Lab 9

Locks: Read/Write lock

A read / write lock is a more sophisticated lock than the classical Lock implementations in Java. Imagine you have an application that reads and writes some resource, but writing it is not done as much as reading it is. Two threads reading the same resource does not cause problems for each other, so multiple threads that want to read the resource are granted access at the same time, overlapping. But, if a single thread wants to write to the resource, no other reads nor writes must be in progress at the same time. To solve this problem of allowing multiple readers but only one writer, you will need a read / write lock.

Example use case:

```
public void increment(long amount) {
        try {
         rwlock.writeLock().lock();
         counter+=amount;
        } finally {
         rwlock.writeLock().unlock();
        }
}

public long getCounter() {
        try {
         rwlock.readLock().lock();
         return counter;
        } finally {
         rwlock.readLock().unlock();
        }
}
```

Reentrant locks can work in fair and non-fair mode. Fair mode means that requests to the lock object are accepted by the order they have been received. This mode prevents starvation and yields predictable latency, although it works much slower.

Java 5 comes with read / write lock implementations in the java.util.concurrent package.

```
ReentrantReadWriteLock lock = new ReentrantReadWriteLock (true);
```

Even so, it may still be useful to know the theory behind their implementation.

```
public class ReadWriteLock{

  private int readers       = 0;
  private int writers       = 0;

  public synchronized void lockRead() throws InterruptedException{
```

```
    while(writers > 0){
      wait();
    }
    readers++;
  }

  public synchronized void unlockRead(){
    readers--;
    notifyAll();
  }

  public synchronized void lockWrite() throws InterruptedException{
    while(readers > 0 || writers > 0){
      wait();
    }
    writers++;
  }

  public synchronized void unlockWrite() throws InterruptedException{
    writers--;
    notifyAll();
  }
}
```

Locks: StampedLock

A new StampedLock class adds a capability-based lock with three modes for controlling read/write access (writing, reading, and optimistic reading). This class also supports methods that conditionally provide conversions across the three modes. The StampedLock can work the same as read write lock, although it assumes contention by default.

The ReentrantReadWriteLock has a lot of shortcomings: It suffers from starvation. You could not upgrade a read lock into a write lock. There was no support for optimistic reads. The new Java 8 StampedLock addresses all these shortcomings. With some clever code idioms we can also get better performance. Instead of the usual locking, it returns a long number whenever a lock is granted. This stamp number is then used to unlock again. For example, here is how the code above would look:

```
StampedLock sl = new StampedLock();

...

long stamp = sl.writeLock();
try {
  // do something that needs exclusive access
} finally {
  sl.unlockWrite(stamp);
}
```

This still begs the question - why do we need this in the first place? Let's take a slightly larger example and write it with different approaches.

BankAccount Without Any Synchronization

Our first BankAccount has no synchronization at all. It also does not check that the amounts being deposited and withdrawn are positive. We will leave out argument checking in our code.

```
public class BankAccount {
  private long balance;

  public BankAccount(long balance) {
    this.balance = balance;
  }

  public void deposit(long amount) {
    balance += amount;
  }

  public void withdraw(long amount) {
    balance -= amount;
  }

  public long getBalance() {
    return balance;
  }
}
```

Synchronized BankAccount

The second version uses the synchronized keyword to protect the methods from being called simultaneously by multiple threads on the same BankAccount object. We could either synchronize on "this" or on a private field. For our example, this would not make a difference. In this approach a thread would not be able to read the balance whilst another thread is depositing money.

```
public class BankAccountSynchronized {
  private long balance;

  public BankAccountSynchronized(long balance) {
    this.balance = balance;
  }

  public synchronized void deposit(long amount) {
    balance += amount;
  }

  public synchronized void withdraw(long amount) {
    balance -= amount;
  }

  public synchronized long getBalance() {
    return balance;
  }
}
```

ReentrantReadWriteLock BankAccount

The next version uses the ReentrantReadWriteLock, which differentiates between exclusive and non-exclusive locks. In both cases, the locks are pessimistic. This means that if a thread is currently holding the write lock, then any reader thread will get suspended until the write lock is released again.

However, the overhead of using a ReentrantReadWriteLock is substantial. As a ballpark figure, we need the read lock to execute for about 2000 clock cycles in order to win back the cost of using it. In our case the getBalance() method does substantially less, so we would probably be better off just using a normal ReentrantLock.

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

public class BankAccountReentrantReadWriteLock {
  private final ReadWriteLock lock = new ReentrantReadWriteLock();
  private long balance;

  public BankAccountReentrantReadWriteLock(long balance) {
    this.balance = balance;
  }

  public void deposit(long amount) {
    lock.writeLock().lock();
    try {
      balance += amount;
    } finally {
      lock.writeLock().unlock();
    }
  }

  public void withdraw(long amount) {
    lock.writeLock().lock();
    try {
      balance -= amount;
    } finally {
      lock.writeLock().unlock();
    }
  }

  public long getBalance() {
    lock.readLock().lock();
    try {
      return balance;
    } finally {
      lock.readLock().unlock();
    }
  }
}
```

StampedLock BankAccount

Our next version uses StampedLock. we have written two getBalance() methods. The first uses pessimistic read locks, the other optimistic. In our case, since there are no invariants on the fields that would somehow restrict the values, we never need to have a pessimistic lock. Thus the optimistic read is only to ensure memory visibility.

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

public class BankAccountStampedLock {
  private final StampedLock sl = new StampedLock();
  private long balance;

  public BankAccountStampedLock(long balance) {
    this.balance = balance;
  }
```

```
  public void deposit(long amount) {
    long stamp = sl.writeLock();
    try {
      balance += amount;
    } finally {
      sl.unlockWrite(stamp);
    }
  }

  public void withdraw(long amount) {
    long stamp = sl.writeLock();
    try {
      balance -= amount;
    } finally {
      sl.unlockWrite(stamp);
    }
  }

  public long getBalance() {
    long stamp = sl.readLock();
    try {
      return balance;
    } finally {
      sl.unlockRead(stamp);
    }
  }

  public long getBalanceOptimisticRead() {
    long stamp = sl.tryOptimisticRead();

long balance = this.balance;
    if (!sl.validate(stamp)) {
      stamp = sl.readLock();
      try {
        balance = this.balance;
      } finally {
        sl.unlockRead(stamp);
      }
    }
    return balance;
  }
}
```

In our getBalanceOptimisticRead(), we could retry several times. However, as we stated before, if memory visibility is all we care about, then StampedLock is overkill. StampedLocks are useful if we had multiple fields and/or an invariant over the field. Let's take for example a point on a plane, consisting of an "x" and a "y". If we move in a diagonal line, we want to update the x and y in an atomic operation. Thus a moveBy(10,10) should never expose a state where x has moved by 10, but y is still at the old point. The fully synchronized approach would work, as would the ReentrantLock and ReentrantReadWriteLock. However, all of these are pessimistic. How can we read state in an optimistic approach, expecting to see the correct values?

Let's start by defining a simple Point class that is synchronized and has three methods for reading and manipulating the state:

```
public class Point {
```

```
  private int x, y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public synchronized void move(int deltaX, int deltaY) {
    x += deltaX;
    y += deltaY;
  }

  public synchronized double distanceFromOrigin() {
    return Math.hypot(x, y);
  }

  public synchronized boolean moveIfAt(int oldX, int oldY,
                                       int newX, int newY) {
    if (x == oldX && y == oldY) {
      x = newX;
      y = newY;
      return true;
    } else {
      return false;
    }
  }
}
```

If we use a StampedLock, our move() method look similar to the BankAccountStampedLock.deposit():

```
public void move(int deltaX, int deltaY) {
  long stamp = sl.writeLock();
  try {
    x += deltaX;
    y += deltaY;
  } finally {
    sl.unlockWrite(stamp);
  }
}
```

However, the distanceFromOrigin() method could be rewritten to use an optimistic read, for example:

```
public double distanceFromOrigin() {
  long stamp = sl.tryOptimisticRead();
  int localX = x, localY = y;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      localX = x;
      localY = y;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return Math.hypot(localX, localY);
}
```

In our optimistic distanceFromOrigin(), we first try to get an optimistic read stamp. This might come back as zero if a writeLock stamp has been issued and has not been unlocked yet. However, we assume that it is non-zero and continue reading the fields into local variables localX and localY. After we have read x and y, we validate the stamp. Two things could make this fail: 1. The sl.tryOptimisticRead() method might have come back as zero initially or 2. After we obtained our optimistic lock, another thread requested a writeLock(). We don't know whether this means our copies are invalid, but we need to be safe, rather than sorry. In this version we only try this once and if we do not succeed we immediately move over to the pessimistic read version. Depending on the system, we could get significant performance gains by spinning for a while, hoping to do a successful optimistic read. In our experiments, we also found that a shorter code path between tryOptimisticRead() and validate() leads to a higher chance of success in the optimistic read case.

Here is another idiom that retries a number of times before defaulting to the pessimistic read version. It uses the trick in Java where we break out to a label, thus jumping out of a code block. We could have also solved this with a local boolean variable, but to me this is a bit clearer:

```
private static final int RETRIES = 5;

public double distanceFromOrigin() {
  int localX, localY;
  out:
  {
    // try a few times to do an optimistic read
    for (int i = 0; i < RETRIES; i++) {
      long stamp = sl.tryOptimisticRead();
      localX = x;
      localY = y;
      if (sl.validate(stamp)) {
        break out;
      }
    }
    // pessimistic read
    long stamp = sl.readLock();
    try {
      localX = x;
      localY = y;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return Math.hypot(localX, localY);
}
```

Conditional Write

The last idiom to demonstrate is the conditional write. We first read the current state. If it is what we expect, then we try to upgrade the read lock to a write lock. If we succeed, then we update the state and exit, otherwise we unlock the read lock and then ask for a write lock. The code is a bit harder to understand, so look it over carefully:

```
public boolean moveIfAt(int oldX, int oldY,
                        int newX, int newY) {
  long stamp = sl.readLock();
  try {
    while (x == oldX && y == oldY) {
```

```
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0) {
        stamp = writeStamp;
        x = newX;
        y = newY;
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } finally {
    sl.unlock(stamp); // could be read or write lock
  }
}
```

Credits:

http://tutorials.jenkov.com/java-concurrency/read-write-locks.html
http://www.slideshare.net/haimyadid/java-8-stamped-lock
http://www.javaspecialists.eu/archive/Issue215.html