could use an `Exchanger` to meet and exchange a full buffer for an empty one. When two threads exchange objects via an `Exchanger`, the exchange constitutes a safe publication of both objects to the other party.

The class represents a kind of rendezvous point where two threads can exchange objects. Here is an illustration of this mechanism:



Two threads exchanging objects via an Exchanger.

Exchanging objects is done via one of the two `exchange()` methods. Here is an example:

```
Exchanger exchanger = new Exchanger();

ExchangerRunnable exchangerRunnable1 =
        new ExchangerRunnable(exchanger, "A");

ExchangerRunnable exchangerRunnable2 =
        new ExchangerRunnable(exchanger, "B");

new Thread(exchangerRunnable1).start();
new Thread(exchangerRunnable2).start();
```

Here is the `ExchangerRunnable` code:

```
public class ExchangerRunnable implements Runnable{

    Exchanger exchanger = null;
    Object    object    = null;

    public ExchangerRunnable(Exchanger exchanger, Object object) {
        this.exchanger = exchanger;
        this.object = object;
    }

    public void run() {
        try {
```

```
            Object previous = this.object;

            this.object = this.exchanger.exchange(this.object);

            System.out.println(
                    Thread.currentThread().getName() +
                    " exchanged " + previous + " for " + this.object
            );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

This example prints out this:

```
Thread-0 exchanged A for B
Thread-1 exchanged B for A
```

Further info:

- An animation explaining how the `Exchanger` works can be found here

Phaser

To understand what the Phaser does and why its useful its important to know how it compares to the aforementioned constructs.

Here are attributes of a `CountdownLatch` and `CyclicBarrier`

Note:

- Number of parties is another way of saying # of different threads
- Not Reusable means you will have to create a new instance of the barrier before reusing
- A barrier is advanceable if a thread can arrive and continue doing work without waiting for others or can wait for all threads to complete

`CountdownLatch`

- Fixed number of parties
- Not resuable
- Advanceable (look at `latch.countDown();` **advanceable** `latch.await();` **must wait**)

`CyclicBarrier`

- Fixed number of parties
- Reusable
- Not advanceable

So the CountdownLatch is not reusable, you must create a new instance each time, but is avanceable. CylicBarrier can be re used but all threads must wait for each party to arrive at the barrier.

`Phaser`

- Dynamic number of parties
- Reusable
- Advanceable

When a thread wants to be known to the Phaser they invoke `phaser.register()` when the thread arrives at the barrier they invoke `phaser.arrive()` **and here is where it is advanceable**. If the thread wants to await for all registered tasks to complete `phaser.arriveAndAwaitAdvance()`

There is also the concept of a phase in which threads can wait on a completion of a other operations that may have not completed. Once all threads arrive at the phaser's barrier a new phase is created (an increment of one).

Further info:

- An animation explaining how the `Phaser` works can be found here

Credits:

http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/

https://habrahabr.ru/post/277669/

http://tutorials.jenkov.com/java-concurrency/anatomy-of-a-synchronizer.html

http://tutorials.jenkov.com/java-concurrency/semaphores.html

http://www.greenteapress.com/semaphores/downey08semaphores.pdf

https://stackoverflow.com/questions/2332765/lock-mutex-semaphore-whats-the-difference

http://www.codeproject.com/Articles/14746/Multithreading-Tutorial

https://stackoverflow.com/questions/17827022/what-is-countdown-latch-in-java-multithreading

http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html

https://stackoverflow.com/questions/6830904/java-tutorials-explanations-of-jsr166y-phaser