Principles of Concurrency

Lecture 13
Memory Models: Java

Based on:
- On Validity of Program Transformations in the Java Memory Model, (Sevcik and Aspinall, ECOOP’08)
The sequential semantics of Java is well-specified and well-understood.

About Java concurrency:
- Do we have a formal definition?
- Do people understand it?
First Try: Sequential Consistency

- During execution, memory actions should appear to execute one at a time in an imaginary interleaving of actions on a shared memory.

- In particular, reads of a shared variable should return the value written most recently to the memory.

- Example:

  \[
  \begin{align*}
  & x \leftarrow 0; y \leftarrow 0 \\
  & r_1 \leftarrow x \quad r_2 \leftarrow y \\
  & y \leftarrow 1 \quad x \leftarrow 1
  \end{align*}
  \]
Some More SC Executions

\[
\begin{align*}
W_0x &\rightarrow W_0y \\
& \downarrow \\
& \text{Not SC} \\
R_y &\rightarrow t_2 \\
& \downarrow \\
W_x[1] &\rightarrow t_2 \\
& \downarrow \\
t_1 &\rightarrow Rx \\
& \downarrow \\
t_1 &\rightarrow W_y[1] \\
r_1 = 1 &\quad r_2 = 1
\end{align*}
\]

\[
\begin{align*}
W_0x &\rightarrow W_0y \\
& \downarrow \\
t_1 &\rightarrow St \\
& \downarrow \\
t_2 &\rightarrow St \\
r_1 = 1 &\quad r_2 = 0
\end{align*}
\]

\[
\begin{align*}
W_0x &\rightarrow W_0y \\
& \downarrow \\
t_1 &\rightarrow Rx \\
& \downarrow \\
t_2 &\rightarrow Ry \\
r_1 = 0 &\quad r_2 = 0
\end{align*}
\]

\[
\begin{align*}
W_0x &\rightarrow W_0y \\
& \downarrow \\
t_1 &\rightarrow W_y[1] \\
& \downarrow \\
t_2 &\rightarrow Ry \\
r_1 = 0 &\quad r_2 = 1
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
x \leftarrow 0; y \leftarrow 0 \\
\hline
r_1 \leftarrow x & y \leftarrow 1 \\
r_2 \leftarrow y & x \leftarrow 1
\end{array}
\end{align*}
\]
**SC memory model**

- Do we have a formal definition? yes
- Do people understand it? yes
- Is it faithful to Java semantics and processor behaviour? no
  - compilers will often reorder program statements
  - and hardware will do the same

---

A Java program may exhibit this execution

---

\[
\begin{align*}
W_{0x} & \\
\downarrow & \\
W_{0y} & \\
\downarrow & \\
t_1 & \text{St} \\
\downarrow & \\
\text{St} & t_2 \\
\downarrow & \\
\text{Ry} & t_2 \\
\downarrow & \\
W_{x[1]} & t_2 \\
\downarrow & \\
t_1 & \text{Rx} \\
\downarrow & \\
W_{x[1]} & W_{y[1]} \\
\downarrow & \\
r_1 = 1 & r_2 = 1
\end{align*}
\]

---

\[
\begin{align*}
W_{0x} & \\
\downarrow & \\
t_1 & \text{St} \\
\downarrow & \\
\text{St} & t_2 \\
\downarrow & \\
t_1 & \text{Ry} \\
\downarrow & \\
W_{x[1]} & W_{y[1]} \\
\downarrow & \\
r_1 = 0 & r_2 = 1
\end{align*}
\]
Sequential Consistency and Java

Let's try...

\[
\begin{align*}
    &x \leftarrow 0; y \leftarrow 0 \\
    y &\leftarrow 1 \quad | \quad x \leftarrow 1 \\
    r_1 &\leftarrow x \quad | \quad r_2 \leftarrow y \\
    r_1 &= 0 \quad r_2 &= 0
\end{align*}
\]

is not visible in SC

```java
public class Relaxed1 extends Thread {
    public static volatile long iter = 0;
    public static volatile int fence = 0;
    public static int res0 = 1;
    public static int res1 = 1;
    private static int x = 0;
    private static int y = 0;
    private int tid;
    private long wait;

    public Relaxed1(int tid, long wait) {
        this.tid = tid;
        this.wait = wait;
    }

    public void run() {
        if (this.tid == 0) {
            while (iter == this.wait);
            y = 1;
        }

        while (res0 != 0 || res1 != 0) {
            Relaxed1 th1 = new Relaxed1(0, iter);
            Relaxed1 th2 = new Relaxed1(1, iter);
            th1.start();
            th2.start();
            iter++;
            try {
                th1.join();
                th2.join();
            } catch (InterruptedException e) {}
            x = 0;
            y = 0;
        }
        if (res0 == 0 && res1 == 0)
            System.out.println("Relaxed after "+iter+" iterations!");
    }
}
```
The Java Memory Model

- Proposed in 1995, but broken.
- New version in 2004 (JSR-133). Published POPL’05 (Manson, Pugh, Adve)
  - Also broken :-)

- JMM tries *squaring the circle*
  - Guarantees for programmers
    - Data-race free programs execute like in a SC model
    - A (safe) formal semantics for all Java programs (incl. those with races)

- Guarantees for optimizers
  - Allows all various hardware reordering semantics
  - Allows aggressive compiler optimizations
An interleaving of actions is better described without global time. It is in fact a consistent extension of a partial order called **happens before**.

The hb relation is the smallest transitive relation that respects
- program order between actions of same thread
- synchronization order between actions of threads
This relation should constrain the write actions that a read action can legally see.

\[
\begin{align*}
x & \leftarrow 0; y \leftarrow 0 \\
r_1 & \leftarrow x \\
y & \leftarrow 1 \\
r_2 & \leftarrow y \\
x & \leftarrow 1
\end{align*}
\]
Happens-Before Model

An axiomatic execution is described by a tuple

$$\langle P, A, ^{p_o} \to, ^{s_o} \to, W \rangle$$

Among all executions, the happens-before model selects the so-called **well-formed executions**

Ex: a read action must not see a write that happens after it.

Such a model
- is quite easy to understand,
- contains all SC executions,
- but is too relaxed: it does not respect the DRF property
- allows cyclic dependencies
**Data Race Freedom**

**Data race free program:**

- Intuitively: a program where two threads don’t access at the same time a non-volatile shared location (at least one of the access is a write)
- Formally: in all executions of the model, conflicting actions must be in the \(hb\) relation

Data race free model: data-race free programs only have SC executions.
Race Committing Sequence

Is \( r_1 = 1 \) \( r_2 = 1 \) allowed?

1. Start from a well-behaved execution
2. Commit races: modifies the write-seen
3. Restart the execution taking the race into account

+ constraints on the sequence to rule out cyclic causality

a well-behaved execution

one write-seen is updated

final JMM execution
## Examples

<table>
<thead>
<tr>
<th>Initially</th>
<th>x = y = 0</th>
<th>r1 = x</th>
<th>r2 = y</th>
<th>y = 1</th>
<th>x = 1</th>
<th>r1 = x</th>
<th>r2 = y</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock m1</td>
<td>lock m2</td>
<td>unlock m1</td>
<td>unlock m2</td>
<td>lock m2</td>
<td>unlock m2</td>
<td>y = r1</td>
<td>x = r2</td>
</tr>
</tbody>
</table>

A. (allowed)  
B. (prohibited)  
C. (prohibited)

Is it possible to get \( r_1 = r_2 = 1 \) at the end of an execution?
Compiler Transformations

The meaning of a Java program is given by its set of traces:

- A transformation that does not change this set is trivially valid

\[
\text{if (r1==1)}
\begin{align*}
\{x=1; y=1\} & \quad \iff \quad x=1 \\
\text{else } \{x=1; y=1\} & \quad \iff \quad y=1
\end{align*}
\]

Independent writes can be reordered

\[
x=1 \quad \rightarrow \quad y=1 \\
y=1 \quad \rightarrow \quad x=1
\]

Elimination of redundant reads

\[
\begin{align*}
r_1 = x & \quad r_1 = x & \quad x = r_1 & \quad x = r_1 \\
r_2 = x & \quad r_2 = r_1 & \quad r_2 = x & \quad r_2 = r_1 \\
\text{if } (r_1==r_2) & \quad \text{if } (r_1==r_2) & \quad \text{if } (r_1==r_2) & \quad \text{if } (r_1==r_2) \\
y = 1 & \quad y = 1 & \quad y = 1 & \quad y = 1
\end{align*}
\]

(read after read) (read after write)

Is this valid?
Irrelevant Read Introduction (Speculation)

\[
\text{if (r1==1)} \begin{cases} 
    r2=x \\
    y=r2
\end{cases} \quad \text{or} \quad \begin{cases} 
    r2=x \\
    y=r2
\end{cases} \quad \text{if (r1==1)}
\]

Is this valid?

Roach Motel Semantics

\[
\begin{align*}
x &= 1 \\
\text{lock m} \\
y &= 1 \\
\text{unlock m}
\end{align*} \quad \begin{align*}
\text{lock m} \\
x &= 1 \\
y &= 1 \\
\text{unlock m}
\end{align*}
\]

Interestingly, this transformation is invalid in general
Redundant Write-After-Read Elimination

<table>
<thead>
<tr>
<th>Initially x = 0</th>
<th>lock m1</th>
<th>lock m2</th>
<th>lock m1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x=2</td>
<td>x=1</td>
<td>r1=x</td>
</tr>
<tr>
<td>lock m1</td>
<td>unlock m1</td>
<td>unlock m2</td>
<td>r1=x</td>
</tr>
<tr>
<td>unlock m1</td>
<td></td>
<td></td>
<td>r2=x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unlock m2</td>
</tr>
<tr>
<td>unlock m1</td>
<td></td>
<td></td>
<td>unlock m1</td>
</tr>
</tbody>
</table>

Is this valid?

- No well-behaved execution contains a data race
- The read “r2 = x” must always see the write “x=r1”
- Removing the redundant write allows r1 and r2 to see different values
Compiler Transformations

<table>
<thead>
<tr>
<th>x = y = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1=x</td>
</tr>
<tr>
<td>r2=y</td>
</tr>
<tr>
<td>y=r1</td>
</tr>
<tr>
<td>if (r2==1) {</td>
</tr>
<tr>
<td>r3=y</td>
</tr>
<tr>
<td>x=r3</td>
</tr>
<tr>
<td>} else x=1</td>
</tr>
</tbody>
</table>

Suppose we rewrite to r3=r2. Can we now observe r1=r2=1?

- Initially only one well-behaved execution, r1=r2=0
- Two data races:
  - between y=r1 and r2=y:
    - r2=0 is guaranteed
  - between r1=x and x=1 with value 1
    - now, commit data race between y=r1 and r2=y with value 1
    - if r1=r2=1, then r3=y must read a value that happens-before it
      - there’s no data race between r3=y and y=r1
      - thus, the only value that it can read is y=0
      - then x=r3 must write 0: contradiction
  - Transforming r3=y to r3=r2 allows r1=r2=1
    - commit the data race between r1=x and x=1
    - then, commit the data race between y=r1 and r2=y
    - can keep commitment to write x=1 since r1=r2=1