

Java Virtual Machine Locks

SS 2008

Synchronized

Gerald SCHARITZER (e0127228)

Table of Contents

1 Scope.....	3
1.1 Constraints.....	3
1.2 In Scope.....	3
1.3 Out of Scope.....	3
1.4 Limits.....	3
2 Logical View.....	4
2.1 Analysis.....	4
2.1.1 Synchronized Methods.....	5
2.1.2 Synchronized Statements.....	6
2.1.3 javap – the class file disassembler.....	6
2.1.4 java.lang.Thread.....	6
2.1.5 joprt.RtThread.....	6
2.1.6 com.jopdesign.sys.RtThreadImpl.....	7
2.2 Design.....	7
2.2.1 Object and Array Reference.....	7
2.2.2 Locks.....	7
2.2.3 Priority Ceiling Emulation.....	8
2.2.4 new.....	8
2.2.5 newarray.....	8
2.2.6 anewarray.....	8
2.2.7 multianewarray.....	8
2.2.8 monitorenter.....	8
2.2.9 monitorexit.....	9
3 Process View.....	10
4 Development View.....	11
4.1 Tools.....	11
4.2 Java Source Code.....	11
4.2.1 com.jopdesign.sys.GC.....	11
4.2.2 com.jopdesign.sys.JVM.....	11
4.2.3 com.jopdesign.sys.PriorityCeilingEmulation.....	11
4.2.4 com.jopdesign.sys.RtThreadImpl.....	11
4.3 JVM Source Code.....	11
4.4 Java Simulation.....	11
5 Physical View.....	12
6 Issues.....	13
7 Acronyms and Abbreviations.....	14
8 References.....	15

1 SCOPE

1.1 Constraints

The JVM is running on a uniprocessor machine.

Threads do not use local copies of fields. Therefore all fields are inherently volatile.

1.2 In Scope

Synchronized methods and synchronized statements are executed without disabling the scheduling interrupt.

1.3 Out of Scope

There is no special treatment for volatile fields, since threads do not make local copies.

1.4 Limits

The lock for every object consists of 4 unsigned 8 bit integers and stores the lock count, thread id, lock priority and thread priority. Therefore all these values must be in the range of 0 to 255.

maximum number of nested locks on one object	255
maximum number of threads	256
maximum priority	255
minimum priority	0

2 LOGICAL VIEW

2.1 Analysis

1 [JLS] 4.3.1 Objects

Each object has an associated lock, which is used by synchronized methods and the synchronized statement to provide control over concurrent access to state by multiple threads.

2 [JLS] 4.3.2 The Class Object

A class method that is declared synchronized (§8.4.3.6) synchronizes on the lock associated with the Class object of the class.

3 [JLS] 8 Classes

A synchronized method automatically locks an object before executing its body and automatically unlocks the object on return, as if by use of a synchronized statement, thus allowing its activities to be synchronized with those of other threads.

4 [JLS] 11 Exceptions

The exception mechanism of the Java platform is integrated with its synchronization model, so that locks are released as synchronized statements and invocations of synchronized methods complete abruptly.

5 [JLS] 15.12.4.5 Create Frame, Synchronize, Transfer Control

If the method *m* is synchronized, then an object must be locked before the transfer of control. No further progress can be made until the current thread can obtain the lock. If there is a target reference, then the target must be locked; otherwise the Class object for class *S*, the class of the method *m*, must be locked. Control is then transferred to the body of the method *m* to be invoked. The object is automatically unlocked when execution of the body of the method has completed, whether normally or abruptly. The locking and unlocking behaviour is exactly as if the body of the method were embedded in a synchronized statement.

6 [JLS] 17.1 Locks

The Java programming language provides multiple mechanisms for communicating between threads. The most basic of these methods is synchronization, which is implemented using monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor. A thread *t* may lock a particular monitor multiple times; each unlock reverses the effect of one lock operation.

The synchronized statement computes a reference to an object; it then attempts to perform a lock action on that object's monitor and does not proceed further until the lock action has

successfully completed. After the lock action has been performed, the body of the synchronized statement is executed. If execution of the body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

A synchronized method automatically performs a lock action when it is invoked; its body is not executed until the lock action has successfully completed. If the method is an instance method, it locks the monitor associated with the instance for which it was invoked (that is, the object that will be known as `this` during execution of the body of the method). If the method is static, it locks the monitor associated with the `Class` object that represents the class in which the method is defined. If execution of the method's body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

The Java programming language neither prevents nor requires detection of deadlock conditions. Programs where threads hold (directly or indirectly) locks on multiple objects should use conventional techniques for deadlock avoidance, creating higher-level locking primitives that don't deadlock, if necessary.

Other mechanisms, such as reads and writes of volatile variables and classes provided in the `java.util.concurrent` package, provide alternative ways of synchronization.

2.1.1 Synchronized Methods

Synchronized methods acquire or reenter their associated lock during the execution of the invocation instructions.

- `invokeinterface`
- `invokespecial`
- `invokestatic`
- `invokevirtual`

Locks are released by synchronized methods during the execution of return instructions.

- `areturn`
- `dreturn`
- `freturn`
- `ireturn`
- `lreturn`
- `return`

The JOP implicitly executes the same `monitorenter` and `monitorexit` instructions for `invoke` and `return` instructions. Therefore it is sufficient to implement acquiring a lock in `monitorenter` and releasing a lock in `monitorexit`. This will increase the execution time of the above instructions.

7 [JLS] 8.4.3.6 Synchronized Methods

A synchronized method acquires a monitor before it executes. For a class (static) method, the monitor associated with the `Class` object for the method's class is used. For an instance method, the monitor associated with `this` (the object for which the method was invoked) is used. These are the same locks that can be used by the synchronized statement.

2.1.2 Synchronized Statements

Synchronized statements are opened with the `monitorenter` instruction and closed with the `monitorexit` instruction. Both instructions consume an object reference from the stack, which specifies the lock to be obtained or released.

8 [JLS] 14.19 The Synchronized Statement

A synchronized statement acquires a mutual-exclusion lock on behalf of the executing thread, executes a block, then releases the lock. While the executing thread owns the lock, no other thread may acquire the lock.

A synchronized statement is executed by first evaluating the Expression.

If evaluation of the Expression completes abruptly for some reason, then the synchronized statement completes abruptly for the same reason.

Otherwise, if the value of the Expression is null, a `NullPointerException` is thrown.

Otherwise, let the non-null value of the Expression be *V*. The executing thread locks the lock associated with *V*. Then the Block is executed. If execution of the Block completes normally, then the lock is unlocked and the synchronized statement completes normally. If execution of the Block completes abruptly for any reason, then the lock is unlocked and the synchronized statement then completes abruptly for the same reason.

Acquiring the lock associated with an object does not of itself prevent other threads from accessing fields of the object or invoking unsynchronized methods on the object. Other threads can also use synchronized methods or the synchronized statement in a conventional manner to achieve mutual exclusion.

The locks acquired by synchronized statements are the same as the locks that are acquired implicitly by synchronized methods; see §8.4.3.6. A single thread may hold a lock more than once.

2.1.3 javap – the class file disassembler

- l Prints out line and local variable tables.
- private Shows all classes and members.
- s Prints internal type signatures.
- c Prints out disassembled code, i.e., the instructions that comprise the Java bytecodes, for each of the methods in the class. These are documented in the Java Virtual Machine Specification.
- verbose Prints stack size, number of locals and args for methods.

2.1.4 java.lang.Thread

The class `java.lang.Thread` only implements the `sleep()` method, which invokes `java.lang.Thread.sleepMs()`.

2.1.5 joprt.RtThread

The class `joprt.RtThread` uses `com.jopdesign.sys.RtThreadImpl`.

2.1.6 com.jopdesign.sys.RtThreadImpl

The threads are managed in a linked list, which is ordered by descending priority. The static field "head" points to the thread with the lowest priority. The field "lower" points to the next lower priority thread. The field "priority" stores the priority of the thread.

2.2 Design

2.2.1 Object and Array Reference

The object reference points to the first instance variable, while the array reference points to the first element. The current size of a reference is 8 x 4 bytes and the last 4 bytes are free for the lock.

byte offset	type	description
0	ptr	object heap address
4	ptr int32	method table address array length
8	uint32	size
12	uint32	type
16	ptr	next handle
20	ptr	gray list
24	uint32	space marker
28	uint32	lock

2.2.2 Locks

The lock is a 32 bit unsigned integer and is part of the object or array. The byte offset is relative to the object or array reference.

byte offset	type	
28	uint32	lock
28	uint8	thread priority
29	uint8	lock priority
30	uint8	thread id
31	uint8	lock count

2.2.2.1 Lock Count

The lock count is the number of times this lock has been acquired without having been released yet. Therefore an unlocked object has a lock count = 0 and a locked object has a lock

count > 0. The lock count is incremented by 1 when the lock is acquired and decremented by 1 when the lock is released. The priority of the thread is only raised to the priority of the lock, when the lock count is changed from 0 to 1 and only restored to its previous value, when the lock count is changed from 1 to 0.

2.2.2.2 Thread ID

If lock count > 0 then thread id specifies the thread, which holds the lock. Only the thread already holding the lock may increase the lock count beyond 1.

2.2.2.3 Lock Priority

The lock priority is the ceiling value for the priority ceiling emulation protocol. The default value is the maximum lock priority.

2.2.2.4 Thread Priority

The priority of the thread is stored in this field, when the thread acquires the lock and raises the lock count from 0 to 1. When the thread releases the lock and decreases the lock count from 1 to 0, then the priority of the thread is reset to the value of this field.

2.2.3 Priority Ceiling Emulation

Locks shall be acquired and released according to PCE as defined in [PCE]. Therefore this method must employ a native method to set the lock priority.

2.2.4 new

Initialise the lock in GC.newObject() with maximum priority.

2.2.5 newarray

Initialise the lock in GC.newArray() with maximum priority.

2.2.6 anewarray

Initialise the lock in GC.newArray() with maximum priority.

2.2.7 multianewarray

Initialise the lock in GC.newArray() with maximum priority.

2.2.8 monitorenter

The interrupts may not be turned off anymore.

9 [JVM] 6.M monitorenter

<p>Each object has a monitor associated with it. The thread that executes monitorenter gains ownership of the monitor associated with objectref. If another thread already owns the monitor associated with objectref, the current thread waits until the object is unlocked, then tries again to gain ownership. If the current thread already owns the monitor associated with objectref, it increments a counter in the monitor indicating the number of times this thread has entered the monitor. If the monitor associated with objectref is not owned by any thread,</p>

the current thread becomes the owner of the monitor, setting the entry count of this monitor to 1.

2.2.9 monitorexit

The interrupts may not be turned on anymore.

10[JVM] 6.M monitorexit

The current thread should be the owner of the monitor associated with the instance referenced by objectref. The thread decrements the counter indicating the number of times it has entered this monitor. If as a result the value of the counter becomes zero, the current thread releases the monitor. If the monitor associated with objectref becomes free, other threads that are waiting to acquire that monitor are allowed to attempt to do so.

3 PROCESS VIEW

Monitorenter and monitorexit still disable interrupts for a few statements, because they contain critical sections, which read and write shared resources like locks and threads.

While the mission is not started, monitorenter still simply disables interrupts, because the thread ids are defined during startMission().

4 DEVELOPMENT VIEW

4.1 Tools

Development Environment	Eclipse 3.3.2 http://www.eclipse.org/
Programming Language	Java 6 http://java.sun.com/
Unit Tests	JUnit 4
Build Tool	Ant 1.7
Version Control	Subclipse 1.2.4 http://subclipse.tigris.org/

4.2 Java Source Code

The Java implementation of `monitorenter` and `monitorexit` is found in `JOP_HOME/java/target/src/common/com/jopdesign/sys/JVM.java`.

4.2.1 `com.jopdesign.sys.GC`

The methods `newObject()` and `newArray()` initialise the lock for new objects and arrays.

4.2.2 `com.jopdesign.sys.JVM`

The methods for `monitorenter` and `monitorexit` implement the priority ceiling emulation.

4.2.3 `com.jopdesign.sys.PriorityCeilingEmulation`

This class provides methods to read and modify object locks.

4.2.4 `com.jopdesign.sys.RtThreadImpl`

The priority of a thread may be changed, which results in reordering the priority list.

4.3 JVM Source Code

The JVM implementation of `monitorenter` and `monitorexit` is found in `JOP_HOME/asm/src/jvm.asm`. These two byte codes are modified, such that they invoke the corresponding methods in `com.jopdesign.sys.JVM`.

4.4 Java Simulation

`JopSim.writeMem()` enables and disables interrupts, when write operations are performed on the interrupt enable address. Writes to the software interrupt address are respected such that an interrupt is simulated. `Monitorenter` and `monitorexit` are executed via the JVM class rather than in `JopSim`.

5 PHYSICAL VIEW

The JVM is running on a JOP, which is a uniprocessor and therefore executes at most one single thread at any point in time.

The scheduling interrupt is processed on byte code boundaries. Therefore acquiring and releasing a lock are atomic instructions.

6 ISSUES

If the priority of the thread is stored in the lock, then changes to the priority of the thread get lost when releasing the lock.

The lock and thread priorities are not tested, whether they are disjunctive.

The number of threads is not checked.

Exceeding the limit of 255 nested locks on one object throws an Error.

7 ACRONYMS AND ABBREVIATIONS

JOP	Java Optimized Processor
JVM	Java Virtual Machine
PCE	Priority Ceiling Emulation

8 REFERENCES

- [JAVAP] javap - The Java Class File Disassembler
<http://java.sun.com/javase/6/docs/technotes/tools/windows/javap.html>

- [JLS] Java Language Specification 3rd Edition
<http://java.sun.com/docs/books/jls/index.html>

- [JOP] Java Optimized Processor
<http://jopdesign.com/>

- [JVM] Java Virtual Machine Specification 2nd Edition
<http://java.sun.com/docs/books/jvms/index.html>

- [PCE] Priority Ceiling Emulation
<http://www.rtsj.org/specjavadoc/javax/realtime/PriorityCeilingEmulation.html>

- [RTSJ] Real Time Specification for Java 1.0.2
http://www.rtsj.org/specjavadoc/book_index.html