The Java Memory Model

Presentation: Ruedi Steinmann

Authors:

Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana-Champaign, IL

Jeremy Manson and William Pugh
Department of Computer Science
University of Maryland, College Park
College Park, MD
What is a Memory Model good for?

- Context: Multiple Threads
- Sequential Consistency
- Optimization -> Relaxed Memory Models

```plaintext
y = 0;
If (x != 0)
  y = 1;

y = 1;
If (x == 0)
  y = 0;
```
Requirements on the JMM

• Common Optimizations Possible
• If a program is correctly synchronized, then all executions of the program will appear to be sequentially consistent.
Requirements on the JMM

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

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 = x; )</td>
<td>( r_2 = y; )</td>
</tr>
<tr>
<td>( y = r_1; )</td>
<td>( x = r_2; )</td>
</tr>
</tbody>
</table>

Incorrectly synchronized, but we want to disallow \( r_1 == r_2 == 42 \).

Figure 2: An Out Of Thin Air Result
“ready” is volatile

- (1, 2)(3, 4)

Thread 1
(1) x=1
(2) ready=true

Thread 2
(3) if (ready)
(4) r2=x
Synchronizes-With order

- Monitor unlock -> monitor lock
- Volatile variable write -> all reads of that write
Synchronizes-With order

Empty

Ready is volatile

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X=1</td>
</tr>
<tr>
<td>2</td>
<td>if (ready)</td>
</tr>
<tr>
<td>3</td>
<td>ready=true</td>
</tr>
<tr>
<td>4</td>
<td>r1=x</td>
</tr>
</tbody>
</table>
Synchronizes-With Order

(2,3)

Ready is volatile

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>X=1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ready=true</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>if (ready)</td>
</tr>
<tr>
<td></td>
<td>r1=x</td>
</tr>
</tbody>
</table>
Happens-Before

Transitive Closure of Synchronizes-With and Program Order

(1,3)(2,4)
Happens-Before

\[(1,2)(1,3)(1,4)(2,3)(2,4)(3,4)\]
Highlevel View of the JMM and Happens-Before Memory Model

Given a program and a execution trace, determines if the execution was legal.
Happens-Before Memory Model

- a read cannot see a write that happens-after it
- \( w \rightarrow w' \rightarrow r \Rightarrow r \) does not see \( w \)
- A read may see a write that is not hb ordered to it
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 $r1 == r2 == 42$.  

**Figure 2: An Out Of Thin Air Result**
The Java Memory Model

- Formal specification in the paper
- Restriction of the Happens Before-Model
Committing in Rounds

- Execution to be justified: E
- Each round i consists of a
  - Sets of actions Pi,Ci,Fi
  - Justifying execution Ei
- All Ei and E obey HB-Model
The JMM

The JMM model can be divided into three main stages: Past, Current, and Future. Each stage is further divided into three categories:

**Past**
- Already committed

**Current**
- Currently being committed

**Future**
- Not yet committed

The mathematical relationships for these stages are given by:

\[ P(i+1) = P_i + C_i \]
\[ F(i+1) = F_i - C(i+1) \]
The Rules

- For any write $w$ in $Pi$ or $Ci$, $Vi(w) = V(w)$
- For any read $r$ in $Fi$, $Wi(r) \leftrightarrow r$
- For any read $r$ in $Ci$, $Wi(r)$ in $Pi$ and $W(r)$ in $Pi$
- For any read $r$ in $Pi$, $Wi(r) = W(r)$
Example 1

Initially $x=y=0$

- **Thread 1**
  - $r_1 := x$
  - $y := 1$

- **Thread 2**
  - $r_2 := y$
  - $x := 1$

$r_1 = r_2 = 1$
Example 2

Initially $x=y=0$

- **Thread 1**
  - $r1:=x$
  - If ($r1!=0$)
    - $y:=r1$
  - else
    - $y:=1$

- **Thread 2**
  - $r2:=y$
  - $x:=r2$

$r1=r2=1$
Example 3

Initially $x=y=0$

- **Thread 1**
  - $r1:=x$
  - If ($r1==0$)
    - $x:=1$
  - $r2=x$
  - $y=r2$
- **Thread 2**
  - $r3:=y$
  - $x:=r3$

$r1=r2=r3=1$
Conclusion

- Big improvement over previous version
- Still surprising results
- Too complicated?
- Personal remark: Paper could be written easier to understand.

\[
\begin{array}{c|c}
\text{x = y = z = 0} \\
r1:=z \\
\text{if (r1==1) \{x:=1; y:=1\}} \\
\quad \text{else \{y:=1; x:=1\}} \\
r2:=x \\
r3:=y \\
\text{if (r2==1 && r3==1)} \\
\quad z:=1
\end{array}
\]
Questions?
The Rules

1. \( C_i \subseteq A_i \)

2. \( \xrightarrow{h^b_i} |C_i| = \xrightarrow{h^b} |C_i| \)

3. \( \xrightarrow{s^o_i} |C_i| = \xrightarrow{s^o} |C_i| \)

4. \( V_i |c_i = V |c_i \)

5. \( W_i |c_{i-1} = W |c_{i-1} \)

6. For any read \( r \in A_i - C_{i-1} \), we have \( W_i(r) \xrightarrow{h^b_i} r \)

7. For any read \( r \in C_i - C_{i-1} \), we have \( W_i(r) \in C_{i-1} \) and \( W(r) \in C_{i-1} \)
The JMM

- Already committed
  - Reads see same write in $E_i$ and $E$

- Newly committed
  - Reads see writes that happen before them
  - Write seen in $E$ by reads must already be committed

- Not yet committed
  - Happens-before consistent
  - Already committed reads see same value in $E$ and $E_i$
Example 1: Proposed Execution

Thread 1
\[ y := 1 \]
\[ r1 := x \]

Thread 2
\[ r2 := y \]
\[ x := r2 \]
Example 1

- All Ei: first Thread 1, then Thread 2
- C0: y:=0; x:=0
- C1: y:=1; r2:=y(C0); r1:=x(C0);
- C2: r2:=y(C1); x:=r2;
- C3: r1:=x(C2);
Example 2: Proposed Execution

Thread 1
y:=1
r1:=x
if (r1!=0)  
y:=r1

Thread 2
r2:=y
x:=r2
Example 2

- All Ei: first Thread 1, then Thread 2
- C0: y:=0; x:=0
- C1: y:=1; r2:=y(C0); r1:=x(C0);
- C2: r2:=y(C1); x:=r2;
- C3: r1:=x(C2); y:=r1
Conflicting Access

A read of or write to a variable is an access to that variable.

Two accesses to the same shared field or array element are said to be conflicting if at least one of the accesses is a write.
Data Race

Two accesses x and y form a data race in an execution of a program if they are from different threads, they conflict, and they are not ordered by happens-before
Correctly Synchronized Program

A program is said to be correctly synchronized or data-race-free if and only if all sequentially consistent executions of a program are free of data races.
Well Formed Executions

- Read r, Write w
- $W(r)$: Write seen by r
- $V(w)$: Value written by w
- Volatile read r:
  - It is not the case that $r \overset{so}{\rightarrow} W(r)$
  - No write w such that $W(r) \overset{so}{\rightarrow} w \overset{so}{\rightarrow} r$
Well Formed Executions

- Non volatile read \( r \):
  - It is not the case that
  - No write \( w \) such that

\[
\begin{align*}
  r & \xrightarrow{hb} W(r) \\
  W(r) & \xrightarrow{hb} w \xrightarrow{hb} r
\end{align*}
\]
Write Rules

- Write must be present in Ei
- Values written are always final
Read Rules

• For newly commited reads (C_i-\text{C}(i-1)):
  – W(r) happens before r in E_i
  – W(r) in E is present in C(i-1)
  – W(r) in E_i is present in C(i-1)

• For reads already commited (C(i-1))
  – Read sees same write in E_i and E
Examples

We only treat examples that work, i.e. that are legal under the JMM.
## Synchronization Order

(2,3)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td>X=1</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>ready=true</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td></td>
</tr>
<tr>
<td><strong>4</strong></td>
<td>if (ready)</td>
</tr>
<tr>
<td></td>
<td>r1=x</td>
</tr>
</tbody>
</table>
Synchronization Order

(2,3)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: X=1</td>
<td>if (ready)</td>
</tr>
<tr>
<td>2</td>
<td>r1=x</td>
</tr>
<tr>
<td>3: ready=true</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>