

Weak memory models

Mai Thuong Tran

PMA Group, University of Oslo, Norway

31 Oct. 2014



1 Introduction

- Hardware architectures
- Compiler optimizations
- Sequential consistency

2 Weak memory models

- TSO memory model (Sparc, x86-TSO)
- The ARM and POWER memory model
- The Java memory model

3 Summary and conclusion

1 Introduction

- Hardware architectures
- Compiler optimizations
- Sequential consistency

2 Weak memory models

- TSO memory model (Sparc, x86-TSO)
- The ARM and POWER memory model
- The Java memory model

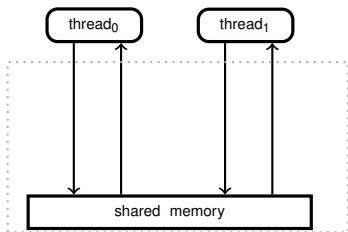
3 Summary and conclusion

Concurrency

“Concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other” (Wikipedia)

- performance increase, better latency
- many forms of concurrency/parallelism: multi-core, multi-threading, multi-processors, distributed systems

Shared memory: a simplistic picture



- one way of “interacting” (i.e., communicating and synchronizing): via **shared memory**
- a number of threads/processors: access common memory/address space
- interacting by sequence of read/write (or load/stores etc)

however: considerably harder to get correct and efficient programs

Dekker's solution to mutex

- As known, shared memory programming requires synchronization: **mutual exclusion**

Dekker

- simple and first known mutex algo
- here slightly simplified

initially: $\text{flag}_0 = \text{flag}_1 = 0$

<pre>flag₀ := 1; if (flag₁ = 0) then CRITICAL</pre>	<pre>flag₁ := 1; if (flag₀ = 0) then CRITICAL</pre>
---	---

Dekker's solution to mutex

- As known, shared memory programming requires synchronization: **mutual exclusion**

Dekker

- simple and first known mutex algo
- here slightly simplified

initially: $\text{flag}_0 = \text{flag}_1 = 0$

<pre>flag₀ := 1; if (flag₁ = 0) then CRITICAL</pre>	<pre>flag₁ := 1; if (flag₀ = 0) then CRITICAL</pre>
---	---

known textbook “fact”:

Dekker is a software-based solution to the mutex problem (or is it?)

Dekker's solution to mutex

- As known, shared memory programming requires synchronization: **mutual exclusion**

Dekker

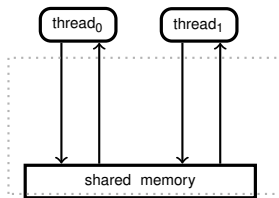
- simple and first known mutex algo
- here slightly simplified

initially: $flag_0 = flag_1 = 0$

<pre>flag₀ := 1; if (flag₁ = 0) then CRITICAL</pre>	<pre>flag₁ := 1; if (flag₀ = 0) then CRITICAL</pre>
---	---

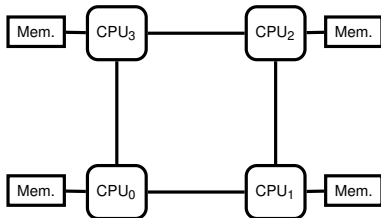
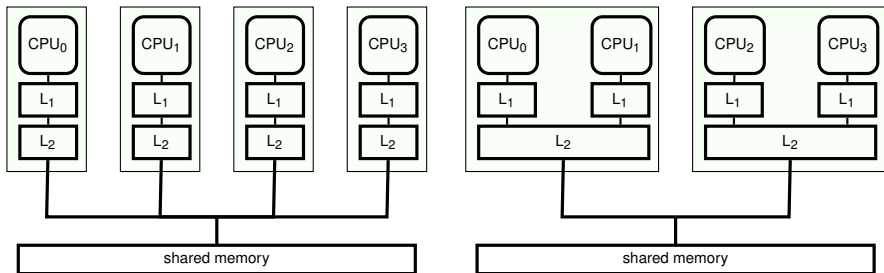
programmers need to know concurrency

Shared memory concurrency in the real world



- the memory architecture does not reflect reality
- out-of-order executions:
 - modern systems: complex memory hierarchies, caches, buffers...
 - compiler optimizations,

SMP, multi-core architecture, and NUMA



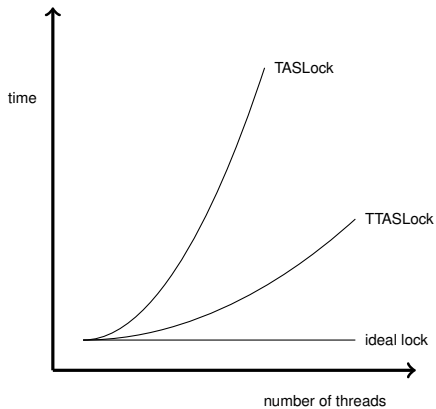
Modern HW architectures and performance

```
public class TASLock implements Lock {
    ...
    public void lock() {
        while (state.getAndSet(true)) { } // spin
    }
    ...
}
```

```
public class TTASLock implements Lock {
    ...
    public void lock() {
        while (true) {
            while (state.get()) {}; //spin
            if (!state.getAndSet(true))
                return;
        }
        ...
    }
}
```

(cf. [Anderson, 1990] [Herlihy and Shavit, 2008, p.470])

Observed behavior



- many optimizations with different forms:
 - **elimination** of reads, writes, sometimes synchronization statements
 - **re-ordering** of independent non-conflicting memory accesses
 - **introductions** of reads
- examples
 - constant propagation
 - common sub-expression elimination
 - dead-code elimination
 - loop-optimizations
 - call-inlining
 - ... and many more

Code reordering

Initially: $x = y = 0$

thread ₀	thread ₁
$x := 1$	$y := 1;$
$r_1 := y$	$r_2 := x;$
print r_1	print r_2

\implies

Initially: $x = y = 0$

thread ₀	thread ₁
$r_1 := y$	$y := 1;$
$x := 1$	$r_2 := x;$
print r_1	print r_2

possible print-outs

$\{(0, 1), (1, 0), (1, 1)\}$

possible print-outs

$\{(0, 0), (0, 1), (1, 0), (1, 1)\}$

Golden rule of compiler optimization

Change the code (for instance re-order statements, re-group parts of the code, etc) in a way that leads to

- better performance, but is otherwise
- **unobservable** to the programmer (i.e., does not introduce new observable result(s))

In the presence of concurrency

- more forms of “interaction”
- ⇒ more effects become observable
- standard optimizations become observable (i.e., “break” the code, assuming a naive, standard shared memory model)

Compiler optimizations

Golden rule of compiler optimization

Change the code (for instance re-order statements, re-group parts of the code, etc) in a way that leads to

- better performance, but is otherwise
- **unobservable** to the programmer (i.e., does not introduce new observable result(s))
when executed **single-threadedly**, i.e. **without** concurrency!

In the presence of concurrency

- more forms of “interaction”
- ⇒ more effects become **observable**
- standard optimizations become **observable** (i.e., “break” the code, assuming a naive, standard shared memory model)

Compilers vs. programmers

Programmer

- want's to understand the code
- ⇒ profits from strong memory models



Compiler/HW

- want to **optimize** code/execution (re-ordering memory accesses)
- ⇒ take advantage of weak memory models



- What are valid (semantics-preserving) compiler-optimations?
- What is a good memory model as compromise between programmer's needs and chances for optimization

Sad facts and consequences

- **incorrect** concurrent code, “unexpected” behavior
 - Dekker (and other well-know mutex algo's) is incorrect on modern architectures¹
- unclear/obstruse/informal hardware specifications, compiler optimizations may not be transparent
- understanding of the memory architecture also crucial for **performance**

Need for unambiguous description of the behavior of a chosen platform/language under shared memory concurrency \implies memory models

¹Actually already since at least IBM 370.

What's a memory model?

“A formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system.” [Adve and Gharachorloo, 1995]

MM specifies:

- How threads interact through memory.
- What value a read can return.
- When does a value update become visible to other threads.
- What assumptions are allowed to make about memory when writing a program or applying some program optimization.

Sequential consistency

- in the previous examples: unspoken assumptions

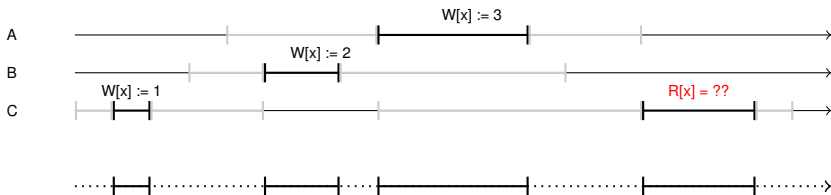
- 1 **Program** order: statements executed in the order written/issued (Dekker).
- 2 **atomicity**: memory update is visible to everyone at the same time

Lamport [Lamport, 1979]: Sequential consistency

“...the results 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**.”

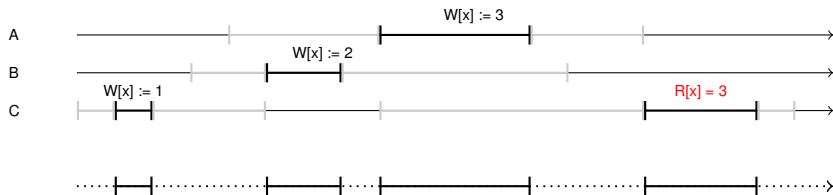
- “classical” model, (one of the) oldest correctness conditions
- simple/simplistic \Rightarrow (comparatively) easy to understand
- straightforward generalization: single \Rightarrow multi-processor
- **weak** means basically “more relaxed than SC”

Atomicity: no overlap



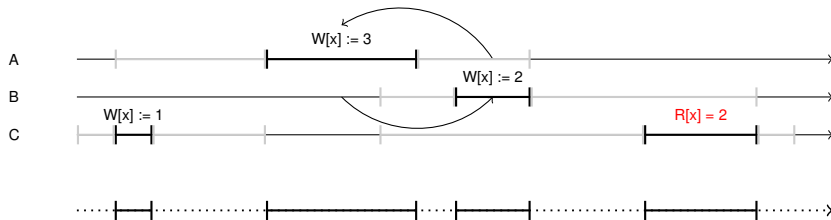
Which values for x consistent with SC?

Atomicity: no overlap



Which values for x consistent with SC?

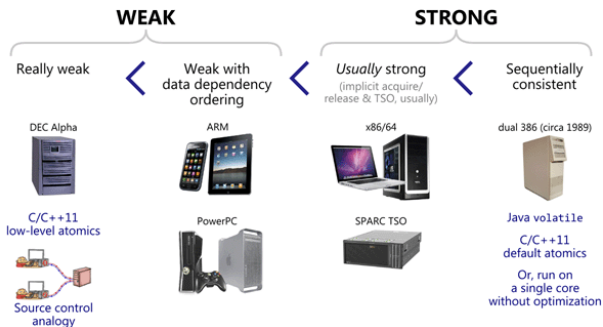
Some order consistent with the observation



- read of 2: observable under sequential consistency (as is 1, and 3)
- read of 0: contradicts **program order** for thread C.

- 1 Introduction
 - Hardware architectures
 - Compiler optimizations
 - Sequential consistency
- 2 Weak memory models
 - TSO memory model (Sparc, x86-TSO)
 - The ARM and POWER memory model
 - The Java memory model
- 3 Summary and conclusion

Spectrum of available architectures



(from <http://preshing.com/20120930/weak-vs-strong-memory-models>)

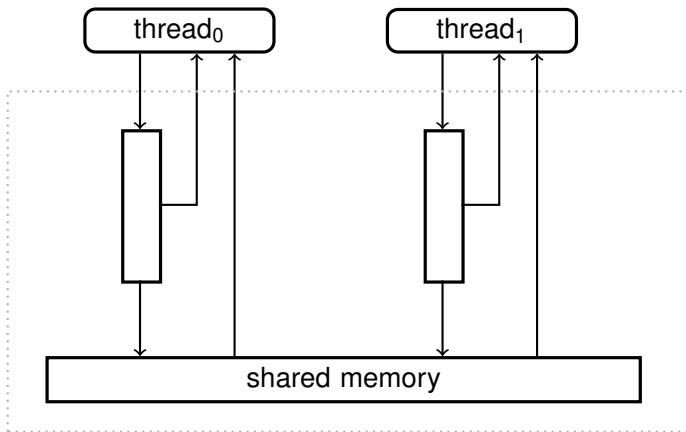
Trivial example

thread ₀	thread ₁
x := 1	y := 1
print y	print x

Result?

Is the printout 0, 0 observable?

Hardware optimization: Write buffers

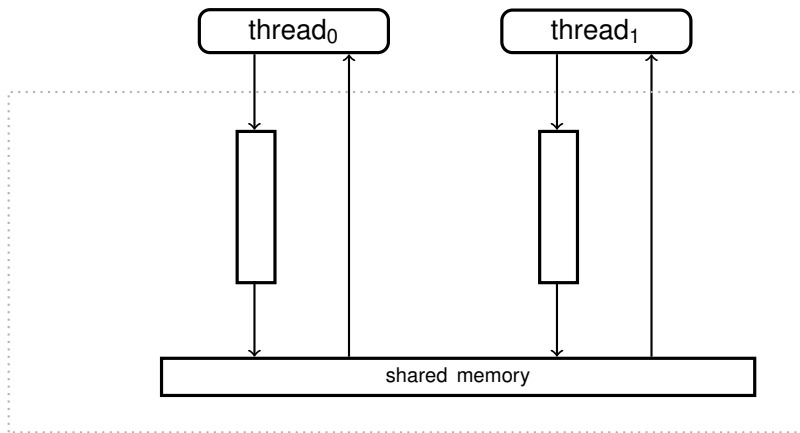


- TSO: SPARC, pretty old already
- x86-TSO
- see [Owell et al., 2009] [Sewell et al., 2010]

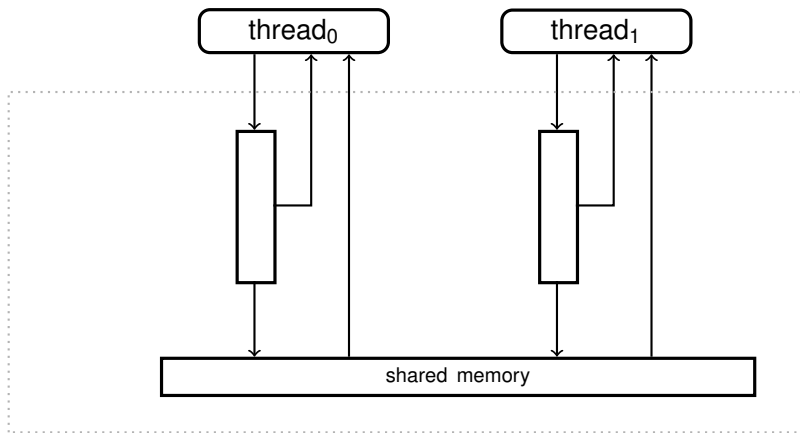
Relaxation

- 1 architectural: adding **store buffers** (aka write buffers)
- 2 axiomatic: **relaxing** program order \Rightarrow **W-R** order dropped

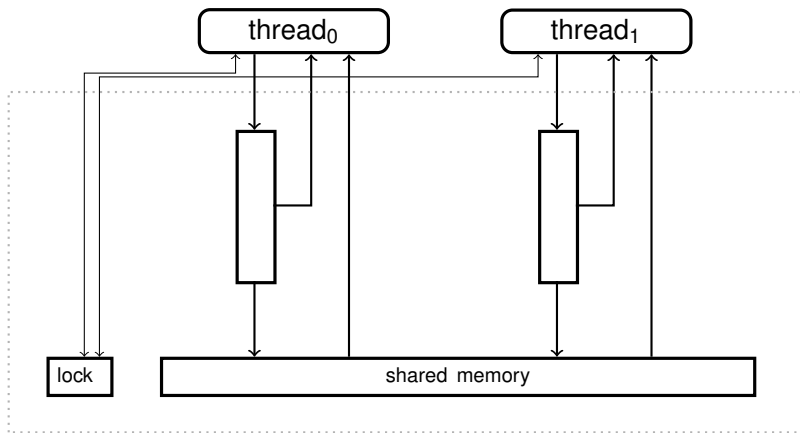
Architectural model: Write-buffers (IBM 370)



Architectural model: TSO (SPARC)



Architectural model: x86-TSO



Directly from Intel's spec

Intel 64/IA-32 architecture software developer's manual [int, 2013]
(over 3000 pages long!)

- single-processor systems:
 - Reads are not **reordered** with other reads.
 - Writes are not **reordered** with older reads.
 - Reads may be **reordered** with older writes to different locations but **not** with older writes to the same location.
 - ...
- for multiple-processor system
 - Individual processors use the same ordering principles as in a single-processor system.
 - Writes by a single processor are observed in the same order by all processors.
 - Writes from an individual processor are NOT ordered with respect to the writes from other processors ...
 - **Locked instructions have a total order**

- FIFO store buffer
- read = read the most recent buffered write, if it exists (else from main memory)
- buffered write: can propagate to shared memory at any time (except when lock is held by other threads).

behavior of LOCK'ed instructions

- obtain global lock
- flush store buffer at the end
- release the lock
- note: no reading allowed by other threads if lock is held

SPARC V8 Total Store Ordering (TSO):

a read can complete before an earlier write to a different address, but a read cannot return the value of a write by another processor unless all processors have seen the write (it returns the value of own write before others see it)

Consequences: In a thread: for a write followed by a read (to different addresses) the order can be swapped

Justification: Swapping of $W - R$ is not observable by the programmer, it does not lead to new, unexpected behavior!

Example

thread	thread'
flag := 1	flag' := 1
A := 1	A := 2
reg ₁ := A	reg' ₁ := A
reg ₂ := flag'	reg' ₂ := flag

Result?

In TSO^a

- $(\text{reg}_1, \text{reg}'_1) = (1, 2)$ observable (as in SC)
- $(\text{reg}_2, \text{reg}'_2) = (0, 0)$ observable

^aDifferent from IBM 370, which also has write buffers, but not the possibility for a thread to read from it's own write buffer

Axiomatic description

- consider “temporal” ordering of memory commands (read/write, load/store etc)
- program order $<_p$:
 - order in which memory commands are issued by the processor
 - = order in which they appear in the program code
- memory order $<_m$: order in which the commands become effective/visible in main memory

Order (and value) conditions

$$\text{RR: } l_1 <_p l_2 \implies l_1 <_m l_2$$

$$\text{WW: } s_1 <_p s_2 \implies s_1 <_m s_2$$

$$\text{RW: } l_1 <_p s_2 \implies l_1 <_m s_2$$

$$\text{Latest write wins: } \text{val}(l_1) = \text{val}(\max_{<_m} \{s_1 <_m l_1 \quad \vee \quad s_1 <_p l_1\})$$

- ARM and POWER: similar to each other
- ARM: widely used inside smartphones and tablets (battery-friendly)
- POWER architecture = **P**erformance **O**ptimization **W**ith **E**nanced **R**ISC., main driver: IBM

Memory model

much *weaker* than x86-TSO

- exposes *multiple-copy* semantics to the programmer

“Message passing” example in POWER/ARM

thread₀ wants to **pass a message** over “channel” x to thread₁,
shared var y used as flag.

Initially: $x = y = 0$

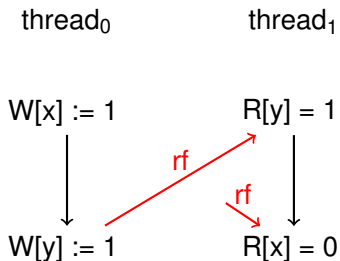
thread ₀	thread ₁
$x := 1$	<code>while (y=0) { };</code>
$y := 1$	$r := x$

Result?

Is the result $r = 0$ observable?

- impossible in (x86-)TSO
- it would violate W-W order

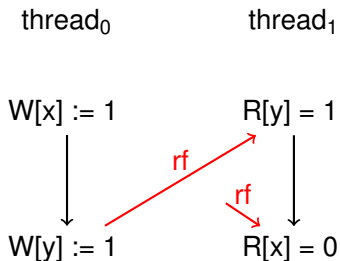
Analysis of the example



How could that happen?

- 1 thread does **stores** out of order
- 2 thread does **loads** out of order
- 3 store **propagates** between threads out of order.

Analysis of the example

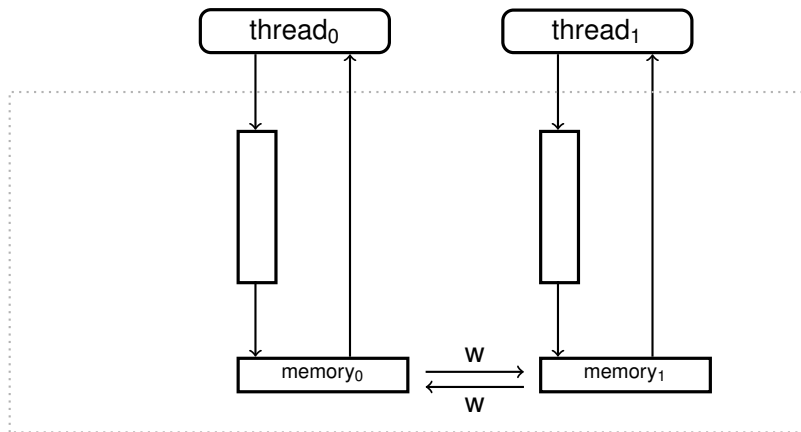


How could that happen?

- 1 thread does **stores** out of order
- 2 thread does **loads** out of order
- 3 store **propagates** between threads out of order.

Power/ARM do **all three!**

Conceptual memory architecture



Power and ARM order constraints

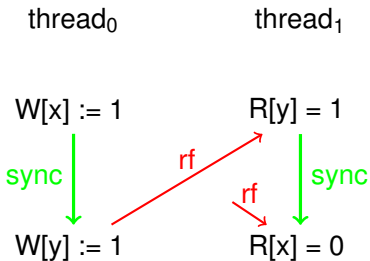
basically, program order **is not preserved!** unless

- writes to the same location
- **address dependency** between two loads
- dependency between a load and a store,
 - 1 address dependency
 - 2 data dependency
 - 3 control dependency
- use of **synchronization instructions**

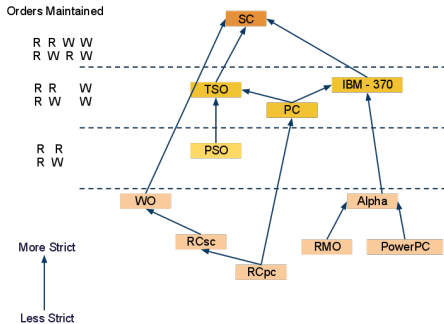
Repair of the MP example

To avoid reorder: Barriers

- heavy-weight: `sync` instruction (POWER)
- light-weight: `lwsync`



Relationship between different models



(from

http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2013/10c_ks)

Java memory model

- known example for a memory model for a programming language.
- specifies how Java threads interact through memory
- **weak** memory model
- under **long** development and debate
- original model (from 1995):
 - widely criticized as flawed
 - disallowing many runtime optimizations
 - no good guarantees for code safety
- more recent proposal: **Java Specification Request 133** (JSR-133), part of Java 5
- see [Manson et al., 2005]

Correctly synchronized programs and others

- 1 **Correctly** synchronized programs: correctly synchronized, i.e., data-race free, programs are **sequentially consistent** (“**Data-race free**” model [Adve and Hill, 1990])
- 2 **Incorrectly** synchronized programs: A clear and definite semantics for **incorrectly** synchronized programs, without breaking Java’s security/safety guarantees.

tricky balance for programs with data races:

disallowing programs violating Java’s security and safety guarantees vs. flexibility still for standard compiler optimizations.

Data race free model

data race free programs/executions are sequentially consistent

Data race

- A data race is the “simultaneous” access by two threads to the same shared memory location, with at least one access a write.
- a program is race free if no execution reaches a race.
- note: the definition is ambiguous!

Data race free model

data race free programs/executions are **sequentially consistent**

Data race with a twist

- A data race is the “simultaneous” access by two threads to the same shared memory location, with at least one access a write.
- a program is race free if no **sequentially consistent** execution reaches a race.

synchronizing actions: locking, unlocking, access to **volatile** variables

Definition

- 1 **synchronization order** $<_{sync}$: total order on all synchronizing actions (in an execution)
- 2 **synchronizes-with order**: $<_{sw}$
 - an unlock action synchronizes-with all $<_{sync}$ -subsequent lock actions by any thread
 - similarly for volatile variable accesses
- 3 **happens-before** ($<_{hb}$): transitive closure of **program** order and **synchronizes-with** order

Happens-before memory model

- simpler than/approximation of Java's memory model
- distinguishing volatile from non-volatile reads
- happens-before

Happens before consistency

In a given execution:

- if $R[x] <_{hb} W[X]$, then the read **cannot** observe the write
- if $W[X] <_{hb} R[X]$ and the read observes the write, then there does not exist a $W'[X]$ s.t. $W[X] <_{hb} W'[X] <_{hb} R[X]$

Synchronization order consistency (for volatile-s)

- $<_{sync}$ consistent with $<_p$.
- If $W[X] <_{hb} W'[X] <_{hb} R[X]$ then the read sees the write $W'[X]$

Incorrectly synchronized code

Initially: $x = y = 0$

thread ₀	thread ₁
$r_1 := x$	$r_2 := y$
$y := r_1$	$x := r_2$

- obviously: a race
- however:

out of thin air

observation $r_1 = r_2 = 42$ not wished, but consistent with the happens-before model!

- cf. also the “message passing” example

ready volatile

Initially: $x = 0$, $ready = false$

thread ₀	thread ₁
$x := 1$	if (ready)
ready := true	$r_1 := x$

- ready volatile $\Rightarrow r_1 = 1$ guaranteed

Problem with the happens-before model

Initially: $x = 0$, $y = 0$

thread ₀	thread ₁
$r_1 := x$	$r_2 := y$
if ($r_1 \neq 0$)	if ($r_2 \neq 0$)
$y := 42$	$x := 42$

- the program is **correctly synchronized!**
- ⇒ observation $y = x = 42$ disallowed
- However: in the happens-before model, **this is allowed!**

violates the “data-race-free” model

- ⇒ add **causality**

JMM

Java memory model = happens before + causality

- circular causality is unwanted
- causality eliminates:
 - data dependence
 - control dependence

Causality and control dependency

Initially: a = 0; b = 1			Initially: a = 0; b = 1	
thread ₀	thread ₁		thread ₀	thread ₁
r ₁ := a	r ₃ := b	⇒	b := 2	r ₃ := b;
r ₂ := a	a := r ₃ ;		r ₁ := a	a := r ₃ ;
if (r ₁ = r ₂)			r ₂ := r ₁	
b := 2;			if (true) ;	

is $r_1 = r_2 = r_3 = 2$
possible?

$r_1 = r_2 = r_3 = 2$ is
sequentially consistent

Optimization breaks control dependency

Causality and data dependency

Initially: $x = y = 0$			Initially: $x = y = 0$		
thread ₀		thread ₁	thread ₀		thread ₁
$r_1 := x;$		$r_3 := y;$	$r_2 := 1$		$r_3 := y;$
$r_2 := r_1 \vee 1;$		$x := r_3;$	$y := 1$		$x := r_3;$
$y := r_2;$			$r_1 := x$		

Is $r_1 = r_2 = r_3 = 1$
possible?

using global analysis

\vee = bit-wise or on integers

Optimization breaks data dependence

Summary: Un-/Desired outcomes for causality

Disallowed behavior

Initially: $x = y = 0$

thread ₀	thread ₁
$r_1 := x$	$r_2 := y$
$y := r_1$	$x := r_2$

$r_1 = r_2 = 42$

Initially: $x = 0, y = 0$

thread ₀	thread ₁
$r_1 := x$	$r_2 := y$
if ($r_1 \neq 0$)	if ($r_2 \neq 0$)
$y := 42$	$x := 42$

$r_1 = r_2 = 42$

Allowed behavior

Initially: $a = 0; b = 1$

thread ₀	thread ₁
$r_1 := a$	$r_3 := b$
$r_2 := a$	$a := r_3;$
if ($r_1 = r_2$)	
$b := 2;$	

is $r_1 = r_2 = r_3 = 2$ possible?

Initially: $x = y = 0$

thread ₀	thread ₁
$r_1 := x;$	$r_3 := y;$
$r_2 := r_1 \vee 1;$	$x := r_3;$
$y := r_2;$	

is $r_1 = r_2 = r_3 = 1$ possible?

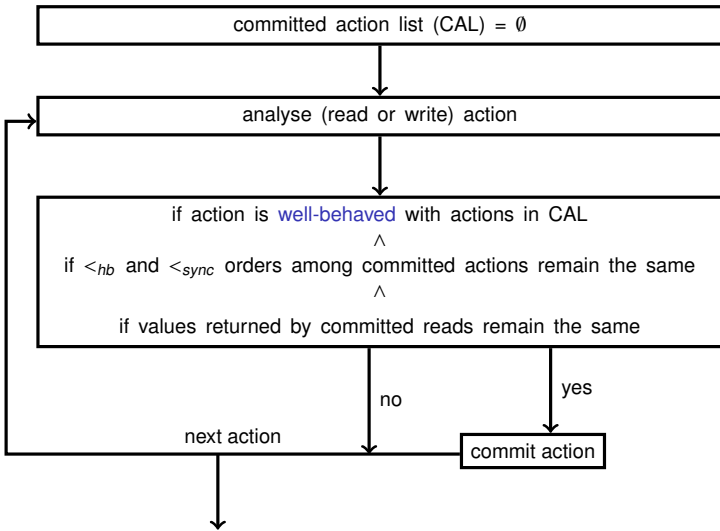
Causality and the JMM

- key of causality: **well-behaved** executions (i.e. consistent with SC execution)
- non-trivial, **subtle** definition
- writes can be done **early** for **well-behaved** executions

Well-behaved

a not yet committed read must return the value of a write which is $<_{hb}$.

Iterative algorithm for well-behaved executions



- considerations for **implementors**
 - control dependence: should not reorder a write above a non-terminating loop
 - weak memory model: semantics allow **re-ordering**,
 - other code transformations
 - synchronization on thread-local objects can be ignored
 - volatile fields of thread local objects: can be treated as normal fields
 - redundant synchronization can be ignored.
- Consideration for **programmers**
 - DRF-model: make sure that the program is correctly synchronized \Rightarrow don't worry about re-orderings
 - Java-spec: no guarantees whatsoever concerning pre-emptive scheduling or fairness

- 1 Introduction
 - Hardware architectures
 - Compiler optimizations
 - Sequential consistency
- 2 Weak memory models
 - TSO memory model (Sparc, x86-TSO)
 - The ARM and POWER memory model
 - The Java memory model
- 3 Summary and conclusion

Conclusion

Take-home lesson

it's *impossible*(!!) to produce

- correct and
- high-performance

concurrent code without clear knowledge of the chosen platform's/language's MM

- that holds: not only for system programmers, OS-developers, compiler builders . . . but also for “garden-variety” SW developers
- reality (since long) much more complex than “naive” SC model

Take home lesson for the impatient

Avoid data races at (almost) all costs (by using synchronization)!

References I

[int, 2013] (2013).

Intel 64 and IA-32 Architectures Software Developers Manual. Combined Volumes:1, 2A, 2B, 2C, 3A, 3B and 3C.
Intel.

[Adve and Boehm, 2010] Adve, S. V. and Boehm, H.-J. (2010).

Memory models: a case for rethinking parallel languages and hardware.
Communications of the ACM, 53(8):90–101.

[Adve and Gharachorloo, 1995] Adve, S. V. and Gharachorloo, K. (1995).

Shared memory consistency models: A tutorial.
Research Report 95/7, Digital WRL.

[Adve and Hill, 1990] Adve, S. V. and Hill, M. D. (1990).

Weak ordering — a new definition.
SIGARCH Computer Architecture News, 18(3a).

[Anderson, 1990] Anderson, T. E. (1990).

The performance of spin lock alternatives for shared-memory multiprocessors.
IEEE Transactions on Parallel and Distributed System, 1(1):6–16.

[Boehm and Adve, 2012] Boehm, H.-J. and Adve, S. V. (2012).

You don't know jack about shared variables or memory models.
Communications of the ACM, 55(2):48–54.

[Herlihy and Shavit, 2008] Herlihy, M. and Shavit, N. (2008).

The Art of Multiprocessor Programming.
Morgan Kaufmann.

[Lampert, 1979] Lampert, L. (1979).

How to make a multiprocessor computer that correctly executes multiprocess programs.
IEEE Transactions on Computers, C-28(9):690–691.

[Manson et al., 2005] Manson, J., Pugh, W., and Adve, S. V. (2005).

The Java memory memory.
In Proceedings of POPL '05. ACM.

- [Maranget et al., 2013] Maranget, L., Sarkar, S., and Sewell, P. (2013).
A tutorial introduction to the ARM and POWER relaxed memory models (draft).
- [Nardelli, 2012] Nardelli, F. Z. (2012).
Shared memory: An elusive abstraction.
Tutorial slides.
- [Owell et al., 2009] Owell, S., Sarkar, S., and Sewell, P. (2009).
A better x86 memory model: x86-TSO.
In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, Theorem Proving in Higher-Order Logic: 10th International Conference, TPHOLs'09, volume 5674 of Lecture Notes in Computer Science.
- [Phreshing, 2013] Phreshing, J. (2013).
Phreshing on programming.
Blog at <http://phreshing.com>.
- [Sarkar, 2013] Sarkar, S. (2013).
Shared memory concurrency in the real world. working with relaxed memory consistency.
Presentation.
- [Sewell et al., 2010] Sewell, P., Sarkar, S., Nardelli, F., and O.Myreen, M. (2010).
x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors.
Communications of the ACM, 53(7).