Threads 2

Java : Compile-Time Locks

Java uses a style where the lock acquisition structure is formally structured at compile time.

Structure

```
lock(x) {
  aaa
  bbb
}
```

vs. RT style (not Java)

```
{
  lock(x);
  aaa
  bbb
  unlock(x);
}
```

CT features

Can't mess up the balance -- exceptions, etc. -- always balance
Less flexible

Split Transaction Problem

Code

```
class Account {
  int balance;

  public synchronized int getBal() { return(balance); }
  public synchronized void setBal(int val) {balance = val;}
}
```

Problem

Two threads could interleave their calls to get/set just so to get the wrong answer.
The synch is at too fine a grain -- the critical section is larger

Solution

Move the synch out so it covers the whole transaction
`public synchronized changeBal(int delta) { balance += delta; }`

Like a Database -- leave in good state

DB's have an idea of a "transaction" -- a change that happens in full or is "rolled back" to not have happened at all.

Leave in good state

Think of your messages that way.. a message gets the lock, waits outside the lock, runs to completion, and leaves the object fully in the new state

Don't worry about order

If the above is true, you don't have to worry about the order the messages went through.

One Big Lock

This example puts one lock over the whole thing

```
public synchronized void foo() {
   int newValue = read1();
   newValue += read2();
   newValue = bigLookup(newValue);

   String result = new String("foo: " + newValue);

   length++;
   array[length] = result;
}
```

Fine Locks

```
public void foo() {
   int value;

   synchronized(this) {
      value = read1();
      value += read2();
   }

   value = dict.bigLookup(newValue);

   String result = new String("foo: " + newValue);

   synchronized(array) {
      length++;
      array[length] = result;
   }
}
```

read1 / read2 lock

Suppose that it's important that read1 and read2 reflect one state of the receiver. We obtain the receiver lock for the duration (depending on setters being synchronized).

read1 / read2 replacement

Maybe a better design would be that there's a one method replacement that effectively does read1/read2 in one operation.
Oop design: methods should meet the needs of the caller

dict.bigLookup()

Suppose the dict object has its own lock -- in this case, we don't need to hold both locks at once.

new String() + I/O

new is very expensive -- try not to be holding a lock when you do it (or I/O)
As above, the memory manager probably has its own lock

array lock

Suppose the array management code always locks the array first.

array method

Better would be to have a dedicated method, so the convention is more explicit...

```
public void addElt(String string) {
  synchronized(array) {
        length++;
        array[length] = string;
  }
}
```

Fine locks Pro: More concurrency

Having finer grain locks, allows more threads to be "in flight" at one time.

Fine locks Con: More cost

Acquiring each lock has a little cost. More locks -> more cost
Especially painful in the common case where we didn't have multiple threads anyway -- we're still paying the cost.

Fine locks Con: More complex

More locks to manage -- the "one big lock" model is conceptually simple

Fine locks Con: Deadlock

==Classic Deadlock Rules

Deadlock

Have locks x and y

One thread acquires x then y
Another thread acquires y then x

The Unhappy Caller

Some code you are calling (and didn't write) may depend on some internal lock, and so create the (x, y) situation without your knowing

One Deep Ok

If the code you call does its own thing and returns (no call backs to you) then deadlock cannot occur -- Yay!
EG Vector.addElementAt() can never cause deadlock -- example of Get In Get Out rule

#1: One Big Lock

If there's just one big lock, you won't have deadlocks.
Further, it's ok if you call things like new that have their own lock, but never come back and do something that depends on the one big lock

#2: Order The Locks

Establish by design, a fixed order for the locks
Everyone must acquire the locks in that order

Conclusion

Most likely to have problems when mixing separate code modules, each with some lock logic in it, and each calls the other.
There is no simple recipe to avoid the problem, it just requires overall understanding.
Simple strategy: have the "one big lock" for correctness, and revisit the decisions if concurrent efficiency is a real problem.

==Java Thread Syntax

Thread Class

Subclass

Implement run()

```
class Foo extends Thread {
  public void run() {
```

```
            // do whatever
    }
}
```

Plan to fall through bottom

Fall through the bottom of run() normally when done

1. Normally

2. Exception

Debug: catch / print exceptions

Catch / print exceptions in your run() so your thread doesn't die off silently when it gets an error

No re-use

Once a Thread is done with its run() it cannot be used again.

Runnable Interface

Implement Runnable

implement run() method -- same as Thread

Pass Runnable obj to Thread ctor

start(), not run()

thread.start()

At this point the thread may be scheduled for time.

Set up first

You can avoid some concurrency problems by getting everthing all set up, and then calling start

do not call run()

Thread vs. Thread of Control

A thread of control represents the real underlying access to CPU.
A Thread object is an object in memory that represents a TOC

Static Methods

TOC

These operate on the current thread of control

Not Thread

Not the thread object that happens to be named.

static Thread Thread.currentThread()

static void Thread.sleep()

static void Thread.yield()

Tragic Syntax #1

```
Thread t1 =  new ThreadSubclass();
t1.start();

t1.yield(); // this does not yield t1, it yields us
```

Tragic Syntax #2

The Thread object is still just a plain old object that messages can run against.
Some messages work on the current running thread, not the receiver Thread object -- yield() and sleep() (these are static in Thread)

```
class Foo extends Thread {
   int value;

   public void run() {
      for (int i = 0; i<1000; i++) {
         bar();
      }
   }

   public synchronized bar() {
      i = i +1;
```

```
      Thread.yield();   // Works on the caller thread
      this.yield();     // NOT the receiver Foo object
      // (Thread.currentThread() == this) is FALSE for the bar() call below
   }
}

test {
   Foo foo = new Foo();
   foo.start();

   foo.bar();  // This causes a yield() on our thread, not the Foo object
}
```

Swing Threading

- There's a special Swing thread (we'll think of it as one thread, although it could be several threads cooperating with locks)
- Dequeues real time user events
- Translates to paint() and action notifications
- Once a swing component is subject to pack() or setVisible, no other thread should send it Swing sensitve messages such as add(), setPreferredSize(), getText ...

The Giant Swing Mutex

- Like a giant mutex over all the Swing state -- only one thread is allowed to touch Swing state
- EG how could paint(), and pack() work if values were simultaneously changing?
- They are essentially using the One Big Lock strategy on all of Swing
- I've come to decide that this is actually a very reasonable way to design Swing

1. Swing Safe

- A few methods are valid to call against Swing, even if you are not the Swing thread...
- repaint(), revalidate(), addXXXListener(), removeXXListener()

2. On the Swing Thread Anyway

- As long as you are just responding in, say, action performed, you are in the Swing thread, so do whatever you want.
- Just don't do something that blocks or takes a long time -- for something costly, create a separate thread and have report back (see below) when it has something.

3. Invoke The Swing Thread

If you are not in the Swing thread, get the Swing thread to do something for you...

Returns immediately :SwingUtilites.invokeLater(new Runnable() { public void run() { ...}

Blocks: SwingUtilities.invokeAndWait(new Runnable() { ...