Close Encounters of the Java Memory Model Kind

Aleksey Shipilev

Presented by Sankalp Gambhir for the LAMP PL Seminar, 14 Nov 2023

This is a summary of a long-running discussion

Thanks to:

- Simon Guilloud, Shardul Chiplunkar, Matt Bovel, Viktor Kunčak, other CS 206 teaching staff
- Guillaume Martres and Sébastien Doeraene for answering stupid questions
- Guillaume specially for the references for this talk!

What is a memory model?

- Spec describes an abstract machine model

What is a memory model?

- Spec describes an abstract machine model
- The model has to subsume memory interactions

What is a memory model?

- Spec describes an abstract machine model
- The model has to subsume memory interactions
- This description is the memory model

What is a memory model?

- Spec describes an abstract machine model
- The model has to subsume memory interactions
- This description is the memory model
- Can be largely disconnected from the rest of the model

What is a memory model?

- Spec describes an abstract machine model
- The model has to subsume memory interactions
- This description is the memory model
- Can be largely disconnected from the rest of the model
- But forms its foundations

Sequential Code

```
// main thread
var x: Int = 0
val a0 = x
x = 1
val a1 = x
x = 2
val a2 = x
x = 3
val a3 = x
x = 4
val a4 = x
x = 5
val a5 = x
(a0, a1, a2, a3, a4, a5)
// (0, 1, 2, 3, 4, 5)
```

Sequential Consistency

```
// t0
x = 1
y = 1
println(x)
```

```
// t1
x = 2
println(y)
y = 2
println(x)
```

Sequential Consistency

```
// t0
x = 1
y = 1
println(x)
```

```
// t1
x = 2
println(y)
y = 2
println(x)
```

The result should be an interleaving of the events

Sequential Consistency

```
// t0
x = 1
y = 1
println(x)
```

```
// t1
x = 2
println(y)
y = 2
println(x)
```

The result should be an interleaving of the events

As the name suggests, as close to emulating sequential behaviour as possible.

Sequential Consistency

```
// t0
x = 1
y = 1
println(x)
```

```
// t1
x = 2
println(y)
y = 2
println(x)
```

The result should be an interleaving of the events

As the name suggests, as close to emulating sequential behaviour as possible. Standard mental model of programmers too.

Running a small test

```
var a: Boolean = false
var b: Boolean = false

```

```
// t0
a = true
if b then
  0
else
  1

```

```
// t1
b = true
if a then
  0
else
  1

```

Outputs?

Running a small test

```
var a: Boolean = false
var b: Boolean = false

```

```
// t0
a = true
if b then
  0
else
  1

```

```
// t1
b = true
if a then
  0
else
  1

```

Outputs?

(0, 0)?

Running a small test

```
var a: Boolean = false
var b: Boolean = false

```

```
// t0
a = true
if b then
  0
else
  1

```

```
// t1
b = true
if a then
  0
else
  1

```

Outputs?

(0, 0)? (1, 0)?

Running a small test

```
var a: Boolean = false
var b: Boolean = false

```

```
// t0
a = true
if b then
  0
else
  1

```

```
// t1
b = true
if a then
  0
else
  1

```

Outputs?

(0, 0)? (1, 0)? (0, 1)?

Running a small test

```
var a: Boolean = false
var b: Boolean = false

```

```
// t0
a = true
if b then
  0
else
  1

```

```
// t1
b = true
if a then
  0
else
  1

```

Outputs?

(0, 0)? (1, 0)? (0, 1)? (1, 1)?

Running a small test

```
object NonSC extends TestHarness[Int]:
    class NonSCTest extends StressTest[Int]:
        // state
        var a: Boolean = false
        var b: Boolean = false

        Test { () =>
            a = true
            if b then
                0
            else
                1
        }

        Test { () =>
            b = true
            if a then
                0
            else
                1
        }
    end NonSCTest

    ↑ testCount
    val testCount: Int = 1000000

    run | debug
    @main def runNonSC =
        val res = test(NonSCTest(), 4)
        println(res)
```

Running a small test

```scala
object NonSC extends TestHarness[Int]:
    class NonSCTest extends StressTest[Int]:
        // state
        var a: Boolean = false
        var b: Boolean = false

        Test { () ⇒
            a = true
            if b then
                0
            else
                1
        }

        Test { () ⇒
            b = true
            if a then
                0
            else
                1
        }
    end NonSCTest

    ↑ testCount
    val testCount: Int = 1000000

    run | debug
    @main def runNonSC =
        val res = test(NonSCTest(), 4)
        println(res)
```

```
Map(
    List(1, 0) -> 499854,
    List(0, 1) -> 500141,
    List(0, 0) -> 1,
    List(1, 1) -> 4
)
```

Running a small test

```
object NonSC extends TestHarness[Int]:
    class NonSCTest extends StressTest[Int]:
        // state
        var a: Boolean = false
        var b: Boolean = false

        Test { () ⇒
            a = true
            if b then
                0
            else
                1
        }

        Test { () ⇒
            b = true
            if a then
                0
            else
                1
        }
    end NonSCTest

    ↑ testCount
    val testCount: Int = 1000000

    run | debug
    @main def runNonSC =
        val res = test(NonSCTest(), 4)
        println(res)
```

```
Map(
    List(1, 0) -> 499854,
    List(0, 1) -> 500141,
    List(0, 0) -> 1,
    List(1, 1) -> 4
)
```

Only $\sim 0.0004\%$ cases!

It ain't what you don't know that gets you into trouble. It's what you know for sure that just ain't so.

- Mark Twain

Just read the spec!

The Java Memory Model is the most complicated part of Java spec that must be understood by at least library and runtime developers. Unfortunately, it is worded in such a way that it takes a few senior guys to decipher it for each other. Most developers, of course, are not using JMM rules as stated, and instead make a few constructions out of its rules, or worse, blindly copy the constructions from senior developers without understanding the limits of their applicability. [1]

Java Memory Model (compacted garbage summary)

- All hail happens-before (hb)

Java Memory Model (compacted garbage summary)

- All hail happens-before (hb)
- Respect each thread's ordering — program order (po)

Java Memory Model (compacted garbage summary)

- All hail happens-before (hb)
- Respect each thread's ordering — program order (po)
- Respect reads and writes — reads-from (rf)

Java Memory Model (compacted garbage summary)

- All hail happens-before (hb)
- Respect each thread's ordering — program order (po)
- Respect reads and writes — reads-from (rf)
- Synchronization order (so) (the big one)

Java Memory Model (compacted garbage summary)

- All hail happens-before (hb)
- Respect each thread's ordering — program order (po)
- Respect reads and writes — reads-from (rf)
- Synchronization order (so) (the big one)
  - volatile ordering (vl)

Java Memory Model (compacted garbage summary)

- All hail happens-before (hb)
- Respect each thread's ordering — program order (po)
- Respect reads and writes — reads-from (rf)
- Synchronization order (so) (the big one)
    - volatile ordering (vl)
    - synchronizes-with (sw/so)

Java Memory Model (compacted garbage summary)

- All hail happens-before (hb)
- Respect each thread's ordering — program order (po)
- Respect reads and writes — reads-from (rf)
- Synchronization order (so) (the big one)
  - volatile ordering (vl)
  - synchronizes-with (sw/so)
  - thread ordering (th)

Java Memory Model (compacted garbage summary)

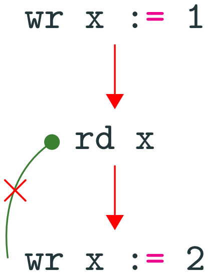
- All hail happens-before (hb)
- Respect each thread's ordering — program order (po)
- Respect reads and writes — reads-from (rf)
- Synchronization order (so) (the big one)
  - volatile ordering (vl)
  - synchronizes-with (sw/so)
  - thread ordering (th)

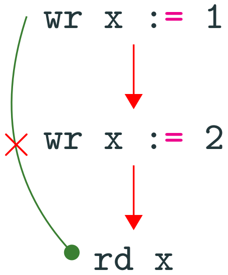
Java Memory Model (compacted garbage summary)

- All hail happens-before (hb)
- Respect each thread's ordering — program order (po)
- Respect reads and writes — reads-from (rf)
- Synchronization order (so) (the big one)
  - volatile ordering (vl)
  - synchronizes-with (sw/so)
  - thread ordering (th)

Two conditions for a valid execution:

- no cycles
- no invalid reads


wr x := 1
rd x
wr x := 2


wr x := 1
wr x := 2
rd x

Explaining the example

Legend

a $\xrightarrow{\texttt{po}}$ b	b is after a in the program order
a $\xrightarrow{\texttt{rf}}$ b	b reads from a in an execution
a $\xrightarrow{\texttt{vl}}$ b	b is after a in an execution, and they are events on the same `volatile` variable
a $\xrightarrow{\texttt{so}}$ b	b is after a in an execution, and they are `synchronized` block start or end events
a $\xrightarrow{\texttt{th}}$ b	b is after a in an execution, and they are related as thread-start/first-event or as last-event/thread-join

https://lampepfl.github.io/courses/cs206/ex-08-solution-9475bbf99cf472ec48e4

https://lampepfl.github.io/courses/cs206/ex-08

```
var a: Boolean = false
var b: Boolean = false
var x: Int = -1
var y: Int = -1

```

```
// t0
a = true
if b then
  x = 0
else
  x = 1

```

```
// t1
b = true
if a then
  y = 0
else
  y = 1

```

```
(wr y -1) (wr b f)  (wr a f) (wr x -1)

initialization

     th0                    th1

  (wr a t)               (wr b t)

 (rd b ??)              (rd a ??)

 (wr x ??)              (wr y ??)

reads

(rd x ??) (rd y ??)
```

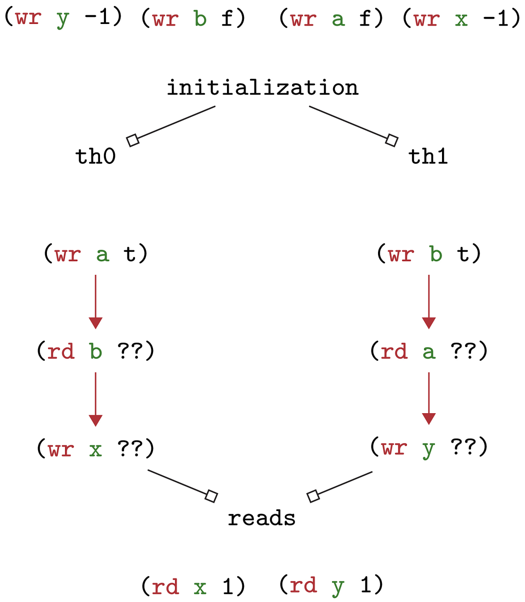
(wr y -1) (wr b f) (wr a f) (wr x -1)

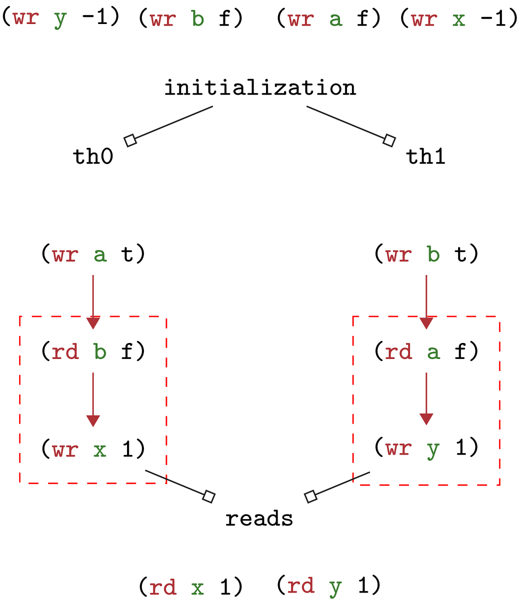
initialization

th0	th1
(wr a t)	(wr b t)
(rd b ??)	(rd a ??)
(wr x ??)	(wr y ??)

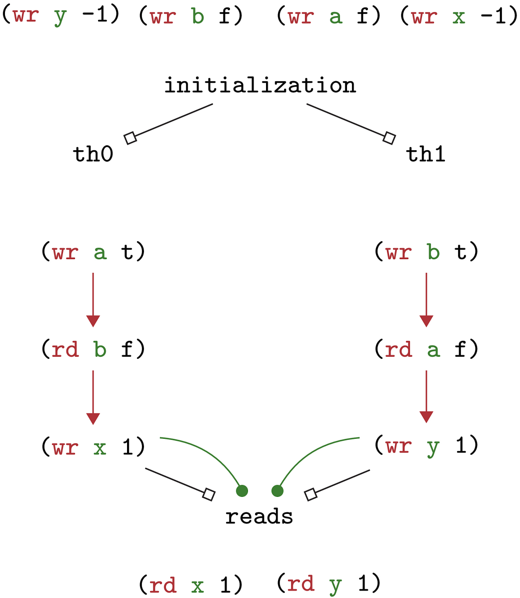
reads

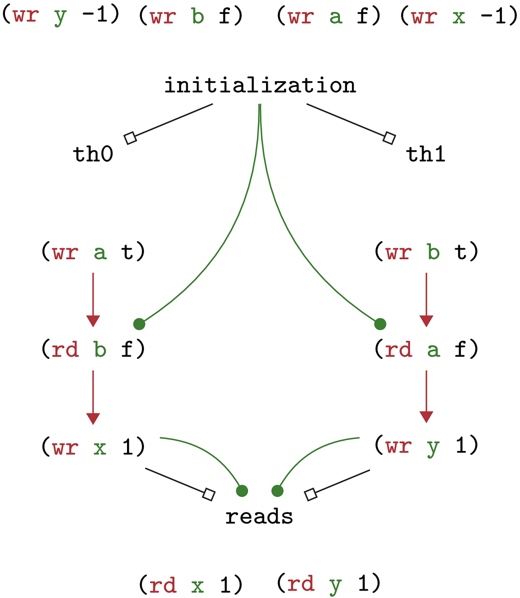
(rd x 1) (rd y 1)


(wr y -1) (wr b f) (wr a f) (wr x -1)
initialization
th0
th1
(wr a t)
(rd b ??)
(wr x ??)
(wr b t)
(rd a ??)
(wr y ??)
reads
(rd x 1) (rd y 1)



(wr y -1) (wr b f) (wr a f) (wr x -1)
initialization
th0
th1
(wr a t)
(rd b f)
(wr x 1)
(wr b t)
(rd a f)
(wr y 1)
reads
(rd x 1) (rd y 1)


(wr y -1) (wr b f) (wr a f) (wr x -1)
initialization
th0
th1
(wr a t)
(rd b f)
(wr x 1)
(wr b t)
(rd a f)
(wr y 1)
reads
(rd x 1) (rd y 1)


(wr y -1) (wr b f) (wr a f) (wr x -1)
initialization
th0
th1
(wr a t)
(wr b t)
(rd b f)
(rd a f)
(wr x 1)
(wr y 1)
reads
(rd x 1) (rd y 1)

Myth: my program is compiled per the spec

The spec is described on an abstract machine. The runtime only *emulates* it.

Myth: my program is compiled per the spec

The spec is described on an abstract machine. The runtime only *emulates* it.

As long as no one can call your bluff, you're good.

Myth: my program is compiled per the spec

The spec is described on an abstract machine. The runtime only *emulates* it.

As long as no one can call your bluff, you're good.

Swap statements? Remove entire blocks? As long as no one, *anywhere*, can differentiate the resulting effects, go ahead.

Myth: my program is compiled per the spec

The spec is described on an abstract machine. The runtime only *emulates* it.

As long as no one can call your bluff, you're good.

Swap statements? Remove entire blocks? As long as no one, *anywhere*, can differentiate the resulting effects, go ahead.

Pain: students seem to *love* thinking about compiler reorderings. To the JMM spec, they don't exist. Students refuse to accept this.

"Locks" as a mental model

```
var x: Int = 0
var y: Int = 0
```

```
// t0
synchronized {x = 1}
synchronized {y = 1}
```

```
// t1
val a = y
val b = x
println((a, b))
```

```
[OK] net.shipilev.jmm.LockCoarsening
(fork: #1, iteration #1, JVM args: [-server, -XX:+UnlockDiagnosticVMOptions, -XX:+StressLCM,
-XX:+StressGCM])
  Observed state   Occurrences              Expectation  Interpretation
            0, 0    43,558,372               ACCEPTABLE  All other cases are acceptable.
            0, 1        22,512               ACCEPTABLE  All other cases are acceptable.
            1, 0         1,565   ACCEPTABLE_INTERESTING  X and Y are visible in different order
            1, 1     1,372,341               ACCEPTABLE  All other cases are acceptable.
```

"Commit to memory" as a mental model

```scala
var x: Int = 0
var y: Int = 0
```

```scala
// t0
x = 1
```

```scala
// t1
y = 1
```

```scala
// t2
val a = x
val b = y
(a, b)
```

```scala
// t3
val b = y
val a = x
(a, b)
```

"Commit to memory" as a mental model

```
var x: Int = 0
var y: Int = 0
```

```
// t0
x = 1
```

```
// t1
y = 1
```

```
// t2
val a = x
val b = y
(a, b)
```

```
// t3
val b = y
val a = x
(a, b)
```

Possible to read `t2: (x = 1, y = 0)`, `t3: (x = 0, y = 1)`!

"Commit to memory" as a mental model

```
var x: Int = 0
var y: Int = 0
```

```
// t0
x = 1
```

```
// t1
y = 1
```

```
// t2
val a = x
val b = y
(a, b)
```

```
// t3
val b = y
val a = x
(a, b)
```

Possible to read `t2: (x = 1, y = 0), t3: (x = 0, y = 1)`!

Pain: lack of "multi-copy atomicity". If many threads are reading a variable, their history of those reads do not need to be consistent in general. Hardware-dependent!

"Fences" as a mental model

```scala
@volatile var greatBarrierReef: Int = 0
var x: Int = 0
var y: Int = 0
```

```scala
// t0
x = 1
y = 1
greatBarrierReef = 1
```

```scala
// t1
greatBarrierReef = 2
(x, y)
```

"Fences" as a mental model

```scala
@volatile var greatBarrierReef: Int = 0
var x: Int = 0
var y: Int = 0
```

```scala
// t0
x = 1
y = 1
greatBarrierReef = 1
```

```scala
// t1
greatBarrierReef = 2
(x, y)
```

Surely it's not possible to read `x = 0`, `y = 1`...?

"Fences" as a mental model

```scala
@volatile var greatBarrierReef: Int = 0
var x: Int = 0
var y: Int = 0
```

```scala
// t0
x = 1
y = 1
greatBarrierReef = 1
```

```scala
// t1
greatBarrierReef = 2
(x, y)
```

Surely it's not possible to read `x = 0, y = 1`...? Exactly 1 out of 1M 😘

"Separability" as a mental model

If I write ***my*** *code well, I'm good!*

- Separate memory locations you touch
- Carefully analyze critical sections

"Separability" as a mental model

If I write ***my*** *code well, I'm good!*

- Separate memory locations you touch
- Carefully analyze critical sections

"Separability" as a mental model

```
var x: Int = 0 // your territory
var y: Int = 0 // your nailbiting neighbour's
```

```
// t0
y = 1
x = 1
val a = y
val c = x
```

```
// t1
x = 2
y = 2
val b = y
val d = x
```

"Separability" as a mental model

```scala
var x: Int = 0 // your territory
@volatile var y: Int = 0 // neighbour felt unsafe
```

```scala
// t0
y = 1
x = 1
val a = y
val c = x

```

```scala
// t1
x = 2
y = 2
val b = y
val d = x

```

"Separability" as a mental model

```
var x: Int = 0 // your territory
@volatile var y: Int = 0 // neighbour felt unsafe
```

```
// t0
y = 1
x = 1
val a = y
val c = x

```

```
// t1
x = 2
y = 2
val b = y
val d = x

```

You are now implicitly attached to an abstract, and *very fragile*, global state!

Not everything is an `Int`

```
var a: Long = 0L

```

```
// t0
a = 7000000000L

```

```
// t1
a = 8000000000L
println(a)

```

Outputs?

Not everything is an `Int`

```
var a: Long = 0L
```

```
// t0
a = 70000000000L
```

```
// t1
a = 80000000000L
println(a)
```

Outputs? 0?

Not everything is an `Int`

```
var a: Long = 0L
```

```
// t0
a = 70000000000L
```

```
// t1
a = 80000000000L
println(a)
```

Outputs? 0? 70000000000L?

Not everything is an `Int`

```
var a: Long = 0L
```

```
// t0
a = 7000000000L
```

```
// t1
a = 8000000000L
println(a)
```

Outputs? 0? 7000000000L? 8000000000L?

Not everything is an `Int`

```kotlin
var a: Long = 0L

```

```kotlin
// t0
a = 70000000000L

```

```kotlin
// t1
a = 80000000000L
println(a)

```

Outputs? 0? 70000000000L? 80000000000L? 78589934592L . . . ?

Not everything is an `Int`

```
var a: Long = 0L
```

```
// t0
a = 70000000000L
```

```
// t1
a = 80000000000L
println(a)
```

Outputs? 0? 70000000000L? 80000000000L? 78589934592L . . . ?

Other technical baggage: access atomicity v memory ordering and the burden of volatility

Horror Circus: sync on strings

```scala
// t0
"Lock".synchronized {
    x = x + 1
}
```

```scala
// t1
"Lock".synchronized {
    x = x + 1
}
```

More horrific examples on:

Aleksey Shipilev. **Close Encounters of The Java Memory Model Kind.** 2016. URL: https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/

Print everything!

Print everything!

Print everything!


```java
/**
 * Prints an Object and then terminate the line.  This method calls
 * at first String.valueOf(x) to get the printed object's string value
 * then behaves as
 * though it invokes {@link #print(String)} and then
 * {@link #println()}.
 *
 * @param x  The {@code Object} to be printed.
 */
public void println(Object x) {
    String s = String.valueOf(x);
    if (getClass() == PrintStream.class) {
        // need to apply String.valueOf again since first invocation
        // might return null
        writeln(String.valueOf(s));
    } else {
        synchronized (this) {
            print(s);
            newLine();
        }
    }
}
```

Conclusions

- Not much
- Take what you can
- Don't write concurrent code (yourself)

References

[1] Aleksey Shipilev. **Java Memory Model Pragmatics (transcript).** 2014. URL: https://shipilev.net/blog/2014/jmm-pragmatics/.

[2] Aleksey Shipilev. **Close Encounters of The Java Memory Model Kind.** 2016. URL: https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/.