Nick Parlante

Swing3 + Threads1

HW #1

A couple things I fixed from the first version of the handout...
"FilteredDBModel" and "FilteredTableModel" are the same thing
The main class should be DBFrame

Swing Recap

I'll go over the things I couldn't show last week with the video converter box problems.

Model Classes x 3

ListModel
AbstractListModel
DefaultListModel

ListModel Data

ListModel Niche

The view sends these methods to get the data. Any object that responds to these can be the model for a list.

Methods

```
int getSize();
Object getElementAt(int index);
```

ListModel Notification

Niche

The model must keep a list of listeners. When the model changes in certain ways, it must notify the listeners of the change. AbstractListModel has support code for the listeners. Use the fireXXX methods to notify the listeners of the various changes.

Methods

```
fireIntervalAdded(this, int, int)
fireIntervalRemoved(this, int, int)
fireContentsChanged(this, int, int)
```

Threading

Thread level vs. Process Level
Threads share address space
OS's now support "inexpensive" threads -- on the order of 10-50 per process
Separate processes are heavyweight -- separate address space, large start-up cost

Multiple processors

CPU intensive could get value from extra processor (but why code in Java for CPU bound problem?)
Memory intensive less so
Disk / Network intensive even less so

Network / Disk -- Hide The Latency

Use threads to efficiently block when data is not there
Even with one CPU, can get excellent results
Suppose very fast CPU, and very slow network -- even with coarse locking, may get excellent results. The threads are blocked most of the time anyway, so lock contention is not really a problem.
This is what Java threads are really good for.

Why Concurrency Is Hard

No language construct can make the problem go away (in contrast to mem management which was made to go away with GC). The programmer must be involved.
There is no fixed programmer recipe that will just make the problem go away.
Hard for classes to pass the "clueless client" test -- the client may really need to understand the internal lock model of a class to use it correctly.
Concurrency bugs are very, very latent. The easiest bugs are the ones that happen every time.
In contrast, concurrency bugs show up rarely, they are very machine, VM, and current machine loading dependent, and as a result they are hard to repeat.
"Concurrency bugs -- the memory bugs of the 21st century."
Rule of thumb: if you see something bizarre happen, don't just pretend it didn't happen. Note the current state as best you can.

Native vs. Green

Thread Implementation

Green = 1 native thread -- easiest to implement
Native = 1 native thread for each Java thread -- most common
Mixed = n native threads for k Java threads

Coding Strategies

Cooperative "green" threads -- schedule on yield(), sleep(), lock acquire (through system call)

In that case, your code should call yield() every now and then.

Native "preemptive" threads -- threads may be scheduled on above + preemptively

If a program works in green threads, it may still fail with native threads.

Green Reliability

Green threads are less likely to expose concurrency bugs since they do not take away the thread of control in the middle of some statements.

```
{
   i = i +1;   // won't loose it here
   next = a[i];   // or here
   foo();   // maybe here, depending on what foo does
}
```

1. Classic Critical Section Problem

```
class Foo {
  int i;

  void incr() {
        i = i + 1;
  }
}
```

2. synchronized

Compile-time

Part of the source code structure

Acquire the lock on the receiver

equivalent to synchronized(this)

Errors

Most common errors derive from loosing track of which lock has been synchronized.

3. Classic synchronized solution

Synch lock on the receiver

```
synchronized void incr() {
  i = i +1;
}
```

Result

Acquires the lock on this -- any other code that uses that lock will block while we're in this section.

Common Synch Errors

1. Error - must volunteer to be synchronized

```
void decr() {
  i = i -1;
}
```

Only methods that are synchronized are locked out. In this case, decr() can still get in while incr() holds the lock.

2. Error - static methods do not synch on an instance

```
static void incrObj(Foo foo) {
  foo.i = foo.i + 1;
}
```

Solution

Having a static method change the state of an object is weird, but if we ignore that, the solution would be to block on the same lock as the regular synchronized methods...

```
static void incrObj(Foo foo) {
  synchronized(foo) {
        foo.i = foo.i + 1;
  }
}
```

3. Error - Shared Static

```
static int count;
synchronized binky() {
  count = count + 1;
}
```

Problem

binky() will not be running concurrently against one object, but with multiple objects, it could be running concurrently against multiple objects.

a. synch(this)

```
void binky() {
  synchronized(this) {
        count = count+1;
  }
```

```
}
```

b. synch(lock)

Add a dedicated lock object used for count...

```
static int count;
static Object countLock = new Object();
void binky() {
  synchronized(countLock) {
        count = count + 1;
  }
}
```

4. Error - Shared Object

```
int[] a;   // suppose all Foo's share a pointer to one a obj
syncronized void binky() {
  a[0] = a[0] + 1;
}
```

Solution

```
void binky() {
  synchronized(a) {
        a[0] = a[0] + 1;
  }
}
```