Sharing and Confinement

Sharing: The root of our problems…
Generally speaking, I’ve noted that the different issues we’ll be seeing this semester related to concurrent programming can be summarized as “shared state” problems.

In a single-threaded program, we didn’t have these problems.

What if we designed things so that there was no sharing of state across threads?
**Guarded Access**

What we’ve mostly been talking about recently is one form of the notion of “guarded” access to shared state. Everyone agrees that they will only access the shared item if they hold a certain lock (such as an intrinsic lock on the item, or an intrinsic lock logically connected to the item).

It is important to make sure that all instructions related to an action including the shared item are seen as if they were atomic by holding the lock continuously during that set of instruction.

**Ad-hoc Confinement**

You could decide on a set of conventions to restrict the use of shared state and then work to have all the code follow them. You agree on an order of who gets to do what and when that ensures locks aren’t needed. If you manage it, that’s great. However, you are using 100% human-level protection rather than anything language-level, which can be difficult.

Part of an ad-hoc agreement could be that you share state in a very specific way; everything that’s shared is marked as `volatile` and for each shared object, only one thread is ever allowed to be able to write to it. In a single-writer, multiple-reader scenario, you don’t need synchronization if the variable is `volatile`.

Very fragile and very subtle!
Stack Confinement

This is a semi-literal descriptive name for an agreement between all programmers that you essentially never have a shared state. Each thread is only allowed to modify memory on its own stack.

The basic convention here is that if you want to modify an object, you had to have created it in that thread and you never publish that object outside the thread (note that this newly created object would be in the heap not the stack, which is why the name is only semi-literally descriptive).

Ad-hoc idea motivated by stack confinement

Consider the following ad-hoc set of rules where a thread can only publish data that it was working on as it is ending.

– When the thread starts, it gets a deep copy of what it needs passed into the constructor for the Thread–based object.
– During the thread’s lifetime, it works solely on that copy, possibly via a member field in the object, possibly via a local reference to it or even another copy of it.
– When the thread is about to finish running, it makes sure the information is back in that member field in the Thread–based object rather than on its own stack and then once a join() happens in the spawning thread, that can be read out of the Thread–based object (which can't be restarted).

Do you think this would be easy for all programmers on a team to follow?
**Where do static values live?**

When a class has a static data member, where is that variable stored?

In the current version of the Java Memory Model, it is stored in something called the “Permanent Generation” area of memory.

It is somewhat like the heap in that this area can have items added to it as needed and stay around until garbage collection notices that they are no longer accessible.

It is also like the heap as it is generally part of the shared state of a program execution, so if multiple threads were to access and modify a static variable, you’d want to protect it.

**ThreadLocal<T> Confinement**

There is a special mechanism in Java for creating static class variables where for all intents and purposes there is actually a different static class variable for each thread that uses the class!

This is a form of thread-based confinement and is accomplished by using `ThreadLocal<T>` static variables.

The `ThreadLocal<T>` methods are a parameter-less constructor, `initialValue()`, `get()`, `set(T)`, and `remove()`.

ThreadLocal<T> Method Descriptions

initialValue(): invoked by the get() method as described below.

get(): action is twofold; [1] if the variable has no value yet for this thread it invokes initialValue() and then [2] returns the current thread’s value for this variable.

set(T): sets the value of this thread’s copy of the variable to what was passed into this method.

remove(): marks the thread’s copy of the variable as having no value anymore.

Imagine a class that keeps track of its instances:

```java
public class IntegerHolder {
    private static Integer totalMade = 0;
    static public int getHowMany() {
        return totalMade;
    }

    private int val;
    public IntegerHolder(int inVal) {
        val = inVal;
        totalMade++;
    }

    public int getValue() {
        return val;
    }
}
```
What if each thread wanted to track its instances:

```java
public class ThreadLocalIntegerHolder {
    private static ThreadLocal<Integer> totalMade =
        new ThreadLocal<Integer>() {
            protected Integer initialValue () {return 0;}
        };
    static public int getHowMany() {
        return totalMade.get();
    }
    private int val;
    public ThreadLocalIntegerHolder(int inVal) {
        val = inVal;
        totalMade.set(totalMade.get()+1);
    }
    public int getValue() {
        return val;
    }
}
```

Thread-level counting rather than Object-level:

```java
public class ThreadNumberTag {
    //Global unique ID
    private static int nextNumber = 0;
    private static ThreadLocal<Integer> numberTag =
        new ThreadLocal<Integer>() {
            protected synchronized Integer initialValue() {
                return new Integer(nextNumber++);
            }
        };
    public static Integer getID() {
        return numberTag.get();
    }
}
```
**Shared Read-Only and Shared Thread-Safe**

Another “easy” convention for a concurrent program to use is that shared objects are to be **read-only** in the spawned threads.

Also, if we have objects which are themselves guaranteed as thread-safe (ie: they do their own internal locking so the outside world can’t misuse them) then there is generally no need to worry about our threads being careful when using them (though we may see an example later where you would want to make sure you do **not** do additional locking on an already thread-safe object).