Java Memory Model

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu
Context

- Memory consistency models
- Hardware memory models
- Language memory models
  —today: Java memory model
Motivation

• Why have a memory model for Java?
  — Java supports threads that shared memory
  — must have a memory model to define program semantics
    – determines the transformations the compiler can make
    – specifies ordering guarantees that a compiler must preserve regardless of the underlying architecture

• Why not sequential consistency?
  — precludes many optimizations important for performance
    – HW optimizations: store buffers, speculation, …
    – compiler optimizations: register allocation, common sub-expression elimination, loop interchange or blocking all have the effect of reordering or eliminating memory operations

‘Out-of-thin-air’ Problem

- Assume an incorrectly synchronized program
- After execution, could \( r1 == r2 == 42 \)?

<table>
<thead>
<tr>
<th>Initially, ( x == y == 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>( r1 = x; )</td>
</tr>
<tr>
<td>( y = r1; )</td>
</tr>
</tbody>
</table>

What if:

1. thread 1 speculatively writes 42 to \( y \)
2. thread 2 reads 42 for \( y \)
3. thread 2 writes 42 for \( x \)
4. thread 1 reads 42 for \( x \)
5. thread 1 validates its write speculation for \( y \)
Analysis of ‘Out-of-thin-air’ Problem

• Should we disallow this ‘optimization’?

• Why not let this error be undefined?

• Consider the Java class loader
  —cornerstone of the Java virtual machine
  —describes behavior of converting a named class into the bits responsible for implementing that class

• Suppose ‘42’ was &loadClass?
  —unintentional errors => violate safety
  —intentional errors => security risk
Outline

• Consider a classic Java synchronization example
  — lazy initialization using double checked locking

• Java Memory Model
  — sequential consistency and Data Races
  — guarantee no “out-of-thin-air” values
  — causality
  — examples of well-formed executions

• Compiler Transformations
  — valid transformations allowed by JMM
  — invalid transformations prohibited by JMM

• Summary
Lazy Initialization

class Foo {
    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }
}

Clearly is not thread safe
Ensuring Thread Safety?

Two things to consider

— **synchronization**
  - if used correctly, can provide mutual exclusion to shared data

— **data visibility**
  - writing a value to a variable from a thread doesn't mean it will be immediately visible in a different thread
Mechanisms in Java

• Synchronization
  — synchronized keyword for methods and blocks
    - permits one thread to enter at any given time
      reentrant: thread can call a synch method within a synch method
    - synchronized block specifies object providing the lock
  — explicit Lock: finer control

• Data Visibility
  — final variable
    - can only be initialized only once
      initializer or assignment statement
    - final modifier applied to a field or variable only determines the properties of the value, not the referenced object
      public final Point p;
      after p is assigned, p.x and p.y can be still be assigned
  — volatile variable
    - never cached: all reads and writes go straight to memory
    - a write to a volatile variable v synchronizes-with all subsequent reads of v by any thread
Approach 1: Synchronized Method

- Idea: guarantee thread safety by mutual exclusion using a synchronized method to control access to helper

```java
// extend to multithread -threaded version, add synchronized on method
class Foo {
    private Helper helper;
    public synchronized Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }
}
```

Critical section highlighted in blue.
Approach 2: Double-checked Locking (DCL)

- Idea: synchronize initialization, but not access
- Why? improve performance

one possible execution sequence

```
1 class Foo {
2     private Helper helper;
3     public Helper getHelper() {
4         if (helper == null) {
5             synchronized(this) {
6                 if (helper == null) {
7                     helper = new Helper();
8                 }
9             }
10         }
11         return helper;
12     }
13 }
```

it seems to work…
Approach 2: Double-checked Locking (DCL)

- Idea: synchronize initialization, but not access
- Why? improve performance

how about this sequence?

```java
class Foo {
    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
}
```

Problem:
compiler or hardware could reorder the writes initializing the fields in helper
some fields of the object might be initialized after the write to helper
volatile ensures that the actions that happen before the write to helper in the code must, when the program executes, actually happen before the write to helper.
Approach 3: DCL + volatile (Java 5)

- Local variable ‘result’ reduces access to volatile variable ‘helper’. After ‘helper’ has been initialized, (most of the time), the volatile field is only accessed once (due to "return result;" instead of "return helper")
- Can improve the method's overall performance by as much as 25 percent.

```java
class Foo {
    private volatile Helper helper;
    public Helper getHelper() {
        Helper result;
        if (result == null) {
            synchronized(this) {
                result = helper;
                if (result == null) {
                    helper = result
                        new Helper();
                }
            }
        }
        return result;
    }
}
```
Terminology

- **Data race**
  - two concurrent accesses to the same shared variable are said to be conflicting if at least one access is a write

- **Correctly synchronized**
  - a program is said to be correctly synchronized or data-race-free iff all sequentially consistent executions of the program are free of data races
Java Memory Model

• Goal
  — sufficiently easy to understand and use
  — permit important optimizations used by compilers and hardware

• Guarantees
  — “Well-Behaved” programs observe sequentially consistency
  — “Incorrect” programs
    – may contain data races
    – still, no out of thin air result
Implications for JVM Implementations

- Different compilers may employ different optimizations
  - common sub-expression elimination (CSE)
  - redundant read elimination
  - ...
- Different architectures
  - Power, X86-TSO, ARM, ...
- Program behaviors are still well defined by memory model
Happens-Before Memory Model

• Happens-before order: transitive closure of
  — program order
  — synchronizes-with order

• Happens-Before consistency
  — a read \( r \) is allow to see a write \( w \) to \( v \) if
    – \( r \) doesn’t happen before \( w \)
    – no other writes after \( w \) and before \( w \)
  — determines what a nonvolatile read can see

• Synchronization consistency
  — synchronization order is consistent with program order
  — each read \( r \) of a volatile variable \( v \) sees the last write to \( v \) that comes before it in the synchronization order
Initially, $x = 0$, $\text{ready} = \text{false}$. $\text{ready}$ is a volatile variable.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1$;</td>
<td>if $(\text{ready})$</td>
</tr>
<tr>
<td>$\text{ready} = \text{true}$</td>
<td>$r1 = x$;</td>
</tr>
<tr>
<td>If $r1 = x$; executes, it will read 1.</td>
<td></td>
</tr>
</tbody>
</table>
Happens-Before Isn’t Intuitive

- r1 == r2 == 0 is the only sequentially consistent result
- Happens-Before order allows r1 == r2 == 42
  — consider the case that both writes occur first

Correctly synchronized

x == y == 0 initially

Thread1   Thread2
1  r1 = x;   1  r2 = y;
2 if (r1 != 0) 2 if (r2 != 0)
3  y = 42;   3  x = 42

Unacceptable Behavior under JMM
Java Memory Model (JMM)

— want to keep the intuitive elements in the HBM

— need to guard against out of thin air results and allow transformations shown next
Redundant Read Elimination

a = 0, b = 1 initially

Thread1                                Thread2
1  r1 = a;                             1  b = 2;
2  r2 = a;                             2  r1 = a;
3  if (r1 == r2)                       3  r2 = r1;
4  b = 2;                             4  if (true);
5  r3 = b;                             6  a = r3;
6  a = r3;

r1 == r2 == r3 == 2 ?

This transformation is allowable by JMM
early write is allowed if it doesn’t depend on a value returned from a data race, a key intuition for causality in JMM
Causality in Java Memory Model

• Justification
  —to justify the early write in the previous example, we need to find a well-behaved execution in which those writes took place, and use that execution to perform the justification.

• Iterative growing
  —allows an action to be committed if it occurs in some well-behaved execution that also contains actions committed so far
  —may commit any uncommitted writes that occur in the well-behaved execution

• Well-behaved execution
  —a read that’s not yet committed must return the value of a write that’s ordered before it by happens-before
  —intuitively, an execution is well-behaved if an action is not dependent on a read returning a value from a data-race
Compiler Transformations

- Trace-preserving transformation
- Redundant read elimination
- Irrelevant read elimination
- Redundant write elimination
if (r1==1) {
    x=1; y=1;
} else {
    x=1; y=1;
}
y=1; x=1;
Redundant Read Elimination

\[
x = r_1 \\
r_2 = x \\
\text{if (} r_1 == r_2 \text{)} \\
y = 1
\]

\[
x = r_1 \\
r_2 = r_1 \\
\text{if (} r_1 == r_2 \text{)} \\
y = 1
\]

(read after write)
Irrelevant Read Elimination

\[ r1 = x \]
\[ r1 = 1 \]
\[ r1 = 1 \]
Redundant Write Elimination

\[ x = 1 \]
\[ x = 3 \]  \[\text{arrow}\] \[ x = 3 \]

(write before read)
Invalid Compiler Transformation

Reordering

• Reordering of independent statements is an important transformation that swaps two consecutive non-synchronization memory accesses—often performed in hardware, or in a compiler’s loop optimizer.

—Manson et al. claim this transformation to be valid in the JMM [Theorem 1]

• Cenciarelli et al. found a counterexample to this—Consider the program where threads 1 and 2 run in parallel:

```
1: r1 = x; r2 = y; if (r1 == 1 && r2 == 1) z = 1;
2: r3 = z; if (r3 == 1) { x = 1; y = 1; } else { y = 1; x = 1; }
```

After reordering the independent statements in the else branch, a compiler may execute assignments x = 1; and y = 1; early, so that r1, r2, r3 can all be assigned 1. However, such a behavior is not legal according to the JMM, as it violates the condition that the happens-before orders during validation be consistent with the final happens-before on the committed actions.
Summary: Purpose of JMM

- Provide a clear semantics for multithreaded code in Java
- Preserve safety and security properties of Java
- Allow aggressive compiler optimizations within a thread
- Reap the performance benefits of relaxed orderings in hardware
- Enable programmers to write correct and efficient programs
References


• http://en.wikipedia.org/wiki/Double-checked_locking