SE 552: Lecture 3

Overview

Homework

Wait/notify

Exclusion

Immutability

Homework review

Wait/notify

Objects often have *state*.

Often, threads have to block waiting for an object to change state.

For example, in the GuardedLogic implementation of the Jack application, we had:

```
while (true) {
    Thread.sleep (200);
    flag.waitForTrue ();
    printChar ();
}
```

What does `flag.waitForTrue ()` do?

Wait/notify

```
class GuardImpl implements Guard {
    protected boolean value;
    protected Object lock = new Object ();
    ...
    public void setValue (final boolean value) {
        synchronized (lock) {
            this.value = value;
            if (value) { lock.notifyAll (); }
        }
    }
    public void waitForTrue () throws InterruptedException {
        if (!value) { synchronized (lock) {
            if (!value) { lock.wait (); }
        } }
    }
}
```

What does this code do? Try TestGuard.java.

Why the *double check* in waitForTrue?

Wait/notify

Methods used by wait/notify in the Object class:

- wait ()
- wait (timeout)
- notify ()
- notifyAll ()

What do these methods do?

What exceptions can these methods throw?

Wait/notify example

An unsafe class for buffers: UnsafeBuffer.java

How would we change this class so that calls to get and put blocked rather than throwing exceptions?

Implementation

How are locks and wait/notify implemented?

The rest of the Thread API

There are a few other bits and pieces: daemon threads, thread groups, suspend/resume, stop...

But these are either deprecated or (fairly) straightforward.

As far as this course is concerned you've now seen all of Java's concurrent programming support!

Exclusion

Reminder: what is an invariant? A critical section?

Locks are used to ensure that only one thread is in a critical section at one time: *exclusion*.

Strategies for ensuring exclusion:

- *Eliminating* critical sections!
- *Dynamically* ensuring exclusion (using locks).
- *Statically* ensuring exclusion (make sure only one thread ever has access to the object).

Race conditions

What is a race condition?

What is a read-write conflict? A write-write conflict?

Can we get read-read conflicts?

Immutable programming

What is a *final* field?

What is an *immutable* class?

```
class BankBalance {
  int balance=0; int deposits=0; int withdrawls=0;
  void deposit (final int amount) {
    deposits=deposits+amount; balance=balance+amount;
  }
  void withdraw (final int amount) {
    withdrawls=withdrawls+amount; balance=balance-amount;
  }
  ...
}
```

Is this class thread-safe? Is it immutable? If not, can we make it immutable? Is the immutable class thread-safe?

Immutable programming

General pattern for mutable programming:

```
class Foo {
  AType field;
  Foo (final AType init) { field = init; }
  BType getSomething () { return ...some value...; }
  void setSomething () { field = ...some value...; }
}
```

General pattern for immutable programming:

```
class Foo {
  final AType field;
  Foo (final AType init) { field = init; }
  BType getSomething () { return ...some value...; }
  Foo setSomething () { return new Foo (...some value...); }
}
```

Note: the set methods return new objects!

Immutable programming

Real example: linked lists.

```
public interface ImmutableList {
    public ImmutableList cons (Object head);
    public int size ();
    public Object head ();
    public ImmutableList tail ();
    public ImmutableList remove (Object element);
    public Iterator iterator ();
}
```

How can we implement this?

The complete source code for ImmutableList is on-line.

Immutable programming

A problem with immutable programming... consider:

```
class Integer {
  final int contents;
  Integer (final int contents) { this.contents = contents; }
  ...
}
```

...but a program may use a lot of Integer objects.

This is expensive! How can we reduce the number of Integer objects generated?

Hint: flyweight pattern.

A real use of the flyweight pattern is in FlyweightImmutableList which uses flyweights for 'hash consing'.

Immutable programming

Mutable *wrapper classes* are very common:

```
class ImmutableFoo { ... }
class MutableFoo {
  ImmutableFoo contents = new ImmutableFoo ();
  SomeType getSomething () { return contents.getSomething (); }
  synchronized void setSomething () { contents = contents.setSomething (); }
}
```

Do we need to synchronize the setSomething method?

A real example: MutableList.

Immutable programming

Be careful about constructors!

What can go wrong here:

```
class Foo extends Runnable {
  final String msg;
  Foo (final String msg) {
    callSomeMethod (this);
    this.msg = msg;
  }
  ...
}
```

A good rule of thumb: only initialize fields in constructors!

Immutable programming

Pros and cons of immutable programming...

Hints for pros: consider thread safety and sharing.

Hints for cons: what data structures can you implement this way?

Personal opinion: you can't always program immutably, but when you can, do so!

Summary

Exclusive access to critical sections is important for thread safety.

One way to achieve this is to have no critical sections! This means immutable programming.

Immutable programming is very powerful (Turing machine complete) but not all data structures can be implemented this way.

Next: dynamic exclusion.

Reading: Lea, Chapter 2, especially Section 2.2.