Hoare-Style Monitors for Java

Theodore S Norvell
Electrical and Computer Engineering
Memorial University

February 17, 2006

1 Hoare-Style Monitors

Coordinating the interactions of two or more threads can be very tricky and can be a source of latent defects in multithreaded code.

In the early seventies Edsger Dijkstra, Per Brinch-Hansen, and C.A.R. Hoare developed the idea of a *monitor*, an object that would protect data by enforcing single threaded access — only one thread would be allowed to execute code within the monitor at one time. A monitor would be an island of single threaded sanity in a turbulent sea of multithreadedness. If threads could confine their interactions to the monitors then there would be a much better hope that they wouldn't interfere with each other in unanticipated ways. A number of variants on the idea of monitors have been proposed and implemented in various programming languages, such as Concurrent Pascal, Mesa, Concurrent Euclid, Turing, and more recently Java. The idea of single-threaded access to a monitor is straightforward and the same in all these variants; where they differ is in how threads wait for the state of the monitor to change, and how they communicate to each other that the state has changed. Of all these variants, I've always found Hoare's version the easiest to use; the reason is that, as we will see later, in Hoare's version occupancy of the monitor can be transferred directly from a signalling to a signalled thread.

The rest of this article presents a Java package that implements Hoare-style monitors.

2 Exclusive Access

A monitor is an object that enforces *exclusive access* to its data, that is, access is by only one thread at a time. In my implementation this is accomplished by inheriting from class AbstractMonitor and invoking inherited method enter() to enter the monitor and inherited method leave() to leave. Between invoking enter() and leave() the thread is said to *occupy* the monitor, except while it is calling signal() or await() which we'll see later. At most one thread can occupy the monitor at any time; a thread that invokes enter() while the monitor is

occupied will be delayed until the monitor is unoccupied. An example of using enter() and leave() is shown in Example 1. Without exclusive access, a client might find that the time is 25:60:60, or the time might be reported as 12:00:00, when it is really 13:00:00. This example could equally well be implemented using Java's **synchronized** keyword. The only advantage to using my monitor package for this example is that the class invariant, embodied as the overridden method invariant() is checked as part of the entering and leaving process. If the invariant is found to be false when a monitor is unoccupied an exception will be thrown. In such a simple class, checking the class invariant is probably not that useful. However in more complex classes, checks of class invariants can provided a valuable indication of a bug during testing.

```
import monitor.* ;

public class TimeOfDay extends AbstractMonitor{
    private volatile int hr = 0, min = 0, sec = 0 ;

    public void tick() {
        enter() ;
        sec += 1 ; min += sec/60 ; hr += min/60 ;
        sec = sec % 60 ; min = sec % 60 ; hr = hr % 24 ; }
        leave() ; }

    public void get( int[] time ) {
        enter() ;
        time[0] = sec ; time[1] = min ; time[2] = hr ;
        leave() ; }

    @Override
    protected boolean invariant() {
        return 0 <= sec && sec < 60
            && 0 <= min && min < 60
            && 0 <= hr && hr < 24 ; }
}
```

example 1: A class with exclusive access

3 Conditions

The real power of Hoare-style monitors comes when a thread may have to wait for some condition to arise. A classic example is a bounded FIFO (First In, First Out) queue. A thread that is fetching from the queue may have to wait

until the queue is not empty; a thread that is depositing data into the queue, must wait until there is data in the queue.

In the FIFO queue (see Example **??**) the number of objects present in the queue is count. A fetching thread may have to wait until count $>$ 0 while a depositing thread may have to wait until count $<$ capacity. Each of these assertions about the state is embodied in a *condition* object. The two condition objects are contained in two private fields initialized using the makeCondition() method inherited from AbstractMonitor.

```
/** Signalled when count < capacity */
private Condition notFull = makeCondition() ;
/** Signalled when count > 0 */
private Condition notEmpty = makeCondition() ;
```

When a thread needs to wait for an assertion to become true, it invokes await() on the appropriate condition object. While the thread is waiting, it does not occupy the monitor, and so another thread can enter and make the condition come true. Since the thread that invokes await() leaves the monitor unoccupied, the class invariant should be true when await() is invoked, and indeed await() checks the invariant.

A thread that makes the assertion associated with a condition object true may (and usually should) transfer occupancy to a waiting thread. It can do this by invoking the method signal() on the condition object. If there are one or more threads waiting on that condition variable, one waiting thread is selected and it enters the monitor, while the signalling thread temporarily leaves, in order to ensure exclusive access. An important point is that there is no point in time, during this exchange, when the monitor is unoccupied, so the waiting thread will find the monitor in the same state as it was when the signalling thread called signal().

If we follow the convention that the monitor will only be in certain states when signal() is called, then it will only be in those same certain states when await() returns. This is sort of a contract that the implementor of the monitor makes with his or herself. Indeed signal() and await() are best understood in terms of design-by-contract, or preconditions and postconditions.

The precondition of await(), as we've seen, is the monitor's invariant, which I'll call I. Since the thread only returns from await() when another thread calls signal(), and the other thread only calls signal when the assertion associated with the condition object is true, that assertion must be true when await() is returned from. I'll use P_c for the assertion associated with condition c. The contract for c.await() is

c.await()	
Require:	I
Ensure:	P_c

Since the signalling thread does not return from signal() until the monitor is empty, the invariant will be true on its return. As a precondition we need to ensure that P_c is true *if there are any waiting threads.* If there are no waiting

threads then the signalling thread simply leaves the monitor empty, in which case the invariant should be true. So how can we tell if there are any waiting threads? Luckily the condition objects have an accessor count() that tells the number of waiting threads. The contract for c.signal() is

	c.signal()
Require:	(c.count()==0 && I) || (c.count()!=0 && P_c)
Ensure:	I

Usually we just make sure that both I and P_c are true before calling c.signal(), this ensures that the precondition is true without worrying about the number of waiting threads.[1]

4 Assertion objects

So far the assertions associated with each condition object have existed only as abstractions, used to design the monitor. Unlike the invariant, I haven't yet given any way to check the them during testing. One way would be to use the static method check(boolean) of the Assertion class, used to check the precondition of the constructor in the FIFO queue in Example **??**, or Java's **assert** statement. However this would mean duplicating the assertion once for each call to await() (or signal()), as well as documenting it at the point of declaration. Furthermore it is common that calls to await() are conditional, so we end up writing something like this

```
if( ! P_c ) {
    c.await() ;
    Assertion.check( P_c ) ; }
```

For good reason software engineers abhor duplication of code. The solution is to create an object to represent the assertion. We extend Assertion while overriding the abstract method assertion().The assertion object is then used to construct the condition object. In the FIFO queue in Example **??** we can change the declarations of the condition objects to

```
private Condition notFull = makeCondition(
    new Assertion() {
        boolean assertion() { return count < capacity ; } } ) ;
private Condition notEmpty = makeCondition(
    new Assertion() {
        boolean assertion() { return count > 0 ; } } ) ;
```

[1] There's one more wrinkle to these contracts. They work if you assume that neither the invariant I nor the assertion P_c depend on the number of processes waiting on c. That's because the acts of starting to wait and ceasing to wait change these quantities. Dahl [[ref]] gives the details for the case where I and P_c may depend on the number of waiting processes.

The assertions have moved out of the comments and into the code. By putting the assertions in the code, they can be (and are) automatically checked upon return from await().

We can now replace the if-statements at the start of each fetch and deposit with conditional awaits:

```
notFull.conditionalAwait() ;
```

and

```
notEmpty.conditionalAwait() ;
```

The meaning of c.conditionalAwait() is simply

```
if( ! P_c ) c.await() ;
```

Similarly c.conditionalSignal() abbreviates

```
if( P_c ) c.signal() ;
```

5 Comparison to Java's Standard Library

It wouldn't be right to such a package without comparing to Java's 'native' monitors built using the **synchronized** keyword and the wait() and notify() methods from the Object class.

The first difference is that my package provides support for defensive programming through the invariant() method and the assertion objects associated with each condition object.

The second difference has to do with the use of condition objects. When threads might have to wait for various assertions to come true in various circumstances, it makes sense for them to wait in different ways. Using Object.wait() there is only one way to wait.

The third and most fundamental difference relates to how control is passed from the signalling thread to a waiting thread. In my package, following Hoare's suggestion, control is passed seamlessly from one thread to the other. This way the thread that awakes can, in a sense, be sure that the thing it has waited for is now true. The notify() routine simply moves a waiting thread back to the entry queue for the monitor. By the time the waiting thread actually becomes the occupant the assertion for which it was waiting may no longer be true. Thus the call to wait() should always be in a loop that rechecks the assertion:

```
while( ! P_c ) { wait() ; }
```

The invocation of notify() that awakes a thread that then re-waits is effectively lost. With my package, sufficient signalling requires that each await() be matched by some signal() in the future. Using Java's routines directly notify() calls can be lost and so there is no way to ensure sufficient signalling.