6.005 Software Construction
Fall 2011
Prof. Rob Miller

L12: Thread Safety

Today

- Confinement
- Threadsafe datatypes

Required reading

- Concurrency
- Wrapper Collections

Optional reading

The material in this lecture and the next lecture is inspired by an excellent book:

- Brian Goetz et al. *Java Concurrency in Practice*. Addison-Wesley, 2006.

Review

Recall race conditions: multiple threads sharing the same mutable variable without coordinating what they're doing. This is unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations.

There are basically four ways to make variable access safe in shared-memory concurrency:

- **don't share** the variable between threads. This idea is called *confinement*, and we'll explore it today.
- make the shared data **immutable**. We've talked a lot about immutability already, but there are some additional constraints for concurrent programming that we'll talk about in this lecture.
- encapsulate the shared data in a **threadsafe datatype** that does the coordination for you. We'll talk about that today.
- use **synchronization** to keep the threads from accessing the variable at the same time. Synchronization is what you need to build your own threadsafe datatype. We'll talk about that next time.

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright held by the authors.

Threads

A look at how to create threads in Java. For more details, see the Java tutorial link above.

```
    public void serve() throws IOException {
        ServerSocket serverSocket = new ServerSocket(PORT);

        while (true) {
            // block until a client connects
            final Socket socket = serverSocket.accept();

            // start a new thread to handle the connection
            Thread thread = new Thread(new Runnable() {
                public void run() {
                    // the client socket object is now owned by this thread,
                    // and mustn't be touched again in the main thread
                    handle(socket);
                }
            });
            thread.start(); // IMPORTANT! easy to forget
            // when does thread.start() return?
            // when will the thread stop?
        }
    }
```

Thread Confinement

Thread confinement is a simple idea: you avoid races on mutable data by keeping that data confined to a single thread. Don't give any other threads the ability to read or write the data directly.

Local variables are always thread confined! A local variable is stored in the stack, and each thread has its own stack. There may be multiple invocations of a method running at a time (in different threads or even at different levels of a single thread's stack, if the method is recursive), but each of those invocations has its own private copy of the variable, so the variable itself is confined.

But be careful – the *variable* is thread confined, but if it's an object reference, you also need to check the *object* it points to. If the object is mutable, then we want to check that the object is confined as well – there can't be references to it that are reachable from any other thread.

What about local variables that are initially shared with a thread's Runnable when it starts, like *socket* in the code above? Java requires those variables to be final! Immutable references are okay to use from multiple threads, because you don't get races with immutability.

Avoid Global Variables

Unlike local variables, global variables (called "static" in Java) are *not* automatically thread confined.

If you have static variables in your program, then you have to make an argument that only one thread will ever use them, and you have to document that fact clearly. Better, you should eliminate the static variables entirely.

Here's an example that we looked at in a previous lecture:

```
// This class has a race condition in it.
public class Midi {

    private static Midi midi = null;
    // invariant: there should never be more than one Midi object created
    private Midi() {
        System.out.println("created a Midi object");
    }

    // factory method that returns the sole Midi object, creating it if it
doesn't exist
    public static Midi getInstance() {
        if (midi == null) {
            midi = new Midi();
        }
        return midi;
    }
}
```

This class has a race in the getInstance() method – two threads could call it at the same time and end up creating two copies of the Midi object, which we don't want.

To fix this race using the thread confinement approach, you would specify that only a certain thread (maybe the "midi playing thread") is allowed to call Midi.getInstance(). The risk here is that Java won't help you guarantee this.

In general, static variables are very risky for concurrency. They might be hiding behind an innocuous function that seems to have no side-effects or mutations. Consider this example:

```
    // is this method threadsafe?
    /**
     * @param x integer to test for primeness; requires x > 1
     * @return true if and only if x is prime
     */
    public static boolean isPrime(int x) {
        if (cache.containsKey(x)) return cache.get(x);
        boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
        cache.put(x, answer);
        return answer;
    }

    private static Map<Integer,Boolean> cache = new HashMap<Integer,Boolean>();
```

This method is not safe to call from multiple threads, and its clients may not even realize it.

Threadsafe

A datatype is *threadsafe* if it behaves correctly when used from multiple threads, regardless of how those threads are executed, and without demanding additional coordination from the calling code.

- "behaves correctly" means satisfying its specification and preserving its rep invariant
- "how threads are executed" means threads might be on multiple processors or timesliced on the same processor
- "without additional coordination" means that the datatype can't put preconditions on its caller related to timing, like "you can't call get() while set() is in progress."

Remember Iterator? It's not threadsafe. Iterator's specification says that you can't modify a collection at the same time as you're iterating over it. That's a precondition put on the caller, and Iterator makes no guarantee to behave correctly if you violate it. So it's not threadsafe.

Immutability

Final variables are constants, so the variable itself is threadsafe.

Immutable objects are generally also threadsafe. We say generally because our current definition of immutability is too loose for concurrent programming. We've said that a type is immutable if an object of the type always represents the same abstract value for its entire lifetime. But that actually allows the type the freedom to mutate its rep, as long as those mutations are invisible to clients. We saw an example of this notion, called *benevolent* or *beneficient mutation*, when we looked at an immutable list that cached its length in a mutable field the first time the length was requested by a client. Caching is a typical kind of beneficent mutation.

For concurrency, though, this kind of hidden mutation is a no-go. An immutable datatype that uses beneficent mutation will have to make itself threadsafe using locks (the same technique required of mutable datatypes), which we'll talk about next lecture.

So in order to be confident that an immutable datatype is threadsafe *without* locks, we need stronger rules:

- no mutator methods
- all fields are private and final
- no mutation whatsoever of mutable objects in the rep -- not even beneficent mutation
- no rep exposure (see the ADT lecture to review what rep exposure is)

If you follow these rules, then you can be confident that your immutable type will also be threadsafe.

Threadsafe Collections

The collection interfaces in Java – List, Set, Map – have basic implementations which are *not* threadsafe. The implementations of these that you've been used to using, namely ArrayList, HashMap, and HashSet, cannot be used safely from more than one thread.

Fortunately, just like the Collections API provides wrapper methods that make collections immutable, it provides another set of wrapper methods to make collections threadsafe, while still mutable.

These wrappers effectively make each method of the collection **atomic**. An atomic action effectively happens all at once – it doesn't interleave its internal operations with other threads, and none of the effects of the action are visible to other threads until the entire action is complete, so it never looks partially done.

Now we see a way to fix that isPrime() method we had earlier in the lecture:

```
    private static Map<Integer,Boolean> cache = Collections.synchronizedMap(
new HashMap<Integer,Boolean>() );
```

A few points here.

First, make sure to throw away references to the underlying non-threadsafe collection, and access it only through the synchronized wrapper. That happens automatically in the line of code above, since the new HashMap is passed only to synchronizedMap() and never stored anywhere else.

Second, even though method calls on the collection itself (get(), put(), add(), etc.) are now threadsafe, iterators created from the collection are still *not* threadsafe. So you can't use iterator(), or the for loop syntax:

```
for (String s: lst) { .... }  // not threadsafe, even if lst is a synchronized
list wrapper
```

The solution to this problem will be to acquire the collection's lock when you need to iterate over it, which we'll talk about next time.

Finally, the way that you *use* the synchronized collection can still have a race condition! Consider this code, which checks whether a list has at least one element and then gets that element:

```
if (! lst.isEmpty()) { String s = lst.get(0); ... }
```

Even if you make lst into a synchronized list, this code still may have a race condition, because another thread may remove the element between the isEmpty() call and the get() call.

Even the isPrime() method still has potential races:

```
if (cache.containsKey(x)) return cache.get(x);
boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
cache.put(x, answer);
```

The synchronized map ensures that containsKey(), get(), and put() are now atomic, so using them from multiple threads won't damage the rep invariant of the map. But those three operations can now interleave in arbitrary ways with each other which might break the invariant that isPrime needs from the cache: if the cache maps an integer x to a value f, then x is prime if and only if f is true. If the cache ever fails this invariant, then we might return the wrong result.

So we have to argue that the races between containsKey(), get(), and put() don't threaten this invariant. First, the race between containsKey() and get() is not harmful because we never remove items from the cache – once it contains a result for x, it will continue to do so. Second, there's a race between containsKey() and put(). As a result, it may end up that two threads will both test the primeness of the same x at the same time, and both will race to call put() with the answer. But both of them should call put() with the same answer, so the race will be harmless.

The need to make these kinds of careful arguments about safety – even when you're using threadsafe datatypes – is the main reason that concurrency is hard.

Goals of Concurrent Program Design

Now is a good time to pop up a level and look at what we're doing. Recall that the primary goals of this course are to learn how to create software that is (1) safe from bugs, (2) easy to understand, and (3) ready for change. There are other properties of software that are important, like performance, usability, security, etc., but we're deferring those properties for the sake of this course.

Building concurrent software has these same overall goals, but they break down more specifically into some common classes. In particular, when we ask whether a concurrent program is safe from bugs, we care about two properties:

- **Safety**. Does the concurrent program satisfy its invariants and its specifications? Races in accessing mutable data threaten safety. Another way to put this is, can you prove that **nothing bad ever happens**?
- **Liveness**. Does the program keep running and eventually do what you want, or does it get stuck waiting forever for events that will never happen? Can you prove that **something good eventually happens**? Deadlocks threaten liveness. Liveness may also require *fairness*, which means that concurrent modules are able to make progress in their computations when they are actually able to run. If Eclipse's editor module hogs the only processor in the system, so that the compiler module never gets a chance to run, then you won't ever get your program compiled – a liveness failure. Fairness is mostly a matter for the operating system's thread scheduler, which decides how to timeslice threads, but you can influence the scheduler's decisions with mechanisms like thread priorities, so it's possible for a system design to threaten fairness.

Concurrent programs also usually worry about **performance**, i.e. the speed or resource usage of the program, since that is often the main reason for introducing concurrency into the system in the first place (making the program work faster or respond more quickly). We've largely been postponing issues of performance in 6.005. 6.172 Performance Engineering is strongly recommended for learning about this, and it covers performance of concurrent programs in great detail. But here are some high-level comments about getting good performance with threads.

Create only a few threads. Threads cost resources – memory for a stack, processor time to switch threads, operating system resources. So don't create them as freely as you create data objects. There are typically two reasons why you create threads: to do I/O (e.g. network, MIDI device, graphical user interface), or to handle computation. Each has some rules of thumb:

- For I/O: create at most one thread per stream (often one for reading and one for writing), so that it can block without preventing other streams from making progress.
- For computation: the sweet spot is slightly more threads than you have processors (Runtime.getRuntime().availableProcessors()). If you start 100 threads but you have only 4 processors for them to run on, then you're just wasting memory and time, and you won't finish the job any faster than 4 threads would. Java has an interface called ExecutorService that manages a pool of threads for a queue of tasks, so you can chop your computation up into bits and use as many threads as make sense to do them.

Don't move work between threads unnecessarily. Sending requests to other threads is expensive; switching threads on a processor is expensive; moving data between threads is expensive; synchronizing and coordinating between threads is expensive. So if thread A has a piece of work that needs to get done, and it has all the data it needs to do the work and can do it safely (without races), then A should just do the work itself, rather than handing it off to another thread B. If an I/O thread in a server receives a request that doesn't require accessing mutable data that is confined to another thread, it should just handle the request itself, rather than sending it to another thread.

How to Make a Safety Argument

We've seen that concurrency is hard to test and debug. So if you want to convince yourself and others that your concurrent program is correct, the best approach is to **make an explicit argument** that it's free from races and deadlocks.

We're going to focus for now on the safety question. Your argument needs to catalog all the threads that are exist in your module or program, and the data that that they use, and argue which of the four techniques you are using to protect against races for each data object or variable: confinement, immutability, threadsafe datatypes, or synchronization. (When you use the last two, you also need to argue that all accesses to the data are appropriately atomic – that is, that the invariants you depend on are not threatened by interleaving. We gave one of those arguments for isPrime above.)

The SocialServer code with this lecture has an example of a safety argument:

```
//
// Thread safety argument
// --------------------------
// The threads in the system are:
// - main thread accepting new connections
// - one thread per connected client, handling just that client
//
// The serverSocket object is confined to the main thread.
//
// The Socket object for a client is confined to that client's thread;
// the main thread loses its reference to the object right after starting
// the client thread.
//
// The friendsOf map and all the lists inside it are confined to the main
thread
// during creation and then immutable after creation.
//
// System.err is used by all threads for displaying error messages.
// No other shared mutable data.
//
```

6.005 Elements of Software Construction
Fall 2011