The Java Memory Model

Jeremy Manson\textsuperscript{1}, William Pugh\textsuperscript{1}, and Sarita Adve\textsuperscript{2}

\textsuperscript{1}University of Maryland

\textsuperscript{2}University of Illinois at Urbana-Champaign

Presented by
John Fisher-Ogden
November 22, 2005
Outline

Introduction
  Motivations
  Definitions
  Problems Addressed

Java Memory Model
  Sequential Consistency
  Out of Thin Air
  Causality
  Well-Behaved Executions
  Optimizations

Conclusion
Overview

- Java memory model—revised as part of Java 5.0 (JSR-133).
- Guarantees sequential consistency for data-race free programs
- Requires that the behavior of incorrectly synchronized programs be bounded by a well-defined notion of causality.
Motivations

- Original Java Memory Model (JMM) not well specified and difficult to understand. Semantics of final fields and volatile unclear.
- Maintain safety and security guarantees in the face of incorrectly or incompletely synchronized programs.
- Balance flexibility for code transformations and optimizations with lucidity for programmers writing concurrent code.
Definitions

Memory Model

- Correspondence between each load instruction and the store instruction that supplies the value retrieved by the load.
- Interesting mainly for multi-threaded programs. Reorderings in single-threaded programs maintain “as if sequential” semantics.
- Partially determines legal JVM and compiler implementations.

Incorrectly Synchronized (Data Race)

- Thread A writes to a variable.
- Thread B reads that same variable.
- The write and read are not ordered by synchronization.
Happens-Before Order

- Transitive closure of program order and synchronizes-with order.
Synchronization

Atomicity

- Locking to obtain mutual exclusion.

Visibility

- Ensuring that changes to object fields made in one thread are seen in other threads.

Ordering

- Ensuring that you aren’t surprised by the order in which statements are executed.
Several serious problems existed in the old memory model.

- Difficult to understand \(\Rightarrow\) widely violated.
- Did not allow reorderings that took place in every JVM.
- Final fields could appear to change value without synchronization (default value \(\rightarrow\) final value).
- Allowed volatile writes to be reordered with nonvolatile reads and writes which was counter-intuitive for most developers.
Volatile Example

class VolatileExample {
    int x = 0;
    volatile boolean v = false;
    public void writer() {
        x = 42;
        v = true;
    }

    public void reader() {
        if (v == true) {
            //uses x - now guaranteed to see 42.
        }
    }
}

Jeremy Manson, William Pugh, and Sarita Adve
The Java Memory Model
// double-checked-locking - don’t do this!

private static Something instance = null;

public Something getInstance() {
    if (instance == null) {
        synchronized (this) {
            if (instance == null)
                instance = new Something();
        }
    }
    return instance;
}
Looks clever, but doesn’t work!

- Writes that initialize the Something object and the write to the instance field can be done or perceived out of order
- Thread could see non-null reference to instance but default values for fields of the Something object.

Make instance volatile

- Brief synchronization not very expensive anymore.
- Stronger volatile semantics increases cost of volatile almost to cost of synchronization.
Java Memory Model

Essentially provides two things:

- For data-race-free programs, guarantees sequential consistency.
- Requires the behavior of incorrectly synchronized programs be bounded by a well defined notion of causality.
Sequential Consistency

Each thread (CPU) executes instructions in order.

Each thread (CPU) sees **all** operations in **some** total order.

Initially, \( x == y == 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( r2 = x; )</td>
<td>3: ( r1 = y )</td>
</tr>
<tr>
<td>2: ( y = 1; )</td>
<td>4: ( x = 2 )</td>
</tr>
</tbody>
</table>

\( r2 == 2, r1 == 1 \) violates sequential consistency.
Out-of-Thin-Air Guarantees

Initially, \( x == y == 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r1 = x; )</td>
<td>( r2 = y; )</td>
</tr>
<tr>
<td>( y = r1; )</td>
<td>( x = r2; )</td>
</tr>
</tbody>
</table>

Incorrectly synchronized, but we want to disallow \( r1 == r2 == 42 \).

An Out Of Thin Air Result
Previous strategy of leaving semantics for incorrectly synchronized programs unspecified inconsistent with Java’s security and safety guarantees.

Thread 1 could speculatively write 42 to y, creating a logical chain that is self-justifying.

Security violation - create a reference “out-of-thin-ar” to an object that should not be accessible.
Causality

Need to incorporate causality into memory model to avoid circular reasoning.

Notion of “cause” tricky—cannot employ data and control dependence

<table>
<thead>
<tr>
<th>Before compiler transformation</th>
<th>After compiler transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially, (a = 0, b = 1)</td>
<td>Initially, (a = 0, b = 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: (r1 = a;)</td>
<td>5: (r3 = b;)</td>
</tr>
<tr>
<td>2: (r2 = a;)</td>
<td>6: (a = r3;)</td>
</tr>
<tr>
<td>3: if ((r1 == r2))</td>
<td></td>
</tr>
<tr>
<td>4: (b = 2;)</td>
<td></td>
</tr>
<tr>
<td>Is (r1 == r2 == r3 == 2) possible?</td>
<td>(r1 == r2 == r3 == 2) is sequentially consistent</td>
</tr>
</tbody>
</table>
Model builds a justified execution iteratively

- Using a sequentially consistent execution is too relaxed in some subtle cases.
- Well-behaved execution—a read that is not yet committed must return the value of a write that is ordered before it by happens-before.
Given a well-behaved execution:

- may commit any uncommitted writes that occur in it
- may commit any uncommitted reads that occur but require that the read return the value of a previously committed write in both the justifying execution and the execution being justified.

Occurrence of a committed action and its value does not depend on an uncommitted data race.
Formally prove legality of various reorderings and transformations:

- Synchronization on thread local objects can be removed
- Redundant nested synchronization can be removed
- Volatile fields of thread local objects can be treated as normal fields

Also prove if an execution of a correctly synchronized program is legal under the Java memory model, it is sequentially consistent.
Java Memory Model:

- Addressed problems with previous model
- Guarantees sequential consistency for correctly synchronized programs
- Bounds behavior of incorrectly synchronized programs by a well-defined notion of causality