SE 552: Lecture 8

Overview

Examples of concurrency controls

Observer/observed pattern

Transactions

Example: a Sync class

```
public interface Sync {
  public void acquire () throws InterruptedException;
  public void release ();
  public boolean attempt (long timeOut) throws InterruptedException;
}
```

How can we implement this interface to give a Mutex class?

How can we implement this interface to give a ReentrantLock class?

Example: a Sync class

How would we use a Sync class rather than a Java lock here:

```
public class BankAccount {
  int balance = 0; int withdrawls = 0; int deposits = 0;
  final Object lock = new Object ();
  public void deposit (final int amount) {
    synchronize (lock) {
      balance = balance + amount;
      deposits = deposits + amount;
    }
  }
  other methods go here
}
```

Example: a Sync class

Comparing using Java `synchronized (lock) { ... }` to
`lock.acquire (); ...; lock.release ();`:

- How do they deal with timeout?
- How do they deal with `Thread.interrupt`?
- What difference does it make that Java locks are block-structured?

Example: Read/Write locks

```
public interface ReadWriteLock {
  public void acquireReadLock () throws InterruptedException;
  public void acquireWriteLock () throws InterruptedException;
  public void releaseReadLock ();
  public void releaseWriteLock ();
  public boolean attemptReadLock (long timeout) throws InterruptedException;
  public boolean attemptWriteLock (long timeout) throws InterruptedException;
}
```

Example policy decisions:

- If there is an active writer, then nobody else can acquire a lock.
- There is no limit on the number of active readers, as long as there are no active writers.
- If there is a waiting reader or writer, then nobody can acquire a write lock.

Why are read/write locks useful?

How can we implement a read/write lock? (*Hint*: use counts of active readers, active writers, waiting readers, waiting writers.)

Example: Channels

```
public interface Channel {
  public void put (Object x) throws InterruptedException;
  public Object get () throws InterruptedException;
}
```

An IVar (or *latch*) implements Channel: put when full throws a PutException; get when empty blocks, get does not remove the contents.

An MVar implements Channel: the same as IVar but get removes the contents.

A Cell implements Channel: the same as MVar but put blocks when full.

A Buffer implements Channel: the same as Cell but put never blocks, and the buffer may contain many elements.

Multiple participants

Often a method call requires the cooperation of multiple objects or messages. This can cause problems!

Here, we'll look at:

- Oberver/observed pattern: notify multiple objects of a state change.
- Transactions: make multiple method calls thread-safe.

Observer/observed pattern

Often we want to have many *observer* objects observing a *subject* object. Whenever the subject changes state, the observers get told about it.

For example, the AWT event model works this way: each component is a subject, each event listener is an observer.

```
public interface Observer {
  public void changed (State newState);
}
public interface Subject {
  public State getState ();
  public void changeState (State newState);
  public void addObserver (Observer o);
  public void removeObserver (Observer o);
}
```

How can we implement this?

Transactions

Transactions are very heavyweight! But they are also very general purpose.

Solving a problem of multiple method calls:

```
interface IntRef {
  int get ();
  void set (int x);
}
void inc (IntRef r) {
   int x = r.get ();
   r.set (x+1);
}
```

How can we make inc thread-safe?

Transactions

The simplest kind of transaction system has empty transactions:

```
interface Transaction {
  public static final TransactionFactory factory = ...;
}
```

An interface for an object which supports transactions:

```
interface Transactor {
  public boolean join (Transaction t);
  public boolean canCommit (Transaction t);
  public void commit (Transaction t) throws Failure;
  public void abort (Transaction t);
  public static final TransactorFactory factory = ...;
}
```

What do these methods do? How should they be used?

Transactions

An example of using transactions:

```
interface TransIntRef extends Transactor {
  public int get (Transaction t) throws Failure;
  public set (Transaction t, int x) throws Failure;
}
```

Important property of transaction systems: the transaction either succeeds completely, or it fails with no effect.

How can we implement inc now?

Transactions

```
class TransIntRefImpl implements TransIntRef {
  int workingContents;
  int committedContents;
  Transaction current;
  final Object lock = new Object ();
  public int get (Transaction t) throws Failure { synchronized (lock) {
    if (current != t) { throw new Failure (); }
    return workingContents;
  } }
  other methods go here
}
```

How can we complete this implementation?

Transactions

Programming with these very simple transactions:

```
interface UsesTransactions extends Transactor {
    Result amethod (Transaction t, Argument arg) throws Failure;
    other methods go here
}
class UsesTransactionsImpl implements UsesTransactions {
    State workingState;
    State committedState;
    Transaction current;
    final Object lock = new Object ();
   Result aMethod (Transaction t, Argument arg) throws Failure { synchronized (lock) {
        if (t != current) { throw new Failure (); }
        ... perform some calculation on the working state ...
        return result;
    } }
    other methods go here
  }
}
```

Transactions

Scaling up...

More complex transactions allow for partial locking of the state, read-only vs read-write transactions, transactions involving more than one object...

Take a database course to find out more!

Summary

We can code common concurrency controls such as Mutextes, Locks, IVars, MVars, etc. using wait/notify.

Multiple participants such as observer/observed patterns can cause problems with concurrent programming.

One solution is transactions, but this is very heavyweight!

Next week: onto Chapter 4.