JAVAWORLD

UNDER THE HOOD

By Bill Venners, JavaWorld
JUL 1, 1997 1:00 AM PT

HOW-TO

How the Java virtual machine performs thread synchronization

Understanding threads, shared data, locks, and more in Java bytecode

All Java programs are compiled into class files, which contain bytecodes, the machine language of the Java virtual machine. This article takes a look at how thread synchronization is handled by the Java virtual machine, including the relevant bytecodes. *(1,750 words)*

This month's **Under The Hood** looks at thread synchronization in both the Java language and the Java virtual machine (JVM). This article is the last in the long series of bytecode articles I began last summer. It describes the only two opcodes directly related to thread synchronization, the opcodes used for entering and exiting monitors.

Threads and shared data

One of the strengths of the Java programming language is its support for multithreading at the language level. Much of this support centers on coordinating access to data shared among multiple threads.

The JVM organizes the data of a running Java application into several runtime data areas: one or more Java stacks, a heap, and a method area. For a backgrounder on these memory areas, see the first **Under the Hood** article: "The lean, mean virtual machine."

Inside the Java virtual machine, each thread is awarded a *Java stack*, which contains data no other thread can access, including the local variables, parameters, and return values of each method the thread has invoked. The data on the stack is limited to primitive types and object references. In the JVM, it is not possible to place the image of an actual object on the stack. All objects reside on the heap.

There is only one *heap* inside the JVM, and all threads share it. The heap contains nothing but objects. There is no way to place a solitary primitive type or object reference on the heap -- these things must be part of an object. Arrays reside on the heap, including arrays of primitive types, but in Java, arrays are objects too.

Besides the Java stack and the heap, the other place data may reside in the JVM is the *method area*, which contains all the class (or static) variables used by the program. The method area is similar to the stack in that it contains only primitive types and object references. Unlike the stack, however, the class variables in the method area are shared by all threads.

Object and class locks

As described above, two memory areas in the Java virtual machine contain data shared by all threads. These are:

- The heap, which contains all objects
- The method area, which contains all class variables

If multiple threads need to use the same objects or class variables concurrently, their access to the data must be properly managed. Otherwise, the program will have unpredictable behavior.

To coordinate shared data access among multiple threads, the Java virtual machine associates a *lock* with each object and class. A lock is like a privilege that only one thread can "possess" at any one time. If a thread wants to lock a particular object or class, it asks the JVM. At some point after the thread asks the JVM for a lock -- maybe very soon, maybe later, possibly never -- the JVM gives the lock to the thread. When the thread no longer needs the lock, it returns it to the JVM. If another thread has requested the same lock, the JVM passes the lock to that thread.

Class locks are actually implemented as object locks. When the JVM loads a class file, it creates an instance of class `java.lang.Class`. When you lock a class, you are actually locking that class's `Class` object.

Threads need not obtain a lock to access instance or class variables. If a thread does obtain a lock, however, no other thread can access the locked data until the thread that owns the lock releases it.

Monitors

The JVM uses locks in conjunction with *monitors*. A monitor is basically a guardian in that it watches over a sequence of code, making sure only one thread at a time executes the code.

Each monitor is associated with an object reference. When a thread arrives at the first instruction in a block of code that is under the watchful eye of a monitor, the thread must obtain a lock on the referenced object. The thread is not allowed to execute the code until it obtains the lock. Once it has obtained the lock, the thread enters the block of protected code.

When the thread leaves the block, no matter how it leaves the block, it releases the lock on the associated object.

Multiple locks

A single thread is allowed to lock the same object multiple times. For each object, the JVM maintains a count of the number of times the object has been locked. An unlocked object has a count of zero. When a thread acquires the lock for the first time, the count is incremented to one. Each time the thread acquires a lock on the same object, a count is incremented. Each time the thread releases the lock, the count is decremented. When the count reaches zero, the lock is released and made

available to other threads.

Synchronized blocks

In Java language terminology, the coordination of multiple threads that must access shared data is called *synchronization*. The language provides two built-in ways to synchronize access to data: with synchronized statements or synchronized methods.

Synchronized statements

To create a synchronized statement, you use the `synchronized` keyword with an expression that evaluates to an object reference, as in the `reverseOrder()` method below:

```
class KitchenSync {
    private int[] intArray = new int[10];
    void reverseOrder() {
        synchronized (this) {
            int halfWay = intArray.length / 2;
            for (int i = 0; i < halfWay; ++i) {
                int upperIndex = intArray.length - 1 - i;
                int save = intArray[upperIndex];
                intArray[upperIndex] = intArray[i];
                intArray[i] = save;
            }
        }
    }
}
```

In the case above, the statements contained within the synchronized block will not be executed until a lock is acquired on the current object (`this`). If instead of a `this` reference, the expression yielded a reference to another object, the lock associated with that object would be acquired before the thread continued.

Two opcodes, `monitorenter` and `monitorexit`, are used for synchronization blocks within methods, as shown in the table below.

Table 1. Monitors

Opcode	Operand(s)	Description
`monitorenter`	none	pop objectref, acquire the lock associated with objectref
`monitorexit`	none	pop objectref, release the lock associated with objectref

When `monitorenter` is encountered by the Java virtual machine, it acquires the lock for the object referred to by objectref on the stack. If the thread already owns the lock for that object, a count is incremented. Each time `monitorexit` is executed for the thread on the object, the count is decremented. When the count reaches zero, the monitor is released.

Take a look at the bytecode sequence generated by the `reverseOrder()` method of the `KitchenSync` class.

Note that a catch clause ensures the locked object will be unlocked even if an exception is thrown from within the synchronized block. No matter how the synchronized block is exited, the object lock acquired when the thread entered the block definitely will be released.

Synchronized methods

To synchronize an entire method, you just include the `synchronized` keyword as one of the method qualifiers, as in:

```
class HeatSync {
    private int[] intArray = new int[10];
    synchronized void reverseOrder() {
        int halfWay = intArray.length / 2;
        for (int i = 0; i < halfWay; ++i) {
            int upperIndex = intArray.length - 1 - i;
            int save = intArray[upperIndex];
            intArray[upperIndex] = intArray[i];
            intArray[i] = save;
        }
    }
}
```

The JVM does not use any special opcodes to invoke or return from synchronized methods. When the JVM resolves the symbolic reference to a method, it determines whether the method is synchronized. If it is, the JVM acquires a lock before invoking the method. For an instance method, the JVM acquires the lock associated with the object upon which the method is being invoked. For a class method, it acquires the lock associated with the class to which the method belongs. After a synchronized method completes, whether it completes by returning or by throwing an exception, the lock is released.

Coming next month

Now that I have gone through the entire bytecode instruction set, I will be broadening the scope of this column to include various aspects or applications of Java technology, not just the Java virtual machine. Next month, I'll begin a multi-part series that gives an in-depth overview of Java's security model.

Bill Venners has been writing software professionally for 12 years. Based in Silicon Valley, he provides software consulting and training services under the name Artima Software Company. Over the years he has developed software for the consumer electronics, education, semiconductor, and life insurance industries. He has programmed in many languages on many platforms: assembly language on various microprocessors, C on Unix, C++ on Windows, Java on the Web. He is author of the book: Inside the Java Virtual Machine, published by McGraw-Hill.

Learn more about this topic

- The book
  The Java virtual machine Specification (http://www.aw.com/cp/lindholm-yellin.html) (http://www.aw.com/cp/lindholm-yellin.html), by Tim Lindholm and Frank Yellin (ISBN 0-201-63452-X), part of The Java Series (http://www.aw.com/cp/javaseries.html) (http://www.aw.com/cp/javaseries.html), from Addison-Wesley, is the definitive Java virtual machine reference.

- **Previous "Under The Hood" articles:**

- "The Lean, Mean Virtual Machine (http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html)" Gives an introduction to the Java virtual machine.
- "The Java Class File Lifestyle (http://www.javaworld.com/javaworld/jw-07-1996/jw-07-classfile.html)" Gives an overview to the Java class file, the file format into which all Java programs are compiled.
- "Java's Garbage-Collected Heap (http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html)" Gives an overview of garbage collection in general and the garbage-collected heap of the Java virtual machine in particular.
- "Bytecode Basics (http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html)" Introduces the bytecodes of the Java virtual machine, and discusses primitive types, conversion operations, and stack operations in particular.
- "Floating Point Arithmetic (http://www.javaworld.com/javaworld/jw-10-1996/jw-10-hood.html)" Describes the Java virtual machine's floating-point support and the bytecodes that perform floating point operations.
- "Logic and Arithmetic (http://www.javaworld.com/javaworld/jw-11-1996/jw-11-hood.html)" Describes the Java virtual machine's support for logical and integer arithmetic, and the related bytecodes.
- "Objects and Arrays (http://www.javaworld.com/javaworld/jw-12-1996/jw-12-hood.html)" Describes how the Java virtual machine deals with objects and arrays, and discusses the relevant bytecodes.
- "Exceptions (http://www.javaworld.com/javaworld/jw-01-1997/jw-01-hood.html)" Describes how the Java virtual machine deals with exceptions, and discusses the relevant bytecodes.
- "Try-Finally (http://www.javaworld.com/javaworld/jw-02-1997/jw-02-hood.html)" Describes how the Java virtual machine implements try-finally clauses, and discusses the relevant bytecodes.
- "Control Flow (http://www.javaworld.com/javaworld/jw-03-1997/jw-03-hood.html)" Describes how the Java virtual machine implements control flow and discusses the relevant bytecodes.
- "The Architecture of Aglets (http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html)" Describes the inner workings of Aglets, IBM's autonomous Java-based software agent technology.
- "The Point of Aglets (http://www.javaworld.com/javaworld/jw-05-1997/jw-05-hood.html)" Analyzes the real-world utility of mobile agents such as Aglets, IBM's autonomous Java-based software agent technology.
- "Method Invocation and Return (http://www.javaworld.com/javaworld/jw-06-1997/jw-06-hood.html)" Explains how the Java virtual machine invokes and returns from methods, including the relevant bytecodes.