Java Memory Model

Jian Cao

Department of Electrical and Computer Engineering
Rice University

Sep 22, 2016
Content

- Introduction
- Java synchronization mechanism
- Double-checked locking
- Out-of-Thin-Air violation and Happens-before
- Causality and the Java memory model
Today’s References


Why a Memory Model for Java?

• For a high-level programming languages like Java, the memory model determines:

  The transformations the compiler may apply to a program when producing bytecode.

  The transformations a virtual machine may apply to bytecode when producing native code.

  The optimizations that hardware may perform on the native code.
Java Memory Model

• Memory models for languages are hard.

• **First** Java memory model had several shortcomings:
  – A class of executions containing data races were either allowed, or were only prohibited by enforcing traditional control and data dependences.
  – Resulted in safety and security issues.

• Today’s paper: **second** memory model for Java.
Today’s Java Memory Model

• Guarantees *sequential consistency* to data-race-free programs.

• The behavior of incorrectly synchronized programs is bound by a well defined notion of causality.

• Provides balance between *safety and security* and compiler and hardware *optimizations*. 
Synchronizing Access to Objects in Java

Multiple threads share a counter

```java
public class Counter {
    private int count = 0;
    public void increment() {
        count = count + 1;
    }
}
```

Data race:
multiple threads call increment()
multiple threads read and write to count variable

Conflicting access:
the accesses to the same shared field,
at least one is a write

Result: unpredictable outcomes
Synchronization Mechanisms in Java

- Synchronized methods
- Synchronized blocks
- Explicit locking
- Volatile variables
• Every object has an **intrinsic lock** (or monitor).
• A thread that needs exclusive and consistent access must: 
  * acquire the object's intrinsic lock before accessing them;*
  * release the intrinsic lock when it's done with them.*
Implicit Locking: **Synchronized Keyword**

**Synchronized method:**

```java
public class SynchronizedCounter {
    private int count = 0;
    public synchronized void increment() {
        count = count + 1;
    }
}
```

**Synchronized block:**

```java
public class SynchronizedCounter {
    private int count = 0;
    public void increment() {
        synchronized(this) {
            count = count + 1;
        }
    }
}
```

In Synchronized block, locks are bound to current thread, and other threads will have to wait when one thread is in.

https://www.google.com/search?q=java+synchronization+mechanisms&espv=2&biw=1174&bih=568&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiEyt_ovJ_PAhVLLlyYKHeqAC3EQ_AUIBygC#imgrc=78-VgskbOayL-M%3A
Explicit Locking: **Lock Interface**

ReentrantLock

```java
ReentrantLock lock = new ReentrantLock();
int count = 0;
void increment() {
    lock.lock();
    try{
        count++;
    }
    finally {
        lock.unlock();
    }
}
```

Only one thread can hold a lock at any given time. A thread waits inside the lock() operation until the lock is available (unlocked)
After the write lock has been released both read tasks are executed in parallel. The read-lock can be held simultaneously by multiple threads as long as no thread holds the write-lock.
Java in Practice: Double-checked Locking

• An efficient method for implementing lazy initialization in a multithreaded environment:

```java
// Single threaded version
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null)
            helper = new Helper();
        return helper;
    }
    // other functions and members...
}
```

When used in a multithreaded context, two or more Helper objects could be allocated.
Avoid Data Races with “synchronized”

```java
// Correct multithreaded version
class Foo {
    private Helper helper = null;
    public synchronized Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }
    // other functions and members...
}
```

Mutual exclusion. When one thread is accessing the object, other threads must wait.
Avoid Data Races with “synchronized”

```java
// Broken multithreaded version
// "Double-Checked Locking" idiom
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null)
            synchronized(this) {
                if (helper == null)
                    helper = new Helper();
            }
        return helper;
    }
    // other functions and members...
}
```

The Helper object is initiated, yet the fields inside constructor might not be fully initiated.
Another Fix by Nested Synchronization

// (Still) Broken multithreaded version
// "Double-Checked Locking" idiom
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            Helper h;
            synchronized(this) {
                h = helper;
                if (h == null)
                    synchronized (this) {
                        h = new Helper();
                    } // release inner synchronization lock
                helper = h;
            }
        }
        return helper;
    }
    // other functions and members...
}

Actions before monitorexit must be performed before the monitor is released. Actions after monitorexit may be performed before the monitor is released.
Problem: Copies of Shared Data

The JVM is free to optimize use of variables as it pleases, for example, putting copies in registers.
Using Java Volatile Keyword for Shared Data

volatile
- Must update the shared variable for each write
- Must read the shared variable for each read

Example

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>( \text{if (ready)} )</td>
</tr>
<tr>
<td>( \text{ready} = \text{true} )</td>
<td>( r1 = x; )</td>
</tr>
</tbody>
</table>

If \( r1 = x \); executes, it will read 1.
A write of a volatile cannot be reordered with respect to any previous read or write; A read of a volatile cannot be reordered with respect to any following read or write.
Avoiding Excess Loads and Fences

When helper is already initialized, the volatile field is only read once, which can improve the method's overall performance by as much as 25 percent.

```java
// Works with acquire/release semantics for volatile in Java 1.5 and later
// Broken under Java 1.4 and earlier semantics for volatile

class Foo {
    private volatile Helper helper;

    public Helper getHelper() {
        Helper result = helper;
        if (result == null) {
            synchronized(this) {
                result = helper;
                if (result == null) {
                    helper = result = new Helper();
                }
            }
        }
        return result;
    }

    // other functions and members...
}
```
Java Memory Model in Detail
Correctly Synchronized Programs

Synchronization order:

• **Synchronizes-with:**
  • unlock and lock operations synchronize
  • volatile variable: a write to a volatile variable $v$ synchronizes-with all subsequent reads of $v$ by any thread

• **Happens-before:**
  • a transitive closure of program order and the synchronization order
Correctly Synchronized Programs

**Data races:** Two accesses $x$ and $y$ form a data race in an execution of a program if they are from different threads, the conflict, and they are not ordered by happens-before.

**Definition:** A program is said to be *correctly synchronized* or *data-race-free* if and only if all sequentially consistent executions of the program are free of *data races.*
An Example of Incorrect Program

<table>
<thead>
<tr>
<th>Initially, $x == y == 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 1</strong></td>
</tr>
<tr>
<td>1: $r_2 = x$;</td>
</tr>
<tr>
<td>2: $y = 1$;</td>
</tr>
</tbody>
</table>

$r_2 == 2$, $r_1 == 1$ violates sequential consistency.
An Example of Incorrect Program

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></td>
</tr>
<tr>
<td>4: $x = 2$</td>
<td></td>
</tr>
</tbody>
</table>

$r2 == 2, r1 == 1$ violates sequential consistency.

Incorrectly synchronized: Any sequentially consistent execution of the code contains conflicting accesses to $x$ and $y$ that are not ordered by happens-before.

Fix: declare $x$ and $y$ as volatile variables.
Another Incorrectly Synchronized Example

Thread 1
\[ y = 0; \]
\[ \text{Lock}(L) \{ \]
\[ \quad y ++; \]
\[ \}
\[ \text{Unlock}(L) \]

Thread 2
\[ \text{Lock}(L) \{ \]
\[ \quad y ++; \]
\[ \}
\[ \text{Unlock}(L) \]
if (y==…){
execute
}

Sync edge
Thread 1
Thread 2

No data races

Data races

Thread 1
\[ \text{Lock}(L) \{ \]
\[ \quad y ++; \]
\[ \}
\[ \text{Unlock}(L) \]
\[ \text{Lock}(L) \{ \]
\[ \quad y ++; \]
\[ \}
\[ \text{Unlock}(L) \]
if (y ==…){
execute
}
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 $r1 == r2 == 42$. 
Incorrect Program: Out-of-thin-air Problem

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

\[
\begin{array}{c|c}
\text{Thread 1} & \text{Thread 2} \\
\hline
r1 = x; & r2 = y; \\
y = r1; & x = r2; \\
\hline
\end{array}
\]

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

Incorrectly synchronized:
Thread 1 could \textit{speculatively} write the value 42 to \( y \), which would allow Thread 2 to read 42 for \( y \) and write it out to \( x \), which would allow Thread 1 to read 42 for \( x \), and \textit{justify its original speculative write} of 42 for \( y \).
Incorrect Program: Out-of-thin-air Guarantees

• A self-justifying write speculation like that one can:
  – Cause code to misbehave
  – Create serious security violations and needs to be disallowed.

• For example, this would be a serious problem if the value that was produced out of thin air was a reference to an object that the thread was not supposed to have.
Simple Version: Happens-before Model

A read $r$ of a variable $v$ is allowed to observe a write $w$ to $v$ if, in the happens-before partial order of the execution:

- $r$ does not happen-before $w$ – a read cannot see a write that happens-after it

- there is no intervening write $w0$ to $v$, ordered by happens before – the write $w$ is not overwritten along a happens-before path
Causality and the Java Memory Model

Consistent with happens-before model

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

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>If ( r_1 != 0 )</td>
<td>If ( r_2 != 0 )</td>
</tr>
<tr>
<td>( y = 42; )</td>
<td>( x = 42; )</td>
</tr>
</tbody>
</table>

Correctly synchronized, so we must disallow \( r_1 == r_2 == 42 \).

Figure 4: Correctly Synchronized Program

Disallow in Java Memory Model

Allow in Java Memory Model

Before compiler transformation

Initially, \( a = 0, b = 1 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( r_1 = a; )</td>
<td>5: ( r_3 = b; )</td>
</tr>
<tr>
<td>2: ( r_2 = a; )</td>
<td>6: ( a = r_3; )</td>
</tr>
<tr>
<td>3: If ( r_1 == r_2 )</td>
<td>( b = 2; )</td>
</tr>
<tr>
<td>4: ( b = 2; )</td>
<td>( a = r_3; )</td>
</tr>
</tbody>
</table>

Is \( r_1 == r_2 == r_3 == 2 \) possible?

Figure 5: Effects of Redundant Read Elimination

After compiler transformation

Initially, \( a = 0, b = 1 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4: ( b = 2; )</td>
<td>5: ( r_3 = b; )</td>
</tr>
<tr>
<td>1: ( r_1 = a; )</td>
<td>6: ( a = r_3; )</td>
</tr>
<tr>
<td>2: ( r_2 = r_1; )</td>
<td>( b = 2; )</td>
</tr>
<tr>
<td>3: If ( true );</td>
<td>( a = r_3; )</td>
</tr>
<tr>
<td>( r_1 == r_2 == r_3 == 2 ) is sequentially consistent</td>
<td></td>
</tr>
</tbody>
</table>
Causality and the Java Memory Model

• Observation: Early execution of an action does not result in an undesirable causal cycle if its occurrence is not dependent on a read returning a value from a data race.

• Formalizing the *well-behaved* executions:
A read that is not yet committed must return the value of a write that is ordered before it by happens-before.
Security and Safety

• Shows that the safety and security properties of Java require prohibiting a class of executions that contain data races and have not been previously characterized.

• Characterizes speculations that constitutes out-of-thin-air violations, that might lead to severe security issue.
Summary: Balances Three Crucial Needs

• Allows compiler to perform code transformations and optimizations

• Provides clear and simple programming model for those writing concurrent code
  – For data-race-free programs, allows programmers to reason about their programs using simple semantics of sequential consistency

• Guarantees that even programs with data races don’t violate security
  – Provides a clear definition for program behaviors allowed when executing programs data races: behaviors must be causal; can’t read values that haven’t been written or have been overwritten
Take Away Messages

• Major design goal for Java memory model:
  Provides a balance between security, compiler and hardware optimizations for performance, and usability.

• Volatile variable:
  Guarantees the happens-before order.

• For incorrect programs:
  Prevents Out-of-Thin-Air violation.