ORACLE

Java Memory Model

...and the pragmatics of it

Aleksey Shipilev
aleksey.shipilev@oracle.com, @shipilev

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Intro

Intro: Detector Slide

Intro: Abstract Machines

- Every programming language describes its semantics via the semantics of the abstract machine executing the source program
- Language spec = abstract machine spec[1]

Very obvious example:
`Brainfuck`[2] is a very straight-forward
assembly language for the Turing Machine

[1] Java $\neq$ Java bytecode $\Rightarrow$ Java spec $\neq$ JVM spec

[2] `http://en.wikipedia.org/wiki/Brainfuck`

Intro: Memory Model

- The significant part of abstract machine specification is the model of machine *storage* = *memory model*
- To serve its purpose, the memory model only needs to answer one simple question:

Intro: Memory Model

- The significant part of abstract machine specification is the model of machine *storage* = *memory model*
- To serve its purpose, the memory model only needs to answer one simple question:

What values can a particular `read` in the program return?

Intro: Sequential programs are easy!

- Program executes sequentially? The memory model is obvious:

> «The reads should see the values written by the latest writes in program order»[3]

- Most people infer that «memory model» really means the «memory model which covers the semantics of multi-threaded programs»

[3] e.g. for C99: ISO/IEC 9899:1999, «5.1.2.3 Program execution»

Intro: ...is not that easy

- Famous C 89/99 example:

```
int i = 5; (.)
i = (++i + ++i); (.)
assert (13 == i); (.) // FAILS
```

- The absence of *sequence points*[4] leads to undefined behavior (implementations are free to do things in between (.) (.))
- Memory models are also needed to reason about single-threaded programs

[4] ISO/IEC 9899:1999, «Annex C: Sequence Points»

Intro: Coming back to reality

Language implementations are doing either of two things:

1. Emulate the abstract machine, and run the source program on that emulation («interpretation»)
2. Specialize the abstract machine for given source program, and run the resulting executable («compilation»)

In **both cases** an implementation
needs to match the semantics of abstract machine.

Translation:
Interpreters are not immune from memory model issues.

Intro: Memory Model is a Trade-Off

How hard it is to use a language?
vs.
How hard it is to build a language implementation?
vs.
How hard it is to build appropriate hardware?

- Sweet new language X can offer tons of juicy features, but will the humanity spend another million years trying to build the high-performance and conforming implementation of it?

Intro: The Logic of the Talk

We frame this talk as following:

1. Express our desires for language semantics
2. Look what is actually available in the real world
3. Understand how spec balances between (1) and (2)
4. See how the conservative implementations work
5. Peek at some other languages

Formal JMM definitions go in the blocks like these

Access atomicity

Access atomicity: Fairy Tale

What do we want?

Access atomicity for all built-in types:

That is, for any built-in `T`:

`T t = V1;`	
`t = V2;`	`T r1 = t;` `assert (r1 ∈ {V1, V2})`

Access atomicity: Reality

Need the hardware support for atomic reads/writes

Caveats:

- The absence of hardware-assisted operations for large reads: how would one read 8-byte `long` on 32-bit x86? 32-bit ARM?
- Memory subsystem requirements: e.g. crossing the cache line usually loses the access atomicity

Access atomicity: Compromise (1/2)

Reads/writes are atomic for everything, except `long` and `double`

`volatile long` and `volatile double` are atomic

- References have the machine bitness
- Almost all HW in 2004 was able to read 32 bits at once, 64 bits read/writes needed the spec relaxation
- Can regain the atomicity (highlighting the performance penalty)

Access atomicity: Compromise (2/2)

Very often the misaligned access loses atomicity
(an almost everywhere loses the performance)

- The implementations are forced to align data:

```
o.o.j.samples.JOLSample_02_Alignment.A[5]
 OFFSET  SIZE  TYPE DESCRIPTION
      0    12       (object header)
     12     4       (alignment/padding gap)
     16     8  long A.f
```

[5] http://openjdk.java.net/projects/code-tools/jol/

Access atomicity: Quiz

What does it print?

```
AtomicLong al = new AtomicLong();
```

`al.set(-1L);`	`println(al.get());`

Access atomicity: Quiz

What does it print?

```
AtomicLong al = new AtomicLong();
```

`al.set(-1L);`	`println(al.get());`

Why not `0 x FFFF FFFF 0000 0000`?

Access atomicity: Quiz

What does it print?

```
AtomicLong al = new AtomicLong();
```

`al.set(-1L);`	`println(al.get());`

Why not `0 x FFFF FFFF 0000 0000`?
No magic involved: «`volatile long`» guarantees atomicity.

Access atomicity: Value types

- Everyone thinks they want to have value types. Among all the benefits, they bring some new memory model issues
- For example, C/C++11 atomics require atomicity for any POD:

```
typedef struct TT {
    int a, b, c, ..., z; // 104 bytes
} T;
std::atomic<T> atomic();
```

`atomic.set(T());`	`T t = atomic.get();`

Access atomicity: Value types

- Everyone thinks they want to have value types. Among all the benefits, they bring some new memory model issues
- For example, C/C++11 atomics require atomicity for any POD:

```
typedef struct TT {
    int a, b, c, ..., z; // 104 bytes
} T;
std::atomic<T> atomic();
```

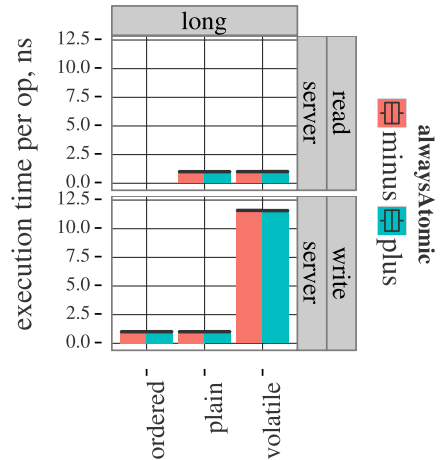
`atomic.set(T());`	`T t = atomic.get();`

- The implementation **is forced** to face the music

Access atomicity: JMM 9

- The exceptions for `long`/`double` were pragmatic in 2004
  - 32-bit x86 everywhere
  - Very simplistic ARMs, and no 64-bit PowerPCs
- It is 2014 now!
  - Are there many 32-bit machines in server world now?
  - Even 32-bit machines have a selection of 64-bit instructions
  - Most of the platforms have de-facto atomic `long`/`double`
  - ...but we require `volatile` anyway, because of WORA
- Q: Is it a good time to purge these exceptions?

Access atomicity: JMM 9



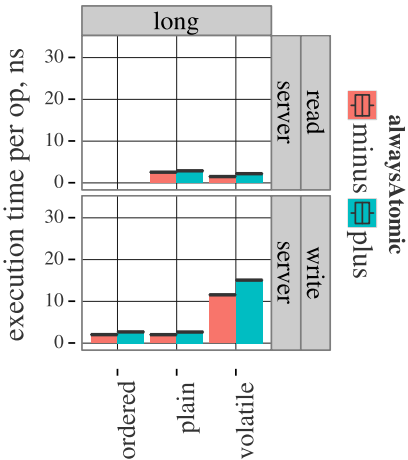
x86, Ivy Bridge, 64-bit:

No difference at all:

- `double` is already atomic
- `long` has native bitness

[6] `http://shipilev.net/blog/2014/all-accesses-are-atomic/`

Access atomicity: JMM 9



x86, Ivy Bridge, 32-bit:

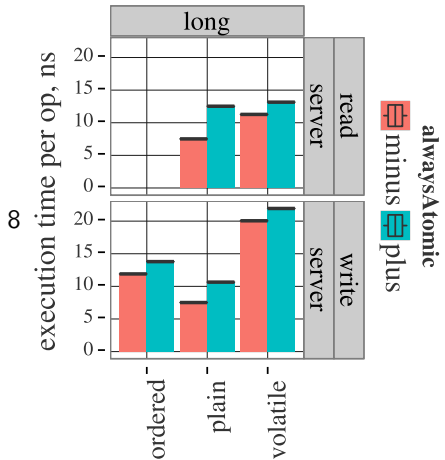
Slight difference:

- `double` is already atomic
- `long` works via vector instructions

[7] `http://shipilev.net/blog/2014/all-accesses-are-atomic/`

Java

Access atomicity: JMM 9



ARMv7, Cortex-A9, 32-bit:

Slight difference:

- `double` is already atomic
- `long` works via vector instructions

[8] `http://shipilev.net/blog/2014/all-accesses-are-atomic/`

Word tearing

Word tearing: Fairy tale

What do we want?

Independence of operations over independent elements (fields, array elements, etc.):

```
T[] as = new T[...]; as[1] = as[2] = V0;
```

`as[1] = V1;` `<term>`	`as[2] = V1;` `<term>`	`<join both>` `T r1 = as[1];` `T r2 = as[2];` `assert (r1 == r2)`

Word tearing: Reality

Need the hardware support for independent reads/writes

Caveats:

- The absence of hardware-assisted read/writes for small types: how would one atomically write the 1-bit `boolean`, if you can only write N ($N \geq 8$) bits?

Word tearing: Compromise

Word tearing is prohibited.

- Most hardware can address 8 bits and up
- If hardware can address as low as N bits, then a **sane** language implementation will have the minimal base type width of N bits
- E.g. on most platforms no built-in Java type loses space (except for 8-bit `boolean`)

Word tearing: Experimental Proof

Objects are aligned by 8 bytes.
Every data type, except for `boolean`,
has the width fitting the value domain:

```
$ java -jar jol-internals.jar ...
 Running 64-bit HotSpot VM.
 Using compressed references with 3-bit shift.
 Objects are 8 bytes aligned.
 Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
 Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
```


Word tearing: Quiz

What does it print?

```
BitSet bs = new BitSet();
```

`bs.set(1);` `<term>`	`bs.set(2);` `<term>`	`<join both>` `println(bs.get(1));` `println(bs.get(2));`

[9] Is there an implementation which can print (F, F)?

Word tearing: Quiz

What does it print?

```
BitSet bs = new BitSet();
```

`bs.set(1);`
 `<term>`
	`bs.set(2);`
	 `<term>`
		`<join both>`
		 `println(bs.get(1));`
		 `println(bs.get(2));`

Any[9] of (T, T), (F, T), (T, F).

[9] Is there an implementation which can print (F, F)?

Word tearing: Bit fields

- Everyone thinks they want a generic way to control the object layout in Java. Control this:

```
typedef struct TT {
    unsigned a:7;
    unsigned b:3;
} T;
```

T t;	
t.a = 42;	r1 = t.b;

Word tearing: Bit fields

- Everyone thinks they want a generic way to control the object layout in Java. Control this:

```
typedef struct TT {
    unsigned a:7;
    unsigned b:3;
} T;
```

T t;	
t.a = 42;	r1 = t.b;

- The implementation **is forced** to face the music on every access to either a or b.

Word tearing: Bit fields

- Everyone thinks they want a generic way to control the object layout in Java. Control this:

```
typedef struct TT {
    unsigned a:7;
    unsigned b:3;
} T;
```

T t;	
t.a = 42;	r1 = t.b;

- The implementation **is forced** to face the music on every access to either a or b. (C/C++11 relaxed this!)

Word tearing: JMM 9

MOVE
ALONG
NOTHING
TO SEE
HERE

SC-DRF

SC-DRF: Fairy Tale

What do we want?

A simple way to reason about correctness.

`opA();`	`opD();`
`opB();`	`opE();`
`opC();`	`opF();`

Very easy to reason if each thread executes in order,
thread executions interleave.

SC-DRF: Fairy Tale (formal)

Sequential Consistency (SC):

(Lamport, 1979): «...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.»

SC-DRF: Fairy Tale (formal)

Sequential Consistency (SC):

(Lamport, 1979): «...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.»

SC-DRF: Fairy Tale (formal)

SC is rather tricky:

- We can change the program in whatever fashion we want, provided there is an execution of the original program which yields SC result

```
int a = 0, b = 0;
```

`a = 1;`	`b = 2;`
`print(b);`	`print(a);`

$\rightarrow$

...

`print(2);`	`print(1);`

SC-DRF: Reality

- The relationship between code transformations and memory model can be expressed via read/write reorderings
- Does this transformation break SC?

```
int a = 0, b = 0;
-----------------
r1 = a;
r2 = b;
```

→

```
int a = 0, b = 0;
-----------------
r2 = b;
r1 = a;
```


SC-DRF: Reality

int a = 0, b = 0;	
r1 = a;	b = 2;
r2 = b;	a = 1;

→

int a = 0, b = 0;	
r2 = b;	
	b = 2;
	a = 1;
r1 = a;	

- Source program executed under SC has either «`r2 = b`» or «`a = 1`» as the last statement, hence `(r1, r2)` is either `(*, 2)` or `(0, *)`.
- Modified program yields `(r1, r2) = (1, 0)`

SC-DRF: Reality

Sequential Consistency is very appealing model.
Somebody submit a JEP already!

- Very hard to tell what transformations are not breaking SC
- *In theory*, some cool and fancy Global MetaOptimizer (GMO) is able to analyse this
- *In practice*, however, both runtimes and hardware are GMO-free $\Rightarrow$ most optimizations are forbidden

SC-DRF: HW Reality

[10] http://en.wikipedia.org/wiki/Memory_ordering

SC-DRF: HW Reality

Hardware speculates and reorders stuff a lot
(for performance!)

Memory ordering in some architectures[2][3]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

10

[10] http://en.wikipedia.org/wiki/Memory_ordering

SC-DRF: a few definitions

- Two memory accesses **conflict**, if they use the same memory location, and at least one of the accesses is write
- The program contains a **data race**, if two memory accesses conflict, and happen simultaneously (i.e. are not ordered by synchronization)

Racy programs yield suprising results!
The language is forced to provide access ordering mechanincs.

SC-DRF: Compromise

Need a weaker model!
(<trade-off rant goes here>)

If we are careful enough:

- Many profitable optimizations are allowed
- Most developers are not suicidal after learning the rules
- The language spec is actually readable

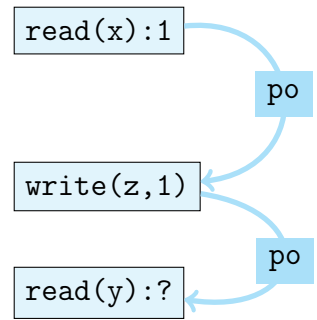
SC-DRF: JMM Formalism TL;DR;

- JMM specifies what *outcomes* are allowed by the language
- JMM defines *actions*. Actions take values with them: e.g. «read(x, 1)» means we actually read «1» from «x». The outcome of the particular program is allowed only if there is an action reading the value which outcome desires
- Actions are aggregated in *executions*, which have orders over actions ($\xrightarrow{po}$, $\xrightarrow{so}$, $\xrightarrow{sw}$, $\xrightarrow{hb}$). Valid execution yields the desired outcome $\Rightarrow$ the outcome is allowed

SC-DRF: Program Order

Program Order (PO) binds the intra-thread actions

```
if (x == 2) {
  y = 1;
} else {
  z = 1;
}
r1 = y;
```

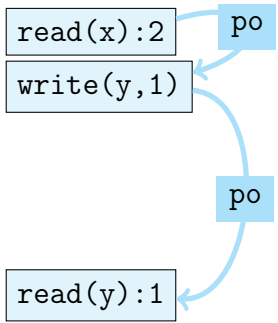


SC-DRF: Program Order

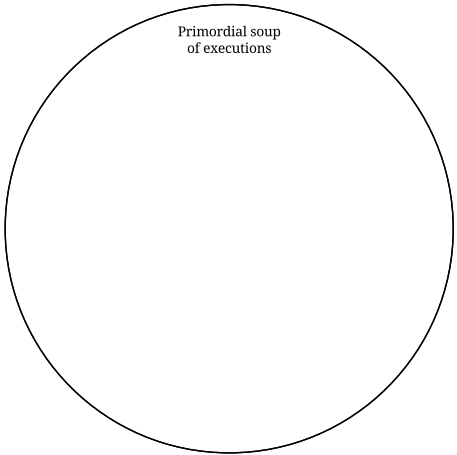
PO is total order

(Note the program statements are **not** in total order!)

```
if (x == 2) {
  y = 1;
} else {
  z = 1;
}
r1 = y;
```



SC-DRF: Towards the viable executions



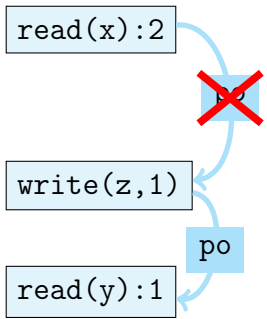
Somewhere in the set of all possible executions may lie the execution which justifies the outcome for the program.

JMM's purpose in life is to figure out if there is such an execution.

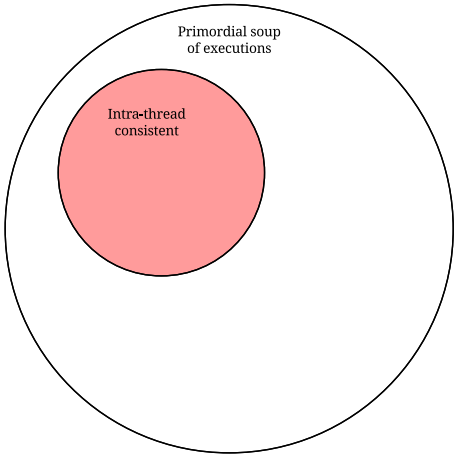
SC-DRF: Program Order

Intra-thread consistency: for each thread, the order of actions in PO is consistent with threads' isolated executions

```
if (x == 2) {
  y = 1;
} else {
  z = 1;
}
r1 = y;
```



SC-DRF: PO constraints



Intra-thread consistency filters out the executions that can be used to reason about the particular program.

This is the only link between JMM and the rest of the language spec.

SC-DRF: Synchronization Actions

Weak memory models do not order all the actions,
only the special actions are ordered.

Synchronization Actions (SA):

- volatile read, volatile write
- lock monitor, unlock monitor
- (synthetic) first and last actions in threads
- actions detecting the thread had terminated (Thread.join(), Thread.isInterrupted(), etc)
- actions that start the thread

SC-DRF: Synchronization Order

Synchronization Actions form the Synchronization Order (SO)

- SO is total order
  - every thread observes SA in same order
  - that's the only cross-thread that needs to be total
- SA order in PO is coherent with SO
  - SA within the single thread are observed in program order
  - lock/unlock invariants are still sound
- **Synchronization order consistency**:
  All reads in SO see the last writes in SO.

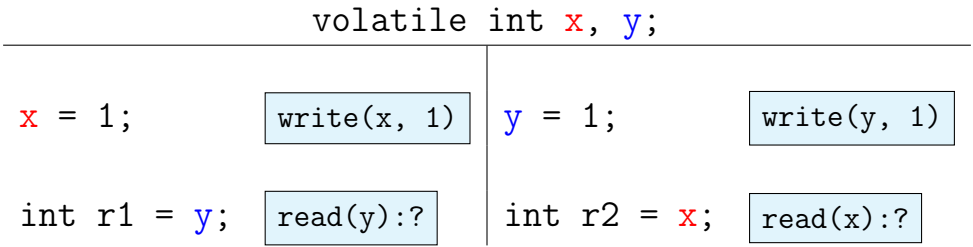
SC-DRF: SO constraints, Dekker example

```
volatile int x, y;
```

`x = 1;`	`y = 1;`
`int r1 = y;`	`int r2 = x;`

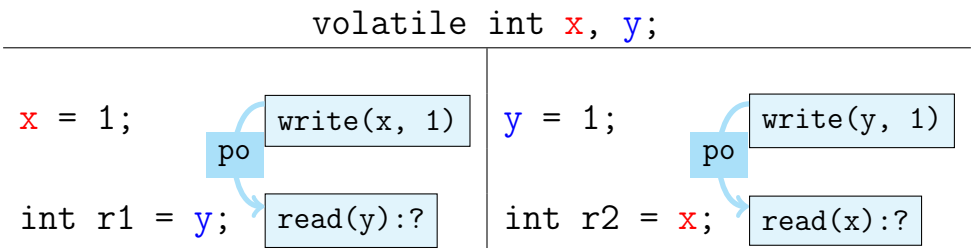
Think about it: what `(r1, r2)` outcomes are allowed?

SC-DRF: SO constraints, Dekker



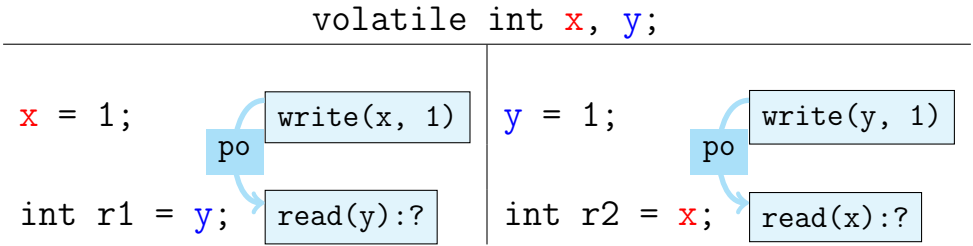
Because of intra-thread consistency, we only consider the executions with these four actions

SC-DRF: SO constraints, Dekker



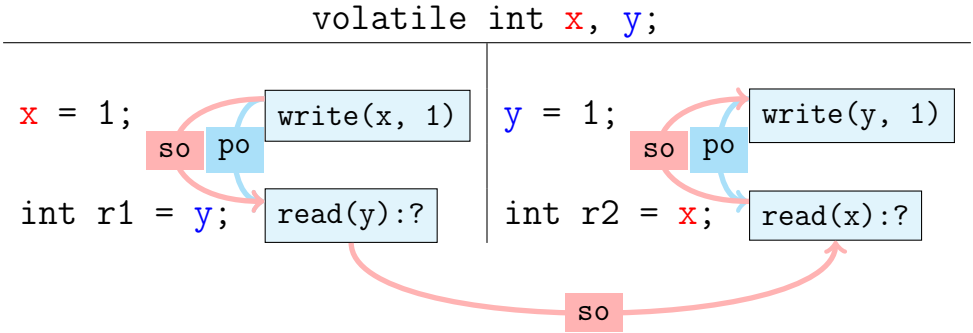
Intra-thread actions are bound by $\xrightarrow{po}$

SC-DRF: SO constraints, Dekker



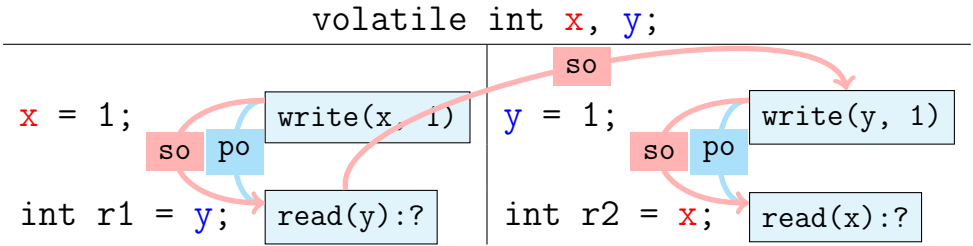
All these actions are over `volatile`, hence they are in SO. However, SO can be laid in a few different ways...

SC-DRF: SO constraints, Dekker



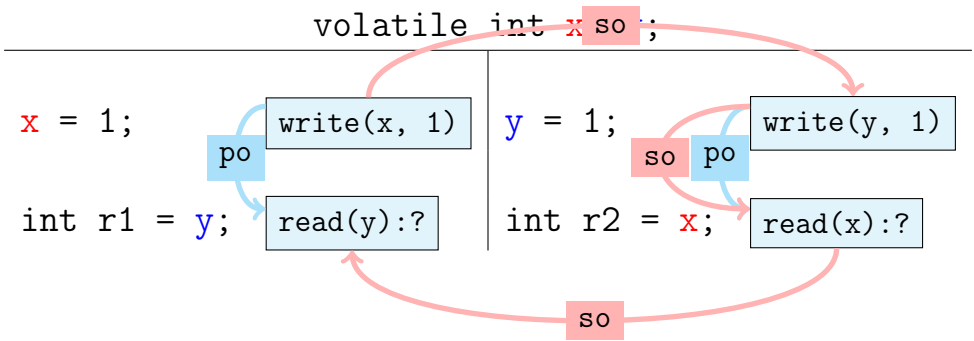
Case 1: $\xrightarrow{so}$ is **NOT** consistent with $\xrightarrow{po}$,
execution should be thrown away

SC-DRF: SO constraints, Dekker



Case 2: $\xrightarrow{so}$ is consistent with $\xrightarrow{po}$, and because of SO consistency the reads are obliged to observe `read(y):0` and `read(x):1`

SC-DRF: SO constraints, Dekker



Case 3: $\xrightarrow{so}$ is consistent with $\xrightarrow{po}$, and because of SO consistency the reads are obliged to observe `read(y):1` and `read(x):1`

SC-DRF: Observation: SA are SC!

Synchronization actions are sequentially consistent!

```
volatile int x, y;
x = 1;          | y = 1;
int r1 = y;     | int r2 = x;
```

- The last action in program order will come last
- Therefore, either `read(y):?` or `read(x):?` will come last, and observe the corresponding write
- Therefore, `(r1, r2) = (0, 0)` is forbidden

SC-DRF: SO constraints, IRIW

Another classic, «Independent Reads of Independent Writes» (IRIW):

```
volatile int x, y;
```

`x = 1;`	`y = 1;`	`int r1 = y;` `int r2 = x;`	`int r3 = x;` `int r4 = y;`

- All executions yielding $(r1, r2, r3, r4) = (1, 0, 1, 0)$ break either SO or SO-PO consistency, and hence forbidden

SC-DRF: SO constraints, IRIW

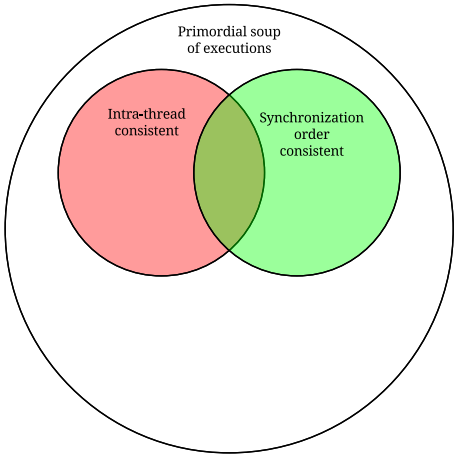
Another classic, «Independent Reads of Independent Writes» (IRIW):

```
volatile int x, y;
```

`x = 1;`	`y = 1;`	`int r1 = y;` `int r2 = x;`	`int r3 = x;` `int r4 = y;`

- All executions yielding $(r1, r2, r3, r4) = (1, 0, 1, 0)$ break either SO or SO-PO consistency, and hence forbidden
- Sprinkle enough `volatiles` around the Java program, and it will eventually turn into sequentially consistent!

SC-DRF: SO constraints



Synchronization order consistency provides the SC skeleton for the program.

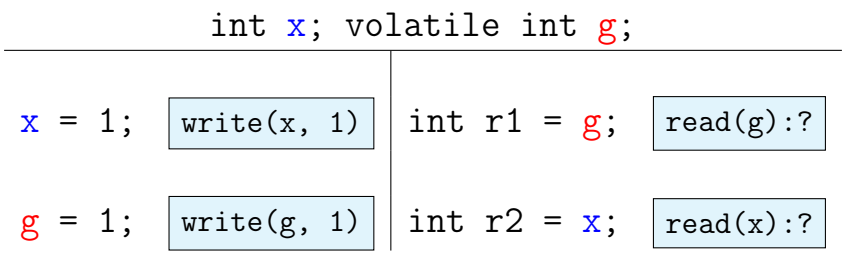
This is the only total ordering of inter-thread actions required by the spec.

SC-DRF: Problems with SO

SO alone is not enough to provide a weaker model:

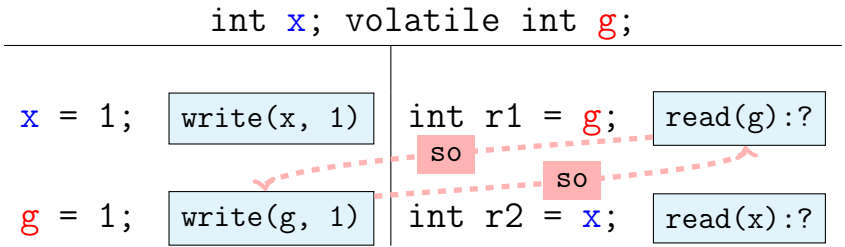
- SO seems «all or nothing»: either you turn all the operations into SA, or you let non-SA operations to float around without constraints, breaking your programs
- Annotating the entire program with `volatile`-s (or locks) turns the program into SC at the expense of optimizations
- Need another weaker order for non-SA operations (Spoiler alert: happens-before)

SC-DRF: HB precursors, publication



Think about it: is (r1, r2) = (1, 0) allowed?

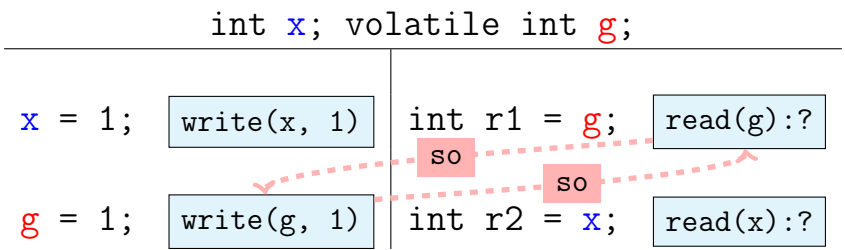
SC-DRF: HB precursors, publication



$\xrightarrow{\text{so}}$ only orders the actions over g!

Yields either `read(g):0`, or `read(g):1`

SC-DRF: HB precursors, publication



There are valid executions either with
`read(x):0`, or with `read(x):1`,
regardless of `read(g):?` result

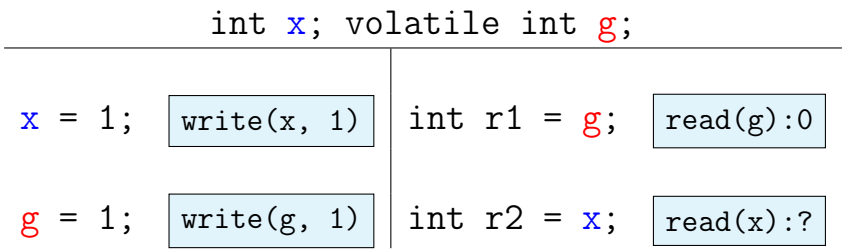
SC-DRF: Synchronizes-With Order (SW)

- PO does not order the actions in the different threads
- Reasoning about inter-thread executions needs something that orders actions across different threads
- So far it was only SO, but SO is total, and using it will impose SC constraints. Therefore, we need some additional partial order:

Synchronizes-With Order (SW):

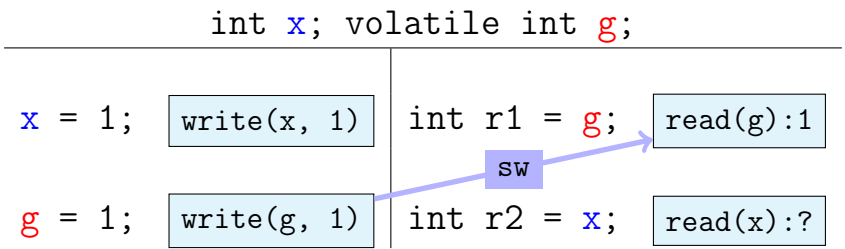
SO suborder, constrained for
concrete reads/writes, locks/unlocks, etc.

SC-DRF: Synchronizes-With (SW)



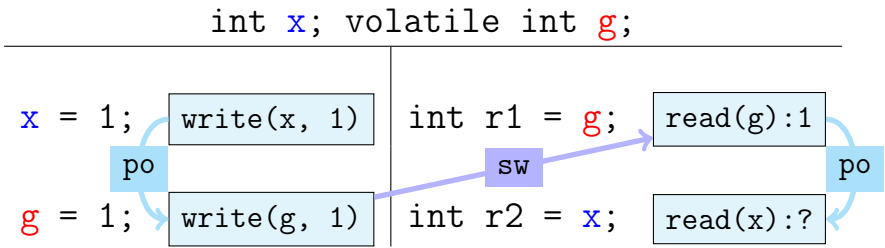
Most SA are not bound in $\xrightarrow{\text{SW}}$

SC-DRF: Synchronizes-With (SW)



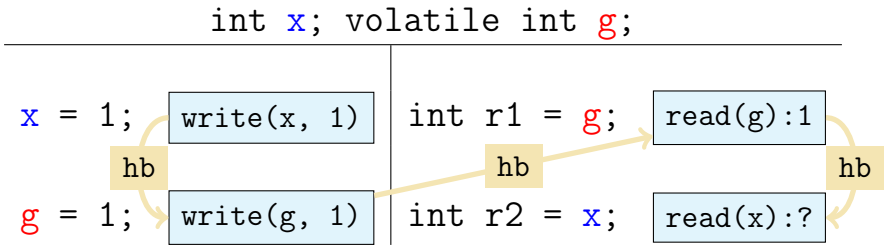
If one SA sees the other, then they are bound in $\xrightarrow{sw}$.
This gives «inter-thread semantics».

SC-DRF: Synchronizes-With (SW)



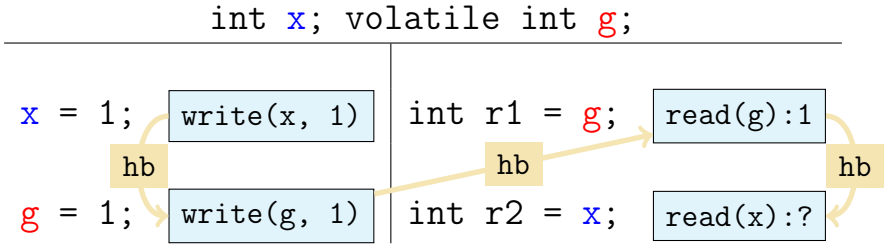
Add $\xrightarrow{\text{po}}$ for remaining actions.
This gives «intra-thread semantics».

SC-DRF: Happens-before (HB)



$\xrightarrow{hb}$ = transitive closure over union of $\xrightarrow{po}$ and $\xrightarrow{sw}$

SC-DRF: Happens-before (HB)



HB consistency: the reads observe the immediately preceeding write in $\xrightarrow{\text{hb}}$, or something else via the race

SC-DRF: Happens-before (HB)

Let: $W(r)$ be the write observed by r, and A be the set of all program actions. Then happens-before consistency:

$$\forall r \in Reads(A) : \neg(r \xrightarrow{\text{hb}} W(r)) \wedge$$
$$\neg(\exists w \in Writes(A) : (W(r) \xrightarrow{\text{hb}} w) \wedge (w \xrightarrow{\text{hb}} r))$$

SC-DRF: Happens-before (HB)

Let: $W(r)$ be the write observed by r, and A be the set of all program actions. Then happens-before consistency:

$$\forall r \in Reads(A) : \neg(r \xrightarrow{\text{hb}} W(r)) \wedge$$

$$\neg(\exists w \in Writes(A) : (W(r) \xrightarrow{\text{hb}} w) \wedge (w \xrightarrow{\text{hb}} r))$$

Either $W(r)$ not ordered with r (race), or $W(r) \xrightarrow{\text{hb}} r$

SC-DRF: Happens-before (HB)

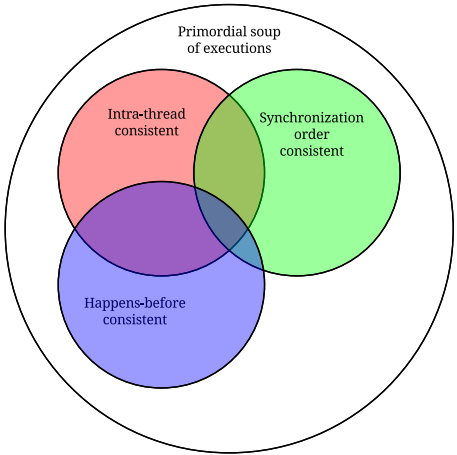
Let: $W(r)$ be the write observed by r, and A be the set of all program actions. Then happens-before consistency:

$$\forall r \in Reads(A) : \neg(r \xrightarrow{\text{hb}} W(r)) \wedge$$

$$\neg(\exists w \in Writes(A) : (W(r) \xrightarrow{\text{hb}} w) \wedge (w \xrightarrow{\text{hb}} r))$$

There are no intervening writes (only care if $W(r) \xrightarrow{\text{hb}} r$)

SC-DRF: HB constraints



Happens-before consistency allows to order the ordinary operations across the threads.

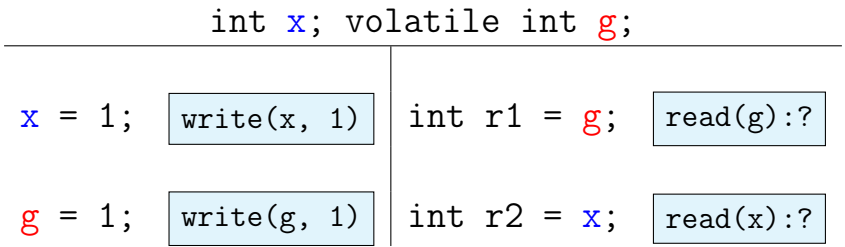
HB makes sense only for the same variable.

SC-DRF: Definition

SequentialConsistency-DataRaceFree:
«Correctly synchronized programs
have sequentially consistent semantics»

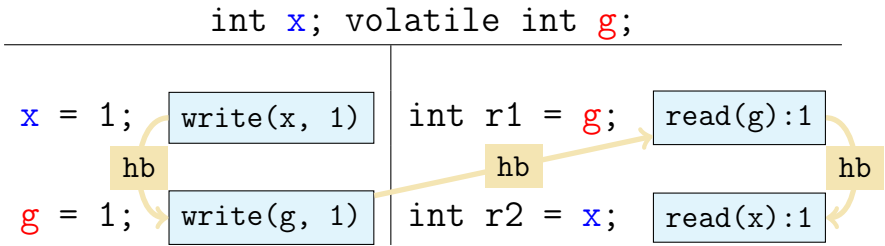
- Translation: No races ⇒ All reads see properly ordered writes ⇒ the outcome can be explained by some SC execution
- Intuition #1: Local operations (almost) always have the outcomes explainable by SC
- Intuition #2: Operations over global data are synchronized with SW-inducing primitives, and are SC

SC-DRF: HB, Publish



Let's analyse with HB consistency rules...

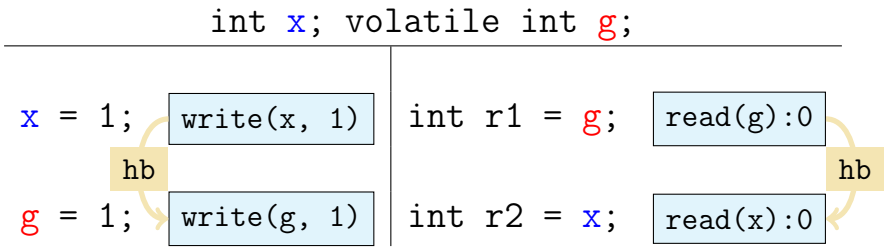
SC-DRF: HB, Publish



Case 1: HB consistent, observe the latest write in $\xrightarrow{hb}$

$$(r1, r2) = (1, 1)$$

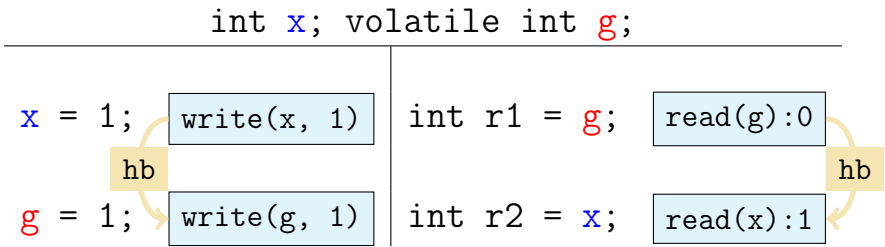
SC-DRF: HB, Publish



Case 2: HB consistent, observe the default value

$$(r1, r2) = (0, 0)$$

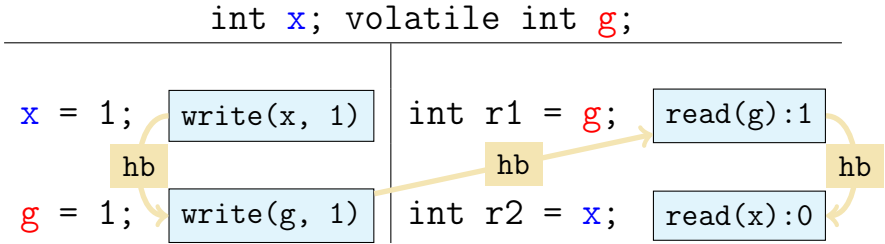
SC-DRF: HB, Publish



Case 3: HB consistent (!), reading via race!

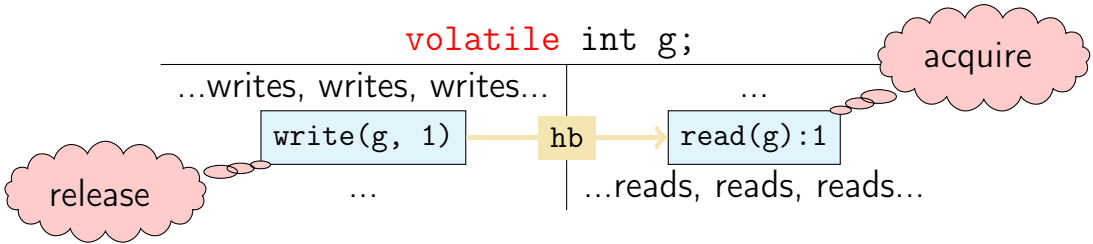
$(r1, r2) = (0, 1)$

SC-DRF: HB, Publish



Case 4: HB **in**consistent, execution can be thrown away

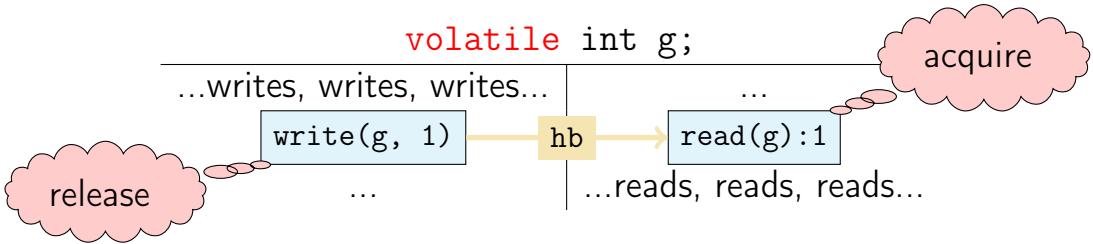
SC-DRF: Publication



Previous example can be generalized as «safe publication»:

- Works only on the same variable

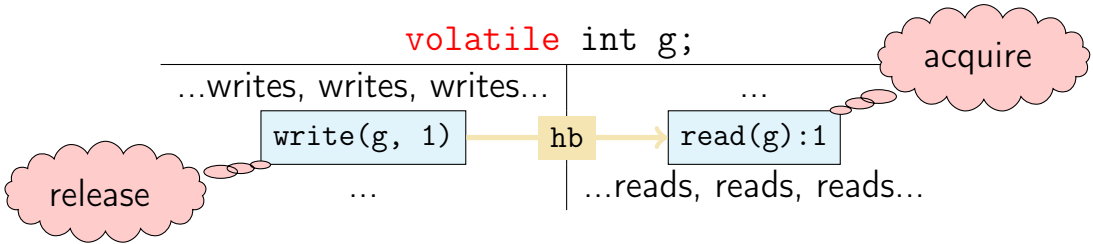
SC-DRF: Publication



Previous example can be generalized as «safe publication»:

- Works only on the same variable
- Works only if we *observed* the `release`-store

SC-DRF: Publication



Previous example can be generalized as «safe publication»:

- Works only on the same variable
- Works only if we *observed* the `release`-store
- Always paired! You can't make one-sided `release`, without doing `acquire` on other side

SC-DRF: Quiz

CR 9234251: Optimize getter in C.get()

```
class C<T> {
  T val;
  public synchronized void set(T v) {
    if (val == null) { val = v; }
  }
  public synchronized T get() {
    // TODO FIXME PLEASE PLEASE PLEASE:
    // THIS ONE IS TOO HOT IN PROFILER!!!111ONEONEONE
    return val;
  }
}
```


SC-DRF: Quiz

RFR (XS) CR 9234251: Optimize getter in C.get():

```
class C<T> {
  T val;
  public synchronized void set(T v) {
    if (val == null) { val = v; }
  }
  public T get() {
    // This one is safe without the synchronization.
    //   (Yours truly, CERTIFIED SENIOR JAVA DEVELOPER)
    return val;
  }
}
```


SC-DRF: Quiz

RFR (XS) CR 9234251: Optimize getter in C.get():

```
class C<T> {
  static volatile int BARRIER; int sink;
  T val;
  public synchronized void set(T v) {
    if (val == null) { val = v; }
  }
  public T get() {
    sink = BARRIER; // acquire membar
    // Obviously, we need a memory barrier here!
    //   (Yours truly, SUPER COMPILER GURU)
    return val;
  }
}
```


SC-DRF: Quiz

That's better: get back SW edge, get back HB.

```
class C<T> {
  volatile T val;
  public synchronized void set(T v) {
    if (val == null) { val = v; }
  }
  public T get() {
    // This one is safe without the synchronization.
    // <Sigh>. Now it's safe.
    //  ($PROJECT techlead, overseeing certified idiots)
    return val;
  }
}
```

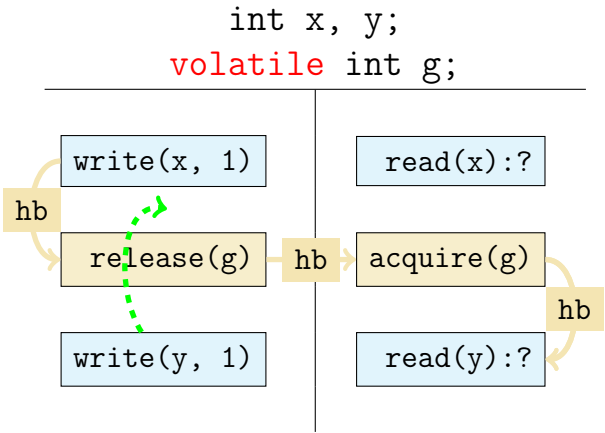

ENTR'ACTE.
Coming back in 10 minutes.

SC-DRF: Roach Motel

One interpretation of the model allows for a simple class of optimizations, «Roach Motel»

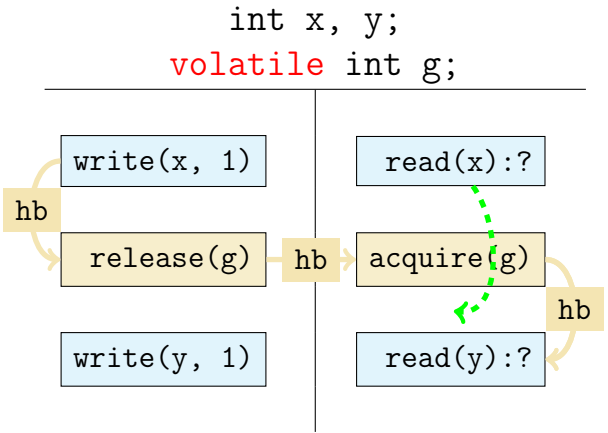
Java

SC-DRF: Roach Motel



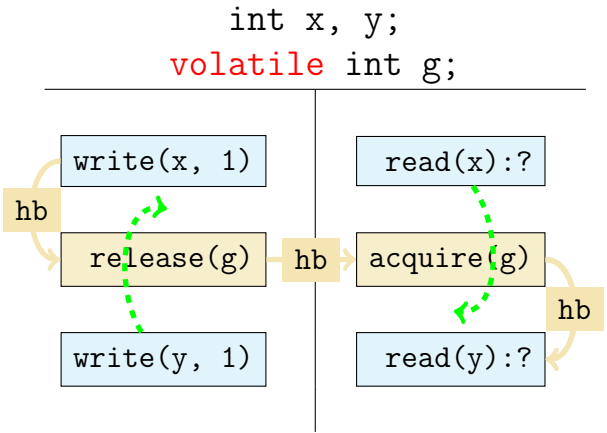
`write(y, 1)` can be reordered before `release`, since it does not break dependencies for x, and `read(y):?` on the right can see that store via the race anyway

SC-DRF: Roach Motel



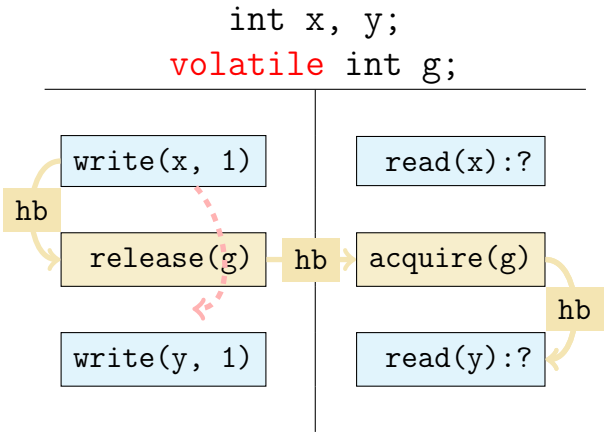
`read(x):?` can be reordered after `acquire`, since it can observe `write(x, 1)` via the race

SC-DRF: Roach Motel



Therefore, «reorderable after `acquire`» + «reorderable before `release`» = «movable into `acquire`+`release` blocks» ⇒ lock coarsening is working

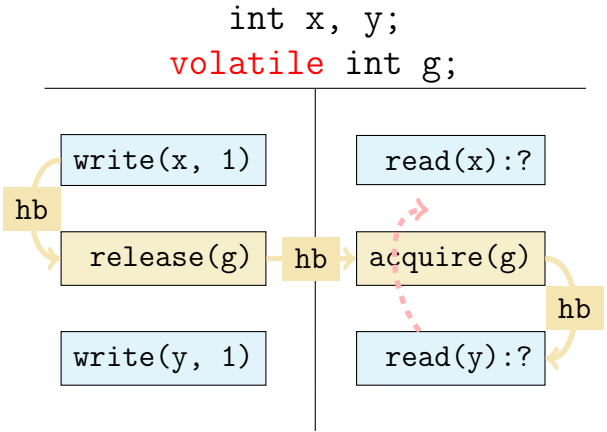
SC-DRF: Roach Motel



`write(x, 1)` can not be easily reordered after `release`, since we move it out from $\xrightarrow{\text{hb}}$. Conservative implementation has no idea if there is read of x, which should see it. GMO is able to use global analysis, and make this reordering.

Java

SC-DRF: Roach Motel



`read(y):?` can not be easily reordered before `acquire`, since we move it out from $\xrightarrow{hb}$. Conservative implementation has no idea if there is a store which it should observe. GMO is able to use global analysis, and make this reordering.

SC-DRF: Quiz

What does it print? Possible answers: 0, 41, 42, 43, <nothing>

```
                int a = 0;
volatile boolean ready = false;
```

`a = 41;`	`while(!ready) {};`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

SC-DRF: Quiz

What does it print? Possible answers: 0, 41, 42, 43, <nothing>

```
                int a = 0;
volatile boolean ready = false;
```

`a = 41;`	`while(!ready) {};`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

Prints either 42 (latest in HB), or 43 (race).

SC-DRF: Quiz #2

What does it print? Possible answers: 0, 41, 42, 43, <nothing>

```
int a = 0;
boolean ready = false;
```

`a = 41;`	`while(!ready) {};`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

SC-DRF: Quiz #2

What does it print? Possible answers: 0, 41, 42, 43, <nothing>

```
int a = 0;
boolean ready = false;
```

`a = 41;`	`while(!ready) {};`
`a = 42;`	`println(a);`
`ready = true;`	
`a = 43;`	

Every answer is possible (race, race, race)

SC-DRF: A few benchmarks

`https://github.com/shipilev/jmm-benchmarks/`

- 2x12x2 Xeon E5-2697, 2.70GHz
- OEL 6, JDK 7u40, x86_64
- We can only measure the performance of some implementation, not the spec itself

SC-DRF: A few benchmarks

`https://github.com/shipilev/jmm-benchmarks/`

- 2x12x2 Xeon E5-2697, 2.70GHz
- OEL 6, JDK 7u40, x86_64
- We can only measure the performance of some implementation, not the spec itself

Java

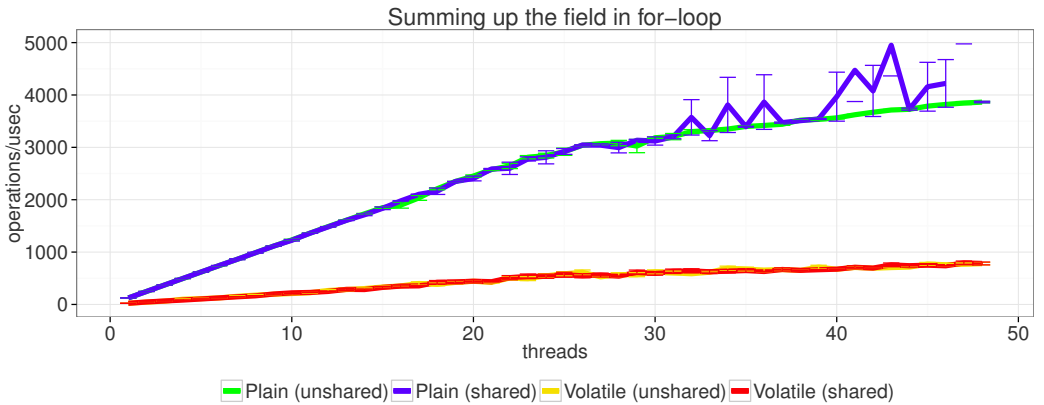
SC-DRF: Hoisting

```
@State(Scope.(Benchmark|Thread))
public static class Storage {
    private (volatile) int v = 42;
}

@Benchmark
public int test(Storage s) {
    int sum = 0;
    for (int c = 0; c < s.v; c++) {
        sum += s.v;
    }
    return sum;
}
```


SC-DRF: Hoisting

It is not `volatile` that is scary, but broken optimizations:



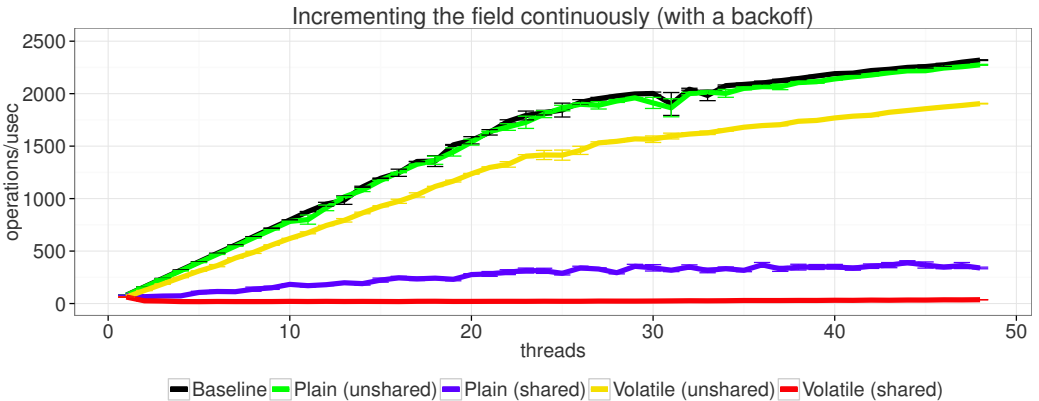
SC-DRF: Writes

```
@State(Scope.(Benchmark|Thread))
public static class Storage {
    private (volatile) int v = 42;
}

@Benchmark
public int test(Storage s) {
    Blackhole.consumeCPU(8); // ~15ns
    return s.v++;
}
```


SC-DRF: Writes

It is not `volatile` that is scary, but data sharing:



SC-DRF: JMM 9

- SC-DRF is agreed to be the successful model
  - Formally known since 1990s
  - Java adopted in 2004
  - C/C++ adopted in 2011
- In some cases, SC is very expensive
  - Ex: PowerPC + IRIW = kills some kittens
  - Ex: Linux Kernel RCU = SC relaxations for ARM/PowerPC make large performance increases ...and arguably without nasty drawbacks
- Q: Can we relax SC-DRF without obliterating the model?

OoTA

OoTA: Fairy Tale

«SC-DRF. All you need is love»

- Local code transformations are allowed until we hit the synchronization primitive
- Local code transformations are playing with synchronizations by some non-breaking rules (e.g. «roach motel»)
- If local transform messed with conflicting accesses, then there was a race, and the user gets what was coming to him!

OoTA: Reality

But there are cases when local transforms break SC.

```
int a = 0, b = 0;
```

`r1 = a;`	`r2 = b;`
`if (r1 != 0)`	`if (r2 != 0)`
`  b = 42;`	`  a = 42;`

Correctly synchronized:
all SC executions have no races.
The only possible result is `(r1, r2) = (0, 0)`.

OoTA: Optimizations

Let's get a splice of speculative optimizations:
Why wouldn't we unconditionally store to b, and then conditionally rollback, if something changed?

```
int a = 0, b = 0;
```

```
int r1 = a;
if (r1 != 0)
   b = 42;
```

OoTA: Optimizations

Let's get a splice of speculative optimizations:
Why wouldn't we unconditionally store to b, and then conditionally rollback, if something changed?

```
int a = 0, b = 0;
```

```
int r1 = a;
if (r1 != 0)
   b = 42;
```

```
int r1 = a;
b = 42;
if (r1 == 0)
  b = 0;
```


OoTA: Optimizations

Let's get a splice of speculative optimizations:
Why wouldn't we unconditionally store to b, and then conditionally rollback, if something changed?

```
int a = 0, b = 0;
```

```
int r1 = a;
if (r1 != 0)
   b = 42;
```

```
int r1 = a;
b = 42;
if (r1 == 0)
  b = 0;
```

```
b = 42;
int r1 = a;
if (r1 == 0)
  b = 0;
```

OoTA: Oedipus closure

```
int a = 0, b = 0;
```

`b = 42;`	
	`r2 = b;`
	`if (r2 != 0)`
	`  a = 42;`
`r1 = a;`	
`if (r1 == 0)`	
`  b = 0;`	

- Yields `(r1, r2) = (42, 42)`
- In the presence of races, the speculation may turn itself into the self-justifying prophecy!

OoTA: Out of Thin Air values

JLS TL;DR: Out of Thin Air values are forbidden

- If we read some value, then somebody else had written that for us before
- JLS 17.4.8 makes a very complicated part of spec to give substance for that «before» thing = «causality requirements»
- JMM defines the special process to validate the executions via committing the actions from the executions

OoTA: Commit semantics

17.4.8 Executions and Causality Requirements

We use $f|_d$ to denote the function given by restricting the domain of f to d. For all x in d, $f|_d(x) = f(x)$, and for all x not in d, $f|_d(x)$ is undefined.

We use $p|_d$ to represent the restriction of the partial order p to the elements in d. For all x,y in d, $p(x,y)$ if and only if $p|_d(x,y)$. If either x or y are not in d, then it is not the case that $p|_d(x,y)$.

A well-formed execution $E = < P, A, po, so, W, V, sw, hb >$ is validated by *committing* actions from A. If all of the actions in A can be committed, then the execution satisfies the causality requirements of the Java programming language memory model.

Starting with the empty set as C_0, we perform a sequence of steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E containing C_i that meets certain conditions.

Formally, an execution *E satisfies the causality requirements of the Java programming language memory model* if and only if there exist:

- Sets of actions C_0, C_1, ... such that:
  - C_0 is the empty set
  - C_i is a proper subset of C_{i+1}
  - $A = \cup (C_0, C_1, ...)$

  If A is finite, then the sequence C_0, C_1, ... will be finite, ending in a set $C_n = A$.

  If A is infinite, then the sequence C_0, C_1, ... may be infinite, and it must be the case that the union of all elements of this infinite sequence is equal to A.
- Well-formed executions E_1, ..., where $E_i = < P, A_i, po_i, so_i, W_i, V_i, sw_i, hb_i >$.

Given these sets of actions C_0, ... and executions E_1, ... , every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally:

1. C_i is a subset of A_i
2. $hb_i|_{C_i} = hb|_{C_i}$
3. $so_i|_{C_i} = so|_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E. Only the reads in C_{i-1} need to see the same writes in E_i as in E. Formally:

4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

All reads in E_i that are not in C_{i-1} must see writes that happen-before them. Each read r in C_i - C_{i-1} must see writes in C_{i-1} in both E_i and E, but may see a different write in E_i from the one it sees in E. Formally:

6. For any read r in A_i - C_{i-1}, we have $hb_i(W_i(r), r)$
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}

Given a set of sufficient synchronizes-with edges for E_i, if there is a release-acquire pair that happens-before (§17.4.5) an action you are committing, then that pair must be present in all E_j, where $j \geq i$. Formally:

8. Let ssw_i be the sw_i edges that are also in the transitive reduction of hb_i but not in po. We call ssw_i the *sufficient synchronizes-with edges for* E_i. If $ssw_i(x, y)$ and $hb_i(y, z)$ and z in C_i, then $sw_j(x, y)$ for all $j \geq i$.

   If an action y is committed, all external actions that happen-before y are also committed.
9. If y is in C_i, x is an external action and $hb_i(x, y)$, then x in C_i.

OoTA: Commit semantics

17.4.8 Executions and Causality Requirements

We use $f|_d$ to denote the function given by restricting the domain of f to d. For all x in d, $f|_d(x) = f(x)$, and for all x not in d, $f|_d(x)$ is undefined.

We use $p|_d$ to represent the restriction of the partial order p to the elements in d. For all x,y in d, $p(x,y)$ if and only if $p|_d(x,y)$. If either x or y are not in d, then it is not the case that $p|_d(x,y)$.

A well-formed execution $E = < P, A, po, so, W, V, sw, hb >$ is validated by *committing* actions from A. If all of the actions in A can be committed, then the execution satisfies the causality requirements of the Java programming language memory model.

Starting with the empty set as C_0, we perform a sequence of steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E containing C_i that meets certain conditions.

Formally, an execution *E satisfies* [illegible] *ents of the Java programming language memory mode*

- Sets of actions C_0, C_1, ... such tha
  - C_0 is the empty set
  - C_i is a proper subset of C_{i+1}
  - $A = \cup (C_0, C_1, ...)$

  If A is finite, then the seq [illegible] et $C_n = A$.

  If A is infi [illegible] ust be the [illegible] ual to A.

  [illegible] V_i, sw_i, hb_i >.

Given these sets of actions C_0, ... and executions E_1, ... , every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally:

1. C_i is a subset of A_i
2. $hb_i|_{C_i} = hb|_{C_i}$
3. $so_i|_{C_i} = so|_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E. Only the reads in C_{i-1} need to see the same writes in E_i as in E. Formally:

4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

All reads in E_i that are not in C_{i-1} must see writes that happen-before them. Each read r in C_i - C_{i-1} must see writes in C_{i-1} in both E_i and E, but may see a different write in E_i from the one it sees in E. Formally:

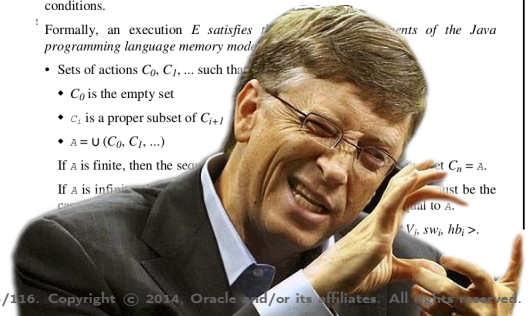
6. For any read r in A_i - C_{i-1}, we have $hb_i(W_i(r), r)$
7. For any read r in (C_i - C_{i-1}), we have $W_i(r)$ in C_{i-1} and $W(r)$ in C_{i-1}

Given a set of sufficient synchronizes-with edges for E_i, if there is a release-acquire pair that happens-before (§17.4.5) an action you are committing, then that pair must be present in all E_j, where $j \geq i$. Formally:

8. Let ssw_i be the sw_i edges that are also in the transitive reduction of hb_i but not in po. We call ssw_i the *sufficient synchronizes-with edges for* E_i. If $ssw_i(x, y)$ and $hb_i(y, z)$ and z in C_i, then $sw_j(x, y)$ for all $j \geq i$.

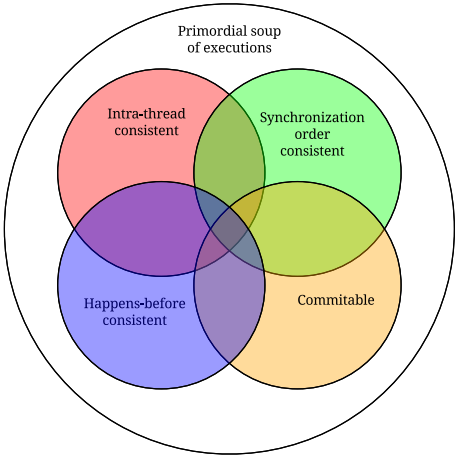
   If an action y is committed, all external actions that happen-before y are also committed.

9. If y is in C_i, x is an external action and $hb_i(x, y)$, then x in C_i.



Java

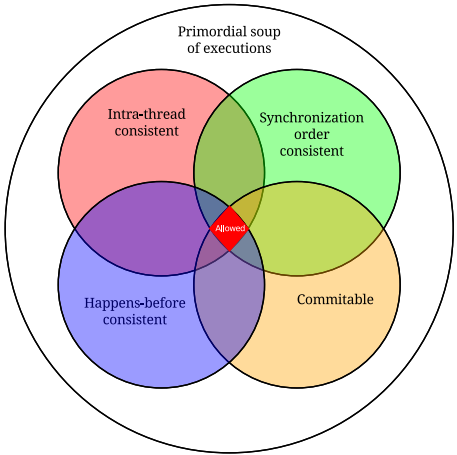
OoTA: Commit semantics



Commit semantics does the final checks for the executions in order to prevent causality violations.

This is needed to prevent Out of Thin Air values.

OoTA: ...Land of Mordor where the Shadows lie



The executions which passed all the checks are the executions we can use to derive the outcomes from.

We can filter out the executions early if they are not meeting at least one the checks.

OoTA: C/C++11

Rigorously specifying OoTA is a mammoth task.
(C/C++11 eventually gave up)

- Makes some easy specification choices (Pyrrhic victory?)
- C/C++1x WG is searching for the way to formally forbid speculative optimizations giving rise to OoTA

OoTA: JMM 9

We seem to be having three options:

1. Continue as we usual: try to simplify/fix the formal spec to aid automatic checkers and humans as well
2. Conservatively forbid the speculative stores: that would mean `LoadStore` before each store
3. Give up, and ask implementations to be «good»

Finals

Finals: Quiz

What does it print?

```
class A {
   int f;
   A() { f = 42; }
}

A a;
```

`a = new A();`	`if (a != null)` `  println(a.f);`

Finals: Quiz

What does it print?

```
class A {
   int f;
   A() { f = 42; }
}

A a;
```

`a = new A();`	`if (a != null)` `  println(a.f);`

<nothing>, 0, 42, or throws NPE.

Finals: Quiz

This one does not throw NPE:

```
class A {
    int f;
    A() { f = 42; }
}

A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Finals: Fairy Tale

We would like to get only «42»:

```
class A {
   ????? int f;
   A() { f = 42; }
}

A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Finals: Fairy Tale

We want to have objects which are safe to publish via races.

- ...so that security would not depend on some (malicious) moron publishing the instance of our otherwise protected class via race
- ...so that we can skip some of the «excess» synchronization actions for immutable objects

Finals: Pragmatics

Final field guarantees are somewhat easy to enforce

- It is usually enough to order the `final` fields initializations and the publishing of the instance. May require memory barriers.
- All known industrial architectures[11] do not reorder the load depending on another load.

[11] Alpha's dead, baby, Alpha's dead

Finals: Formally

There is a «freeze action» at the end of constructor.

Freeze action «freezes» the field values

- If a thread reads the reference to new object with final fields, then it will always observe the frozen values
- If a thread reads the reference to some other object through the `final` field, then its state is at least as fresh as it was at the moment of freeze

Finals: Formally

$$(w \xrightarrow{\text{hb}} F \xrightarrow{\text{hb}} a \xrightarrow{\text{mc}} r1 \xrightarrow{\text{dr}} r2) \Rightarrow (w \xrightarrow{\text{hb}} r2),$$

w – target field write, F – freeze action, a – some action (not the final field read), $r1$ – final field read, $r2$ – target field read

Introduce two new partial orders:

- *dereference order (dr)* (access chains within the thread)
- *memory order (mc)* (access chains within/across the threads)

Finals: Formally

$$\boxed{(w \xrightarrow{\text{hb}} F \xrightarrow{\text{hb}} a \xrightarrow{\text{mc}} r1 \xrightarrow{\text{dr}} r2) \Rightarrow (w \xrightarrow{\text{hb}} r2)},$$

w – target field write, F – freeze action, a – some action (not the final field read), $r1$ – final field read, $r2$ – target field read

If there is only a path via this chain of $\xrightarrow{\text{hb}}$, $\xrightarrow{\text{dr}}$ and $\xrightarrow{\text{mc}}$, then we can only observe the frozen value. But if there are other paths, it is (probably) a racy read, and we can observe something else.

Finals: Example[12]

Thread 1

```
T l = new T() {{
  fx = 42; w
}}; f
GLOBAL = l; a
```

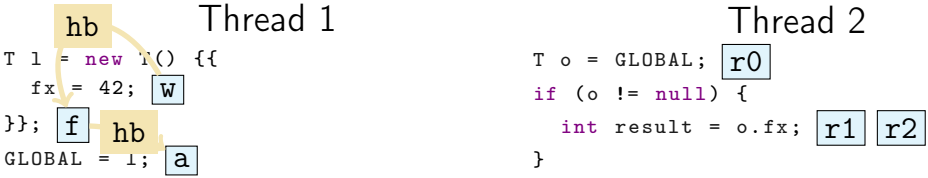
Thread 2

```
T o = GLOBAL; r0
if (o != null) {
  int result = o.fx; r1 r2
}
```

Can we get `result = 0`?

[12] Courtesy Vladimir Sitnikov and Valentin Kovalenko

Finals: Example[12]

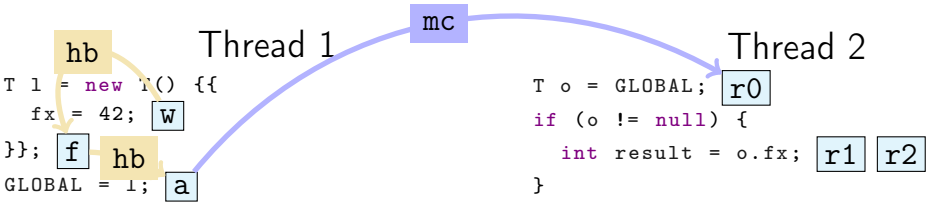


Inter-thread actions induce happens-before:

$$w \xrightarrow{\text{hb}} f,\ f \xrightarrow{\text{hb}} a$$

[12] Courtesy Vladimir Sitnikov and Valentin Kovalenko

Finals: Example[12]

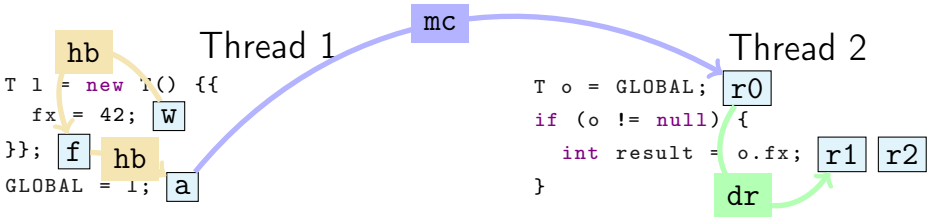


$r0$ observes write a:

$$a \xrightarrow{\text{mc}} r0$$

[12] Courtesy Vladimir Sitnikov and Valentin Kovalenko

Finals: Example[12]

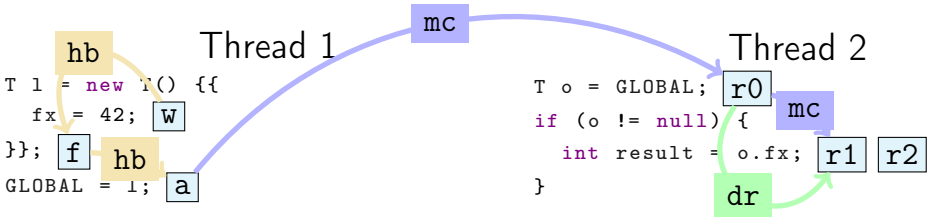


Thread 2 did not create the object, $r1$ reads the object's field, but $r0$ is the only action which reads object address, therefore we have dereference chain:

$$r0 \xrightarrow{\text{dr}} r1$$

[12] Courtesy Vladimir Sitnikov and Valentin Kovalenko

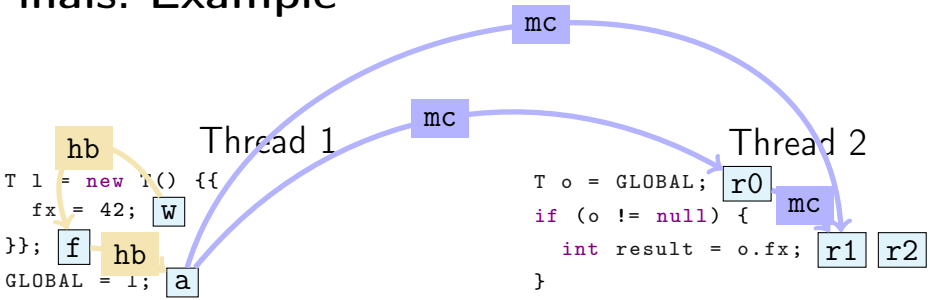
Finals: Example[12]



$$r0 \xrightarrow{dr} r1 \Rightarrow r0 \xrightarrow{mc} r1$$

[12] Courtesy Vladimir Sitnikov and Valentin Kovalenko

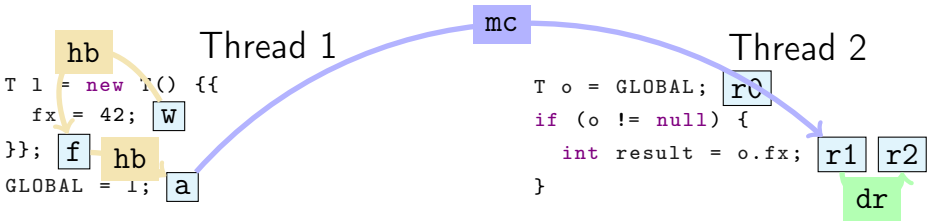
Finals: Example[12]



$a \xrightarrow{\text{mc}} r1$ (by MC transitivity)

[12] Courtesy Vladimir Sitnikov and Valentin Kovalenko

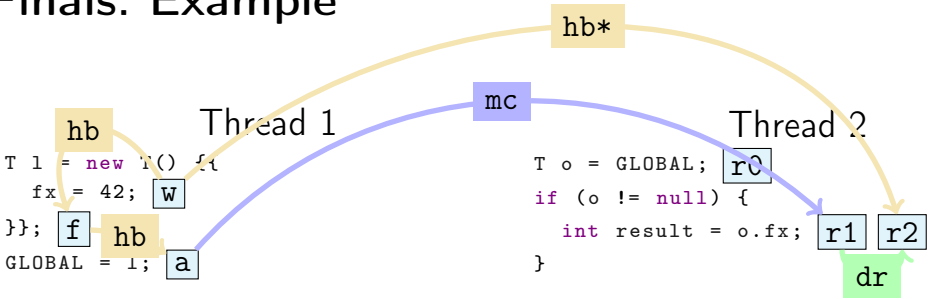
Finals: Example[12]



Let $r2 = r1$, then $r1 \xrightarrow{dr} r2$ (by DR reflectivity)

[12] Courtesy Vladimir Sitnikov and Valentin Kovalenko

Finals: Example[12]

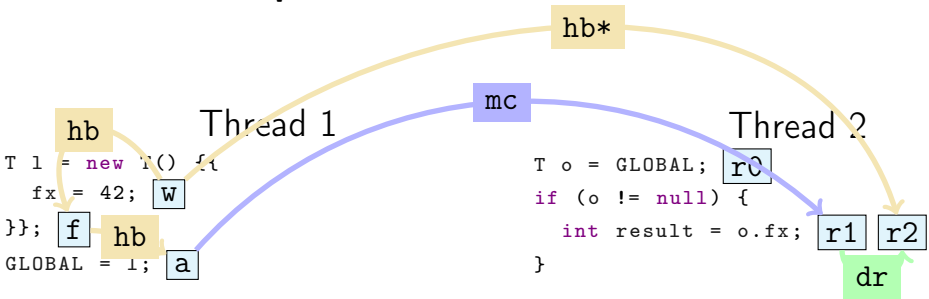


Found everything for HB^*:

$$w \xrightarrow{hb} f \xrightarrow{hb} a \xrightarrow{mc} r1 \xrightarrow{dr} r2 \Rightarrow w \xrightarrow{hb} r2$$

[12] Courtesy Vladimir Sitnikov and Valentin Kovalenko

Finals: Example[12]



$$(w \xrightarrow{hb} r2) \Rightarrow \mathrm{r} \in \{42\}$$

[12] Courtesy Vladimir Sitnikov and Valentin Kovalenko

Finals: Pragmatics

All bets are off with premature publication:

`T p, q;`

`T t1 = <new>`	`T t2 = p`	`T t4 = q`
`t1.f = 42`	`r2 = t2.f`	`r4 = t4.f`
`p = t1`	`T t3 = q`	
`<freeze t1.f>`	`r3 = t3.f`	
`q = t1`		

$\texttt{r4} \in \{42\}$;
however `r2`, `r3` $\in \{0, 42\}$, because `p` had «leaked».

Finals: Pragmatics

`final` fields are cacheable![13]

- «All references are created equal»: do not have to track complete/incomplete initializations
- As soon as an optimizer discovered the `final` field, it can cache its value
- If an optimizer saw the under-initialized object, we are screwed!

[13] Well, not really.

Finals: Quiz

What does it print?

```
class A {
   final int f;
   { f = 42; }
}
A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Finals: Quiz

What does it print?

```
class A {
   final int f;
   { f = 42; }
}
A a;
```

`a = new A();`	`A ta = a;`
	`if (ta != null)`
	`  println(ta.f);`

Of course, either 42 or <nothing>.

Finals: JMM 9

In current spec, `final` is somewhat harsh:

- What if a field was initialized in constructor, and never ever modified? (e.g. user forgot `final`)
- What if a field already bears `volatile`? (e.g. `AtomicInteger`)
- What if an object is built with builders?

Q: Should we extend the same guarantees to **all** fields and **all** constructors?

Finals: JMM 9

`https://github.com/shipilev/jmm-benchmarks/`

- 2x12x2 Xeon E5-2697, 2.70GHz; OEL 6, JDK 8b121, x86_64
- 1x4x1 Cortex-A9, 1.7 GHz; Linaro 12.11, JDK 8b121, SE Embedded
- We can only measure the performance of some implementation, not the spec itself

Finals: JMM 9

`https://github.com/shipilev/jmm-benchmarks/`

- 2x12x2 Xeon E5-2697, 2.70GHz; OEL 6, JDK 8b121, x86_64
- 1x4x1 Cortex-A9, 1.7 GHz; Linaro 12.11, JDK 8b121, SE Embedded
- We can only measure the performance of some implementation, not the spec itself

Java

Finals: JMM 9: Initialization (chained)

```
@Benchmark
public Object test() {
   return new Test_[N](v);
}

// chained case
class Test_[N] extends Test_[N-1] {
   private [plain|final] int i_[N];
   public <init>(int v) {
      super(v);
      i_[N] = v;
   }
}
```

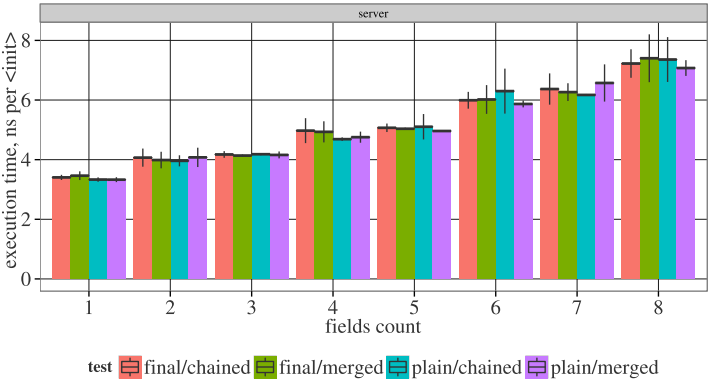

Finals: JMM 9: Initialization (merged)

```
@Benchmark
public Object test() {
   return new Test_[N](v);
}

// merged case
class Test_[N] {
   private [plain|final] int i_1, ..., i_[N];
   public <init>(int v) {
      i_1 = i_2 = ... = i_[N] = v;
   }
}
```


Finals: JMM 9: Results (x86)

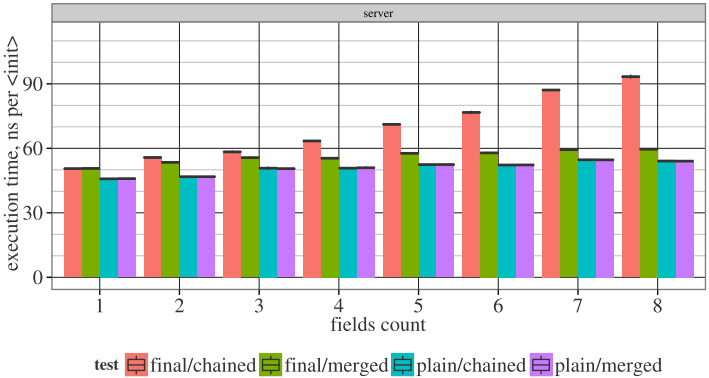
Total Store Order has it for free:[14]



[14] http://shipilev.net/blog/2014/all-fields-are-final/

Finals: JMM 9: Results (ARMv7)

Need barrier coalescing on weakly-ordered architectures: [15]



[15] http://shipilev.net/blog/2014/all-fields-are-final/

Conclusion

Conclusion: Lingua Latina...

«The best way is to build up a small repertoire of constructions that you know the answers for and then never think about the JMM rules again unless you are forced to do so! Literally nobody likes figuring things out from the JMM rules as stated, or can even routinely do so correctly. This is one of the many reasons we need to overhaul JMM someday.»

Conclusion: Lingua Latina...

«The best way is to build up a small repertoire of constructions that you know the answers for and then never think about the JMM rules again unless you are forced to do so! Literally nobody likes figuring things out from the JMM rules as stated, or can even routinely do so correctly. This is one of the many reasons we need to overhaul JMM someday.»

(Doug Lea, private communication, 2013)

Conclusion: Known Problems

- JSR 133 Cookbook misses some machine-specific things, which were discovered after JMM had been sealed
- Some library primitives are not expressible in current model (e.g. `lazySet`, `weakCompareAndSet`)
- JMM is specified for Java, what guarantees other JVM languages have?
- Formal spec has some errors which make automatic checkers cry

Conclusion: JMM Overhaul

«Java Memory Model update»
`http://openjdk.java.net/jeps/188`

- Improved formalization
- JVM languages coverage
- Extended scope for existing unspec-ed primitives
- C11/C++11 compatibility
- Testing support
- Tool support

Conclusion: Reading List

- Goetz et al, «Java Concurrency in Practice»
- Herilhy, Shavit, «The Art of Multiprocessor Programming»
- Adve, «Shared Memory Models: A Tutorial»
- McKenney,«Is Parallel Programming Hard, And, If So, What Can You Do About It?»
- Manson, «Java Memory Model» (Special PoPL issue)
- Huisman, Petri, «JMM: The Formal Explanation»

Java

Q/A

Backup

Backup: Actions

Action: $A =< t, k, v, u >$

- t – the thread performing the action
- k – the kind of action
- v – the variable or monitor involved in the action
- u – an arbitrary unique identifier for the action

Backup: Executions

Execution: $E =< P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} >$

- P – program; A – set of program actions
- $\xrightarrow{po}$ – program order;
- $\xrightarrow{so}$ – synchronization order
- $\xrightarrow{sw}$ – synchronizes-with order
- $\xrightarrow{hb}$ – happens-before order
- $W(r)$ – «write seen function», answers what write the read observes; $V(r)$ – answers what value the read observes

