Synchronization in Java

Ronald L. Rockhold

1/19/2006

Introduction

Java's synchronization model is based on another model, the Monitor. Like Hoare's (and Hansen's) Monitor model, Java's synchronization is used to protect shared resources from simultaneous access/manipulation in a multi-threaded environment.

Synchronization has costs. One component of which is the overhead of the synchronization implementation, e.g. acquiring/releasing a lock. Another, less obvious one, is the level of interference between multiple computations that do not require mutual exclusion, but manage to prevent each other from otherwise executing some code "simultaneously". If two computations can safely take place simultaneously, but a mechanism for providing exclusion prevents this overlap, performance can be affected. If, for example, a single shared lock is used to control access to all critical sections, even safe combinations, then throughput can suffer.

For this reason, critical sections are typically (logically) correlated with the data that each manipulates, and a lock is associated with that data. The critical section is then protected by a protocol (an agreement) requiring that the specific lock related to the specific instance of data needing protected access is first acquired.

Protecting Instance Variables

If a section of code must modify the internal fields of a specific object of type ***T***, then class ***T*** can be "given" a lock as an instance variable, and the critical sections operating on instances ***t1*** of type ***T***, first acquire ***t1.lock***. This allows different instances of ***T*** to be manipulated simultaneously by multiple threads, even by the same code.

```
/* Some C/C++ code */
struct T {
     /* data fields, including: */
     int count;
     struct   Lock   t_lock;
};

void foo(struct T * t) {
     lock_acquire(t->t_lock);     /* critical section entry */
     count += 2;                  /* critcal section */
     lock_release(t->t_lock);     /* critical section exit */
}
```

Since each instance of ***T*** has a different Lock, function ***foo()*** can be executing simultaneously in its critical section provided its argument ***t*** is pointing to different

instances of ***T***. But for those threads calling ***foo()*** with the same ***T*** instance, execution of the critical section will be serialized.

It is critical to note that acquiring lock ***t_lock*** for some instance of ***T*** in no way protects the contents of that instance. For example:

```
void spam(struct T* t) {
     t->count += 4;
}
```

Here, the ***spam()*** function will modify count for whatever instance it's passed, even if some other thread has acquired the lock ***t->m_lock***.

So, critical sections are protected by a protocol, an "agreement" between programmers, that specifies a process that, if everyone follows, will ensure the coherence of shared data. For instances of ***T***, the protocol is that critical section entry requires acquiring lock ***m_lock***; exit consists of releasing ***m_lock***.

Note that we could have achieved safety in our critical sections by using a single lock shared by all the ***T*** instances. However, this would ***not*** have allowed different ***T*** instances to be accessed simultaneously, i.e. lock granularity would be too coarse.

Synchronized Blocks

The approach of defining a synchronization variable for each instance of a type is so common in multi-threaded environments, that Java does this automatically. In Java, every object (remember that an object is an instance of some class or is an array) has a (unique) monitor associated with it. We'll talk more about Java's monitors, but for now, think of a monitor as a lock (with a condition variable).
Here's the same code from above, this time using Java:

```
class T {
     /* data fields, including: */
     int count = 0;
     //  Not needed:   struct   Lock   t_lock;

     void foo() {  /* instance method – i.e. not static */
          synchronized(this) {    /* critical section entry */
               count += 2;        /* critical section */
          }                       /* critical section exit */
     }
}
```

The Java synchronized statement is of the general form:

```
synchronized(object_handle) {  /* synchronized block */  }
```

The synchronized block is not executed until the monitor associated with ***object_handle*** is acquired by the currently running Thread (only one Thread at a time can "own" the monitor). When the synchronized block is exited (e.g. return, falling off the end, un-caught exception), then the monitor is automatically released.
So Java synchronization is block-based; the monitor is acquired before the block is entered, and it's released (always and only – there is no "un-synchronized" keyword) when the synchronized block is exited. Unlike the lock model, there is no ***explicit*** mechanism for releasing a monitor.

Like the lock model, there is nothing about acquiring an object's monitor that in any way (other than programming protocol) protects the instance data of the object. If we add this ***spam()*** method to ***T*** :

```
void spam() {
     count += 4;
}
```

Then our class instance methods are no longer thread-safe. If some other Thread "owns" the monitor for some instance ***t1*** of ***T*** , and another Thread invokes ***t1.spam()***, both Threads can be modifying ***t1*** simultaneously.

Synchronized Instance Methods

Using the ***synchronized(this)*** block is so common in instance methods that Java provides a shortened notation for specifying that an instance method is synchronized against its invoking object. These are functionally equivalent definitions of instance method ***foo***:

```
synchronized void foo() {

     count += 2;

}
```

```
void foo() {
     synchronized(this) {
               count += 2;
     }
}
```

When a synchronized block is exited, another Thread can enter the monitor that's now "free". If there had been Threads waiting for the monitor, exactly one is allowed in. Java does not specify an ordering, so the Thread waiting the longest ***may not be the one allowed into the monitor***.

Protecting Class Variables

So far our examples have focused on protecting access to per-object data. Sometimes there is data maintained per-class, rather than per-instance, that must be protected. In Java, per-class data are static members ("class variables"), and per-instance data are non-static ("instance variables").

Protecting class variables requires a protocol that ensures all critical sections synchronize against the same object. It wouldn't work if one critical section used one object instance, and another critical section synchronized against a different object instance.
What's needed is an object of which there is only one for our class. As it turns out, every Java class has such an object, it's the class's java.lang.Class object. For any class ***S***, a handle to its Class object can be retrieved in various ways, but the compiler allows us to generate a handle to the Class object by using the "***S.class***" notation. Here's a class with class data (static data). Note that changes to class data are protected by synchronizing against the class' Class object:

```
class S {
     static int count = 0;
     static void eggs() {
          synchronized (S.class) {
               count += 1;
          }
     }
}
```

Protecting class data in this way is so common that, once again, Java allows for a shorthand notation. These implementations are identical:

```
synchronized static void eggs() {
     count += 2;
}
```

```
static void eggs() {
     synchronized(S.class) {
          count += 2;
     }
}
```

We've already covered the notion that every object (including array "objects"), has an associated Monitor, and that entering/leaving the "monitor" is accomplished with the Java synchronized block. The monitor is entered prior to beginning execution of the block; it is exited implicitly when the block is exited.

Monitors – Condition Variables, Wait/Notify

Like Hoare's monitor model, Java's monitors support the ability to (atomically) wait on a condition variable and release the monitor. Every monitor has an associated Condition Variable (CV). Although Hoare's model allows for an unlimited number of named CVs, in Java, each monitor has only a single (nameless/anonymous) CV.

Hoare's Wait(cv) operation is represented in Java by the ***wait()*** method. The ***wait()*** method is defined in class java.lang.Object (the root ancestor of every Java class), and is final (cannot be overridden.) So, ***wait()*** (like all instance methods) is invoked on an object instance.

In Java, as in Hoare's monitor, a ***wait()*** operation can only be requested by a caller who is executing in ("owns") the CV's monitor. So when ***x.wait()*** is executed, the invariant is that the calling thread must be executing in the monitor associated with ***x***:

```
class T {
      void foo(Object x) {
          ….
          synchronized (x) {
               ….
               x.wait(); // Note that a runtime exception would be
                         // thrown is we simply coded "wait()",
                    //  which the compiler would 'expand' to
                    //  this.wait(), because we're not currently
                    //  executing in the monitor for "this" (unless by
some strange coincidence this == x).
          }
      }
}
```

As in Hoare's Wait(), Java atomically blocks the caller and releases the monitor associated with ***x***.

Why doesn't Java just let you code a 'naked' ***wait()***? Well, it's because synchronized blocks can be nested, and you have to explicitly tell Java which monitor's (anonymous) CV you're going to wait on:

```
void spam() {
     Object y = new Object();
     Object z = new Object();
     synchronized (y) {
          synchronized(z) {
               z.wait(); // or y.wait()
          }
     }
}
```

You must, however, be very careful when you call ***wait()*** when you're code is in nested synchronized blocks. It turns out that Java only releases one of the monitors – the one on which you invoked ***wait()***. The other monitors are still locked, which leads to quick deadlocks!

Java's Signal() counterpart is the ***Object.notify()*** method. Again, ***notify()*** must be invoked on an object whose monitor is currently "owned" by the caller:

```
 void bar(Object x) {
      synchronized (x) {
           x.notify();
           …
      }
```

}

When ***notify()*** is called on an object, Java checks its monitor to see if there are any Threads that are blocked (by having called ***wait()*** on the object). If so, one of them (which one is unspecified) is awaked and added to the list of Threads trying to execute in the monitor. When that Thread does again "own" the monitor, it executes the next statement after its ***wait()*** call (if there are no waiting Threads, the ***notify()*** has no effect.) Note that Java is implementing the ***Hansen version*** of Signal() where the Thread calling ***notify()*** continues to execute (still owning the monitor), and the awakened Thread has no opportunity to execute until the ***notify()ing*** Thread leaves the monitor (the awakened thread competes with any other Threads that are waiting to enter).

Since Java uses the Hansen model, it's critical for correctness that a "***while***" construct is used in your code that blocks:

```
….
synchronized (x) {
      while( ! some_good_condition) {
          x.wait();
      }
     // Ok – "some_good_condition" is now true
}
```

This is because the just-notified waiter isn't guaranteed to be the next Thread that gets control of the monitor. So, even though the interesting condition likely became true just prior to someone calling ***notify(),*** it may no longer be true by the time the waiter runs again.

Java's ***x.notifyAll()*** is the counterpart to Broadcast(). Like Broadcast, ***notifyAll()*** awakens all (if any) Threads that are waiting in the monitor for object x.

You'll often see synchronized instance methods coded on the left, which can be considered functionally equivalent to the implementation on the right:

```
class T {
    boolean boiled = false;
    synchronized void eggs() {

          while (! boiled) {
             wait();
          }
          // OK it's boiled now

    }
}
```

```
class T {
  boolean boiled =  false;
  void eggs() {
     synchronized (this) {
        while (! boiled) {
          this.wait();
        }
        // OK it's boiled now
     }
  }
}
```

If you can mentally translate the version on the left to the version on the right, you'll be miles ahead of the game (and most other Java programmers.)