# CS601: Principles of Software Development

## Custom Locks. Reentrant Lock.

Olga A. Karpenko

Parts of this presentations are based on the materials of Prof. Engle.

# Announcements

- Lab 3 Part 1 is out
  - Due on Monday
- Quiz on Wednesday

# Synchronization in Java

- Volatile variables
- Synchronized code blocks or methods
- Custom lock objects

# Synchronized vs Lock

```
public class Counter {

 private int count  = 0;

 public void increment()
 {
    synchronized(this) {
       count++;
    }
 }
}
```

```
public class Counter {

private int count = 0;
private Lock lock = new
Lock();

public void increment() {
   try {
     lock.lock();
     count++;
   }
   finally {
     lock.unlock();
   }
 }
}
```

http://tutorials.jenkov.com/

# Custom Lock Class: "MultiReadLock"

- May read to shared data structure if…
  - No other threads are writing to it
- May write to shared data structure if…
  - No other threads are reading from it
  - No other threads are writing to it
- Must track…
  - Number of active readers and writers

# MultiReadLock

```
public synchronized void lockRead() {
  while (writers > 0) {
      try {
            this.wait();
      }
      catch (InterruptedException ex) {
          // log the exception
      }
  }
  readers++;
}
```

# MultiReadLock

- `public synchronized void lockRead()`
  - Wait until no active writers
  - Use a loop to avoid spurious wakeups
  - Increase number of readers and “give” lock

- `public synchronized void unlockRead()`
  - Decrease number of readers to “free” the lock
  - Wake up threads, if necessary, using notifyAll()

# MultiReadLock

- `public synchronized void lockWrite()`
  - Wait until no writers and no readers
  - Use a loop to avoid spurious wakeups
  - Increase number of writers and "give" lock

- `public synchronized void unlockWrite()`
  - Decrease number of writers to "free" the lock
  - Wake up threads if necessary using notifyAll()

# Using the Lock

```
Class SynchronizedMap {
    private final Map<String, Data> m = new TreeMap<>();
    private final MultiReadLock lock = new MultiReadLock();

    public Data get(String key) {
        lock.lockRead ();
        try { return m.get(key); }
        finally { lock.unlockRead(); }
    }

    public Data put(String key, Data value) {
        lock.lockWrite();
        try { return m.put(key, value); }
        finally { lock.unlockWrite(); }
    }
}
```

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html

# Performance of MultiReadLock

- Whether it will improve the performance depends on:
  - The frequency of reads compared to writes
  - The duration of reads and writes
  - The # of threads that will compete for for the lock

http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html

# Fairness Policy

- If there are writers and readers waiting to get the lock, who gets priority?
  - Often, writers ("writes" expected to be short and infrequent)
  - "Fair" order - based on arrival order
- Unfair lock might postpone some requests indefinitely
  - The simple lock we discussed does not have a fairness policy

http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html

# Implementation Decisions

- Will the lock be *Reentrant*? Can the thread holding the lock re-acquire it?
  - Can a reader obtain multiple read locks?
  - Can a writer hold multiple write locks?
- Can a writer obtain a read lock while holding the write lock?

http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReadWriteLock.html

# Lab 3 Part 1

- Write `ReentrantReadWriteLock` that allows
  - concurrent read operations,
  - a reader to reacquire a read lock
  - non-concurrent write or read/write operations
  - a writer to reacquire a write lock
  - allows the current writer to get the read lock
- `ThreadSafeHotelData` class
  - Make thread-safe using `ReentrantReadWriteLock`
- `HotelDataBuilder` class that processes reviews concurrently

# Reentrant Read Write Lock

- Need to keep track of:
  - the number of held read locks for each thread
  - the number of held write locks for each thread
- Consider storing this info in two maps
  - one of readers, one for writers
  - thread id is the key,
  - number of locks is the value

# Reentrant Read Write Lock

- isReadLockHeldByCurrentThread()
  - Use `Thread.currentThread().getId()`
- tryAcquiringReadLock()
  - Checks  conditions for acquiring the lock
  - If true, acquires the read lock (updates readersMap)
  - If false, returns false
- lockRead()
  - Calls tryAcquiringReadLock() in a while loop,
  - if it returns false, ***waits***
- unlockRead()
  - updates readersMap, calls notifyAll when needed

# Reentrant Read Write Lock

- isWriteLockHeldByCurrentThread()
  - Use `Thread.currentThread().getId()`
- tryAcquiringWriteLock()
  - Checks  conditions for acquiring the lock
  - If true, acquires the write lock (updates writersMap)
  - If false, returns false
- lockWrite()
  - Calls tryAcquiringWriteLock() in a while loop,
  - if it returns false, ***waits***
- unlockWrite()
  - updates writersMap, calls notifyAll when needed

# Built-in Lock Objects

- See java.util.concurrent.locks
  - May **not** use in this class
  - Might be useful for debugging
- Our Lab 3 lock is closest to: ReentrantReadWriteLock
  - Their version provides a fairness policy
  - Ours is prone to starvation