Technical Note TN1153

Thread-Safe Toolbox Access From MRJ

CONTENTS

Java is pervasively multi-threaded. The Mac OS isn't. Most of it isn't re-entrant, and parts of it are very dependent on global state that needs to remain consistent from one call to the next. This can cause big problems when trying to call the Mac OS directly from Java. This technote describes synchronization techniques that will allow your native or JDirect code to play safely when making OS or Toolbox calls, especially as we move forward to Mac OS X.

Updated: [Mar 20 1999]

How We Play Safely With the Toolbox

As we all know, the basics of the Mac OS and Toolbox (which for simplicity I'll just lump together as the "Toolbox" here) were designed in the early 1980's for a machine that could run only a single app at a time and a single thread of execution in that app. Things have been improved a bit since that time, but the constraints of that basic design require that the Toolbox is still only single-threaded. Cooperative multitasking between applications means process switching occurs only at well known times (when `WaitNextEvent` is called) and requires swapping a whole mess of low-memory global state.

This causes problems when trying to run multiple threads in a Mac OS app. Toolbox calls are not re-entrant, which means that while one thread is executing a Toolbox call, no other thread can enter the Toolbox. Furthermore, the pervasive use of global state (like the current `GrafPort` or the current resource chain) means that a thread that needs to make a sequence of calls that rely on that state needs to prevent other threads from using or changing that state.

We had to deal with these issues in implementing the AWT for MRJ 2.1, since most of the AWT is written in Java code that calls the Toolbox via JDirect2. We used a pretty standard solution of using "critical sections" in the code wherever the Toolbox is used, with only one thread able to enter such a critical section at a time.

In Java terms, this is implemented by having a single global object (accessed via a public static variable) serve as a synchronization lock, and putting all critical sections into Java blocks synchronized to that object. This object is known as:

```
    com.apple.mrj.macos.toolbox.Toolbox.LOCK
```

(In other words, it's a static final variable called `LOCK` in the class `Toolbox` in the package `com.apple.mrj.macos.toolbox`.) Usage of this, presuming the appropriate import directive, looks like:

```
    synchronized( Toolbox.LOCK ) {
         ...
         }
```

We on the MRJ team call this "LOCKing" or "using the LOCK", capitalized as shown. (When saying it, you emphasize the first syllable to indicate that it's capitalized...)

Why You Should Care

We implemented this so that our own AWT would work correctly. But if other developers are going to write code that uses JDirect -- and many of you are -- and if that code is going to run in apps that also use AWT, then we need to use the same synchronization technique so that your code doesn't step on our code and vice versa. Thus, this technote.

Example

Here's a simple example showing a fully synchronized `SysBeep` call:

```
    import com.apple.mrj.macos.toolbox.Toolbox;
    import SoundFunctions;    // From JDirect Sample Code from SDK

    ...

    public void playABeep( ) {
        synchronized(Toolbox.LOCK) {
            SoundFunctions.SysBeep(1);
        }
    }
```

Native-code (JNI)

This isn't an issue specific to JDirect. If you write native methods (probably using JNI) that call the Toolbox, the same issues can arise. If your native code alters Toolbox state like the current port or the state inside a `GrafPort`, you should synchronize it against the LOCK. The easiest way to do this is to leave the native code alone and ensure that you LOCK every call to the native method. However, you can also use the JNI API to locate the LOCK object and acquire/release its monitor. Just make absolutely sure that your native method cannot exit without releasing the monitor, or you will cause MRJ to hang!

Back to top

Paranoia and Reality

The previous section is actually a bit alarmist ... for now. The truth is that it is *currently* not necessary to LOCK around every single Toolbox call -- in fact, individual calls that don't require or change system state -- like the `SysBeep` above -- don't currently need any synchronization. This is because MRJ 2.1 threads aren't really pre-emptive -- they just pretend to be, by time-slicing Java code. They will not preempt while running native code, including the Toolbox. Any individual Toolbox call will run as long as it wants without any other Java threads getting time, so there's no danger of another thread re-entrantly calling the Toolbox.

Note carefully that I said *currently* . The current state of the art is due to the current limitations of how you can implement threads on the Mac OS, which has a lot to do with the nanokernel at the heart of the system. Future releases of Mac OS 8 ("Blue") will have improvements to the nanokernel that might at some future time allow us to support true pre-emption. Moreover, Mac OS X will be based on the Mach 3 microkernel which supports true pre-emptive threads, which Java will use.

This will requires changes to any native method implementations as well (which is not surprising, since all MacOS code will need to be revised to run with the Carbon APIs in OS X.) In OS X it will become possible for a native method to be preempted by another thread, which makes synchronization a lot more important. You'll need to start acquiring/releasing the LOCK within your native methods just as you would in the equivalent Java code. We'll be providing more details of what kind of synchronization is necessary, as Mac OS X becomes more of a reality.

Therefore, if you want your JDirect- or JNI-based Toolbox calls to continue to function in future releases of the Mac OS with future Java implementations, you should start LOCKing all your Toolbox calls right away. If you don't have time to do it religiously for everything, you at least need to LOCK groups of Toolbox calls that need an undisturbed global environment.

Back to top

The Perils of LOCKing

Unfortunately, all this synchronization comes with dangers of its own, as anyone who's done much multithreaded programming can tell you. The principal one is the classic problem of *deadlock* .

A short tutorial on deadlock

Deadlock results when a thread holding a monitor (it's inside a `synchronized` block, in Java parlance) tries to acquire another monitor which is being held by another thread, which is already blocked trying to acquire the monitor the first thread is holding. Or, in human terms, "I won't give you the can until you give me the can-opener, but you won't give me the can-opener until I give you the can." We both starve; in Java both threads will be blocked forever.

The LOCK object is nothing special in this regard, but the fact that you're going to be synchronizing against it a lot means you have to watch out for deadlocks. Here's a typical scenario that causes a deadlock:

```
    public synchronized void foo( ) {
        System.out.println("In foo!");
        synchronized(Toolbox.LOCK) {
            SoundFunctions.SysBeep(1);
        }
    }

    public void bar( ) {
        synchronized(Toolbox.LOCK) {
            SoundFunctions.SysBeep(1);
            foo();
        }
    }
```

This may look innocuous, but consider a thread that calls foo, gets as far as the `println`, then is preempted. Control switches to a second thread that calls the bar method on the same object. The second thread proceeds through the bar method, calls foo, and is now blocked because the first thread is already holding the monitor for that object (because foo is synchronized).

Later, the first thread wakes up and tries to enter the foo method's `synchronized` statement. Unfortunately the second thread is already holding the LOCK, so the first thread also blocks.

Now both threads are blocked, and neither can proceed until the other one releases a monitor. Permanent deadlock.

How to avoid this

One of the textbook solutions for avoiding deadlock is to always acquire monitors in the same order. The reason for the deadlock in the above example is that the first thread acquired first the receiving object's lock and then the Toolbox LOCK, while the second thread acquired them in the opposite order.

The ordering we've used in our own code is that *the LOCK is always the last monitor acquired.* In Java terms, this means that, while synchronized to the LOCK, you should never synchronize on anything else or call any method that synchronizes on anything else. (In the example, the bar method violates this rule when it calls foo. The best fix is probably to move the foo call out of the synchronized block.)

A specific and very important corollary of this is that you should not call into the AWT while holding the LOCK, since the public AWT methods do a lot of synchronization, as do our private peer classes that they call.

If you feel very smart, you can make certain exceptions to this rule -- you can call synchronized methods from with a LOCKed block as long as you're certain that you never LOCK while synchronized against that object. For instance, you may feel safe calling methods on a Vector object within a LOCKed block, even though most Vector methods are synchronized, because it's pretty unlikely that you have other code that synchronizes against that Vector.

This leads to a programming style in which you wrap LOCKed blocks pretty tightly around your Toolbox calls. If you have a method that needs to call the Toolbox, do some other Java stuff, then call the Toolbox again, don't LOCK the entire method. Instead, LOCK the first and second group of Toolbox calls, leaving the stuff in the middle out. Of course, this implies that global state might have changed between the first and second group of calls. If this is unacceptable, you'll need to have the second group set the state up again, or figure out how to re-order the code so the two groups can be merged into one with the rest of the Java calls coming before or after.

Debugging deadlocks

We had so much fun with deadlocks while developing MRJ that we added some support to the MRJ VM to help us debug them.

First, the debug build of MRJLib includes a deadlock sniffer in its thread scheduler. This will detect the classic deadlock case described above, and will immediately drop into MacsBug with a user-break telling you that a deadlock was detected. You can then use the techniques below to get more info. (The debug build is supplied with the MRJ 2.1 SDK. See its accompanying Read-Me file for installation instructions and more info.)

Even with the regular optimized build of MRJLib, if you have the MRJ `'dcmd'` installed you can break into MacsBug and then use the `'mrj dl'` and `'mrj sync'` commands to see if there are currently any deadlocks or deadlock-like synchronization problems. These commands will print information about the threads and objects involved. (The `'dcmd'` is supplied with the MRJ 2.1 SDK; for more information about it and these specific commands, see its accompanying Read-Me file and the *Debugging Java Code With MacsBug* technote.)

Back to top

References

Technote 1154: *Debugging Java Code With MacsBug*

Using JDirect To Access MacOS Code From Java. A quick introduction to JDirect. In the "docs" folder of the MRJ 2.1 SDK.

Oaks, Scott and Wong, Henry. *Java Threads* . O'Reilly, 1997.
Another typically excellent O'Reilly book, serving as a strong introduction to multithreading and threads in Java. Very readable for beginners but has some good advice for more advanced users, too.

Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns* . Addison-Wesley, 1997. This is a very good, solid book that's nonetheless somewhat academic and may be overkill for many people. I've never managed to read all the way through it!

Back to top

Downloadables

📄	Acrobat version of this Note (K).	Download

Back to top