CMSC 433
Programming Language Technologies and Paradigms
Spring 2013

Wait / Notify / NotifyAll
Optimistic Retries
Composition Follow-up (the risk I mentioned)

ReentrantLock, Wait, Notify, NotifyAll

Some new tools for us…

`java.util.concurrent.locks.ReentrantLock` – this can be used to acquire and release a mutual-exclusion lock (`lock()` and `unlock()`) other than simply around a self-contained block of code. You can also use `tryLock()` to get the lock if it’s available without blocking if it’s not.

`.wait()` – releases a lock that the thread holds on that object only and has the thread basically go to sleep and get added to a “wait set” for that object until it’s told that the lock is once again ready to be acquired – when it is awoken, it is removed from the wait set and tries to reacquire the lock so that it can resume (this can be used from both intrinsic locks and `ReentrantLock` objects)

`.notify()` – chooses one thread that is waiting for the lock (which you must hold when you call this method and should be about to release) and wakes it up to allow it to try to acquire the lock (note that another request could sneak in and steal it or it might not be free yet if you don’t release it soon after notifying)

`.notifyAll()` – informs all threads waiting for the lock (which you must hold when you call this method) to wake up and try to acquire it – this should be used if more than one thread might be waiting to avoid deadlock risks
Recall our robotic Japanese friends who could deadlock?

```java
static class JapaneseRobotBusinessFriends {
    String name;

    public JapaneseRobotBusinessFriends(String nameIn) {
        name = nameIn;
    }

    public synchronized void bow(JapaneseRobotBusinessFriends myBusinessFriend) {
        System.out.println(name + " has bowed to " + myBusinessFriend.name);
        myBusinessFriend.bowback(this);
    }

    public synchronized void bowback(JapaneseRobotBusinessFriends bower) {
        System.out.println(name + " has returned the bow to " + bower.name);
    }
}
```

What if we add a little more foresight to them?

Have a `volatile` Boolean flag that you set when you are busy in some way.

You are busy if:

(a) You are in the process of bowing.
(b) You are in the process of triggering the other robot to bow back at you.

However, before actually starting to trigger the other robot to bow back at you, check to see if it is busy. If it is busy then set yourself as not busy and wait for the other robot to become available.

When you wait, you give up your lock and go into standby mode until you are notified that the lock is available to grab again, or until a certain amount of time has passed.
Fixed robotic Japanese friends who don’t deadlock!

```java
static class JapaneseRobotBusinessFriends {
  String name;
  volatile boolean busy = false;
  
  public JapaneseRobotBusinessFriends(String nameIn) { name = nameIn; }

  public boolean isBusy() { return busy; }

  public synchronized void bow(JapaneseRobotBusinessFriends myBussinessFriend)
      throws InterruptedException {
    busy = true;
    System.out.println(name + " has bowed to " + myBussinessFriend.name);
    while (myBussinessFriend.isBusy()) {
      busy = false;
      wait(1);
      busy = true;
    }
    myBussinessFriend.bowback(this);
    busy = false;
  }

  public synchronized void bowback(JapaneseRobotBusinessFriends bower)
      throws InterruptedException {
    busy = true;
    System.out.println(name + " has returned the bow to " + bower.name);
    busy = false;
    notifyAll();
  }
}
```

Can still deadlock with wait/notify use...

Consider a bounded buffer:

```java
public synchronized void put (Object elt) throws InterruptedException {
  while (elements.size() == maxSize) wait();
  elements.add(elt);
  notify();
}

does not cause deadlock:

public synchronized void take () throws InterruptedException {
  while (elements.size() == 0) wait();
  Object elt = elements.get(0);
  elements.remove(0);
  notifyAll();
  return elt;
}
```

Rance Cleaveland ©2012 University of Maryland
Deadlock Scenario

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>elements.size()</th>
<th>Wait-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>take (w)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>T1</td>
</tr>
<tr>
<td>1</td>
<td>take (w)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>T1, T2</td>
</tr>
<tr>
<td>2</td>
<td>put</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>T2</td>
</tr>
<tr>
<td>3</td>
<td>put (w)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>T2, T3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>put (w)</td>
<td>T2, T3, T4</td>
</tr>
<tr>
<td>5</td>
<td>take (0)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>T3, T4</td>
</tr>
<tr>
<td>6</td>
<td>take (w)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>T1, T3, T4</td>
</tr>
<tr>
<td>7</td>
<td>take (w)</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>T1, T3, T4, T2</td>
</tr>
</tbody>
</table>

Legend
op(w) – operation waits / “rewaits”
op(i) – operation begun at time i completes
op – operation begins and completes without waiting

When classes use other objects...

In addition to making sure two threads aren’t trying to interact with the same thread-unsafe object at the same time, or that invariants that cross objects are being properly guarded, there are some corner cases to consider, even with a thread-safe object.

Some examples:

- You can’t remove something from an empty object (like a bank account with no money, a stack or queue with no items, an empty food carousel, etc.).
- You can’t add something to a bounded collection that has already reached its capacity (like a buffer).
- You can’t use a resource that isn’t currently “ready” (printer with no paper, network if connection is down, file that’s no open for reading yet).
Handing what might be a temporary fault

In a typical single-threaded program, if you reach a fault condition, the method that encounters it will typically end up doing one of the following:

• Ignore it and move on.
• Throw a language-supported exception so that another part of your program can handle it.
• Return an error code rather than a success code.

```java
public class BNC { // BoundedNormalCounter
    private final int minValue;
    private final int maxValue;
    private int currentValue;
    private BNC(min, int max, int initial) {
        minValue = min;   maxValue = Math.max(min, max);
        currentValue = Math.max(initial, min);
    }
    public static BNC getInstance(int min, int max, int init) {
        return new BoundedNormalCounter(min, max, init);
    }
    public int get() { return currentValue; }
    public boolean set(int newVal) {
        if ((newVal<minValue)||(newVal>maxValue)) return false;
        currentValue = newVal;   return true;
    }
    public boolean increment() {
        if (currentValue==maxValue) return false;
        currentValue++;   return true;
    }
    public boolean decrement() {
        if (currentValue==minValue) return false;
        currentValue--;   return true;
    }
}
```
private static void helperOne(BoundedNormalCounter countIn) {
    // Ignore it and move on.
    for (int i=0; i<ADD_THIS_MANY; i++) {
        countIn.increment();
    }
}

private static void helperTwo(BoundedNormalCounter countIn)
    throws LimitExceededExpectation {
    // Throw an exception.
    for (int i=0; i<ADD_THIS_MANY; i++) {
        if (!countIn.increment()) throw new LimitExceededExpectation("Tried to go to high.");
    }
}

private static boolean helperThree(BoundedNormalCounter countIn) {
    // Return a failure code.
    for (int i=0; i<ADD_THIS_MANY; i++) {
        if (!countIn.increment()) return false;
    }
    return true;
}

Handing what might be a temporary fault

In a typical single-threaded program, if you reach a fault condition, the method that encounters it will typically end up doing one of the following:

• Ignore it and move on.
• Throw a language-supported exception so that another part of your program can handle it.
• Return an error code rather than a success code.

What about multi-threaded programs where the issue might be resolved by another thread?

• Hopeful retries – sleep/retry spin cycle.
private static int aggressiveAdd(BoundedNormalCounter countIn) {
    int succeeded = 0;
    for (int i=0; i<ADD_THIS_MANY; i++) {
        counterLock.lock();
        while (!countIn.increment()) {
            try { //if we omit this sleep then our busy-wait would become a livelock
                Thread.sleep(1);
                counterLock.unlock();
            } catch (InterruptedException e) { e.printStackTrace(); }
            counterLock.lock();
        }
        succeeded++;
        counterLock.unlock();
    }
    return succeeded;
}

private static int aggressiveSubtract(BoundedNormalCounter countIn) {
    int succeeded = 0;
    for (int i=0; i<SUBTRACT_THIS_MANY; i++) {
        counterLock.lock();
        while (!countIn.decrement()) {
            try { //if we omit this sleep then our busy-wait would become a livelock
                Thread.sleep(1);
                counterLock.unlock();
            } catch (InterruptedException e) { e.printStackTrace(); }
            counterLock.lock();
        }
        succeeded++;
        counterLock.unlock();
    }
    return succeeded;
}

Manual locking and unlocking

In general, since locking mechanism where you manually lock and unlock won’t release the lock automatically, you’ll want to use a model such as:

    yourManualLock.lock();
    try {
        //Do the protected work.
    } catch (Exception e) {
        //Deal with cleanup.
    } finally {
        yourManualLock.unlock();
    }
Handing what might be a temporary fault

In a typical single-threaded program, if you reach a fault condition, the method that encounters it will typically end up doing one of the following:

• Ignore it and move on.
• Throw a language-supported exception so that another part of your program can handle it.
• Return an error code rather than a success code.

What about multi-threaded programs where the issue might be resolved by another thread?

• Hopeful retries – sleep/retry spin cycle.
• Guarded suspension – wait/reacquire (hopefully minimal spin).

```java
private static int aggressiveAdd(BoundedNormalCounter countIn) {
    int succeeded = 0;
    for (int i=0; i<ADD_THIS_MANY; i++) {
        synchronized(countIn) {
            while (!countIn.increment()) {
                try {
                    countIn.wait();
                } catch (InterruptedException e) { e.printStackTrace(); }
            }
            succeeded++;
            countIn.notify();
        }
    }
    return succeeded;
}

private static int aggressiveSubtract(BoundedNormalCounter countIn) {
    int succeeded = 0;
    for (int i=0; i<SUBTRACT_THIS_MANY; i++) {
        synchronized(countIn) {
            while (!countIn.decrement()) {
                try {
                    countIn.wait();
                } catch (InterruptedException e) { e.printStackTrace(); }
            }
            succeeded++;
            countIn.notify();
        }
    }
    return succeeded;
}
```
Mix the ideas…

We could replace
```java
counterLock.unlock();
Thread.sleep(1);
counterLock.lock();
```
with
```java
counterLock.wait(1);
```
to be able to benefit from quicker releases from other threads and also take advantage of any Java-level efficiencies, but we’d need to use an intrinsic lock (so we could just make the `counterLock` a reference to an `Object`).

Handing what might be a temporary fault

In a typical single-threaded program, if you reach a fault condition, the method that encounters it will typically end up doing one of the following:

- Ignore it and move on.
- Throw a language-supported exception so that another part of your program can handle it.
- Return an error code rather than a success code.

What about multi-threaded programs where the issue might be resolved by another thread?

- Hopeful retries.
- Guarded suspension – wait/reacquire (hopefully minimal spin).
- Optimistic retries.
**Optimistic Retries**

As we’ve seen, the process of locking and unlocking can be an “expensive” thing.

If we generally expect there to not be contention on a resource, and if that resource is “small” in some way, it might make sense to do the following when you are ready to perform your operation that will alter the shared object, try the following until you succeed:

- Make a copy of the object. (this could be costly)
- Perform the operation on the copy.
- Call a *synchronized* commit method (sending it your old a new versions of the shared item) that will then check to see whether the object you altered is still the current object, and commit the change if so / offer to yield if not.

For our example we will create a new class that will handle thread-safety itself called `BoundedSafeOptimisticCounter` as well as a less aggressive “driver” program.

---

**BoundedSafeOptimisticCounter (I)**

The `get()` method and a new `commit(old, new)` method will be synchronized.

```java
public synchronized int get() {
    return currentValue;
}

public synchronized boolean commit(int oldStateValue, int newStateValue) {
    if (currentValue != oldStateValue) {
        return false;
    }
    currentValue = newStateValue;
    return true;
}
```
**BoundedSafeOptimisticCounter (II)**

The `set(val)` method and `increment()` and `decrement()` methods will **not** be synchronized.

```java
public boolean set(int newVal) {
    if ((newVal<minValue) || (newVal>maxValue) ) return false;
    while (true) {
        int localStateCopy = currentValue;
        if (commit(localStateCopy, newVal)) return true;
        Thread.yield();
    }
}

public boolean increment() {
    while (true) {
        int localStateCopy = currentValue;
        if (localStateCopy==maxValue) return false;
        if (commit(localStateCopy, localStateCopy+1)) return true;
        Thread.yield();
    }
}
```

**Less aggressive workers...**

```java
private static int lessAggressiveAdd(BoundedSafeOptimisticCounter countIn) {
    int succeeded = 0;
    for (int i=0; i<ADD_THIS_MANY; i++) {
        while (!countIn.increment()) {}  
        succeeded++;

        try{Thread.sleep(1);} catch(InterruptedException e) {}  
    }
    return succeeded;
}
```
Nested Monitors Issues

If we use our “instance confinement” trick on a thread-safe object that uses wait/notify techniques, we could have a problem! Imagine a bounded safe counter that was written to use the wait/notifyAll model internally.

```java
public class BSC_NotAtNoon { // We don't allow increments at noon!
    private final BSC confinedCounter;

    private synchronized BSC_NotAtNoon(int min, int max, int init) {
        confinedCounter = BSC.getInstance(min, max, init);
    }

    public synchronized int get() { return confinedCounter.get(); }
    public synchronized boolean set(int newVal) {
        return confinedCounter.set(newVal);
    }
    public synchronized boolean increment() {
        if (!currentlyNoon()) { return confinedCounter.increment(); }
        else return false;
    }
    public synchronized boolean decrement() {
        return confinedCounter.decrement();
    }
}
```

What happens if a `BSC_NotAtNoon` object’s increment is called at 1:00pm but the counter is at its limit?

Nested Monitors Lockout!

When the `BSC_NotAtNoon` object’s increment is called at 1:00pm and the counter is at its limit:
- the thread will wait and releases the lock on the inner `BSC` object
- however, even though it is in a wait set for the `BSC` object, it does not release its lock on the `BSC_NotAtNoon` object
- no other thread can have this `BSC_NotAtNoon` object’s decrement called, so this thread will be waiting forever!

Could try to “solve” this by not having the outer class synchronize, but then we are back at the problem of invariants across multiple composed objects…