Programming Language Memory Models: What do Shared Variables Mean?

Hans-J. Boehm

Disclaimers:

- This is an overview talk.
- Much of this work was done by others or jointly. I'm relying particularly on:
  - Basic approach: Sarita Adve, Mark Hill, Ada 83 …
  - JSR 133: Also Jeremy Manson, Bill Pugh, Doug Lea
  - C++0x: Lawrence Crowl, Clark Nelson, Paul McKenney, Herb Sutter, …
  - Improved hardware models: Peter Sewell's group, many Intel, AMD, ARM, IBM participants …
  - Conflict exception work: Ceze, Lucia, Qadeer, Strauss
  - Recent Java Memory Model work: Sevcik, Aspinall, Cenciarelli
- But some of it is still controversial.
  - This reflects my personal views.

The problem

- Shared memory parallel programs are built on shared variables visible to multiple threads of control.
- But there is a lot of confusion about what those variables mean:
  - Are concurrent accesses allowed?
  - What is a concurrent access?
  - When do updates become visible to other threads?
  - Can an update be partially visible?
- Many recent efforts with serious technical issues:
  - Java, OpenMP 3.0, UPC(?), Go happens-before consistency, …

Outline

- Emerging consensus:
  - Interleaving semantics (Sequential Consistency)
  - But only for data-race-free programs
- Brief discussion of consequences
  - Software requirements
  - Hardware requirements
- Major remaining problem:
  - Java can’t outlaw races.
  - We don’t know how to give meaning to data races.
  - Some speculative solutions.

Naive threads programming model (Sequential Consistency)

- Threads behave as though their memory accesses were simply interleaved. (Sequential consistency)

Thread 1

```
x = 1;
z = 3;
```

Thread 2

```
y = 2;
```

  - might be executed as

```
x = 1; y = 2; z = 3;
```

Locks restrict interleavings

Thread 1

```
lock(l);
r1 = x;
x = r1+1;
unlock(l);
```

Thread 2

```
lock(l);
r2 = x;
x = r2+1;
unlock(l);
```

– can only be executed as

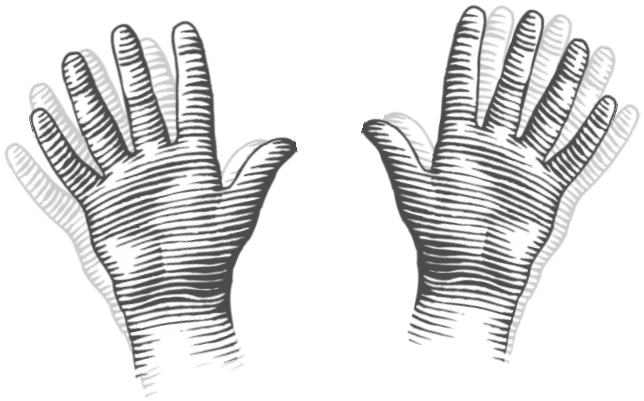
```
lock(l); r1 = x; x = r1+1; unlock(l); lock(l);
r2 = x; x = r2+1; unlock(l);
```

or

```
lock(l); r2 = x; x = r2+1; unlock(l); lock(l);
r1 = x; x = r1+1; unlock(l);
```

since second lock(l) must follow first unlock(l)

Atomic sections / transactional memory are just like a single global lock.



But this doesn't quite work …

- Limits reordering and other hardware/compiler transformations
  - "Dekker's" example (everything initially zero) should allow `r1` = `r2` = 0:

Thread 1	*Thread 2*
`x = 1;`	`y = 1;`
`r1 = y;`	`r2 = x;`

- Sensitive to memory access granularity:

Thread 1	*Thread 2*
`x = 300;`	`x = 100;`

  - may result in x = 356 with sequentially consistent byte accesses.

And we didn't quite want that anyway …

- Sensitive to memory access granularity:

Thread 1	*Thread 2*
`x = 300;`	`x = 100;`

  - may result in x = 356 with sequentially consistent byte accesses.
- And taking advantage of sequential consistency involves reasoning about memory access interleaving:
  - Much too hard.
  - Want to reason about larger "atomic" code regions
    - which can't be visibly interleaved.

Real threads programming model (1)

- Two memory accesses conflict if they
  - access the same scalar object*, e.g. variable.
  - at least one access is a store.
  - E.g. `x = 1;` and `r2 = x;` conflict
- Two ordinary memory accesses participate in a data race if they
  - conflict, and
  - can occur simultaneously
    - i.e. appear as adjacent operations in interleaving.
- A program is data-race-free (on a particular input) if no sequentially consistent execution results in a data race.

* or contiguous sequence of bit-fields

Real threads programming model (2)

- Sequential consistency only for data-race-free programs!
  - Avoid anything else.
- Data races are prevented by
  - locks (or atomic sections) to restrict interleaving
  - declaring synchronization variables
    - (next slide …)

Synchronization variables

- Java: `volatile`, `java.util.concurrent.atomic`.
- C++0x: `atomic<int>`
- C++0x, C1x: `_Atomic(int)`, `_Atomic int`? `atomic_int`?
- Guarantee indivisibility of operations.
- "Don't count" in determining whether there is a data race:
  - Programs with "races" on synchronization variables are still sequentially consistent.
  - Though there may be "escapes" (Java, C++0x, not discussed here).
- Dekker's algorithm "just works" with synchronization variables.

Data Races Revisited

- Are defined in terms of sequentially consistent executions.
- If x and y are initially zero, this does *not* have a data race:

Thread 1

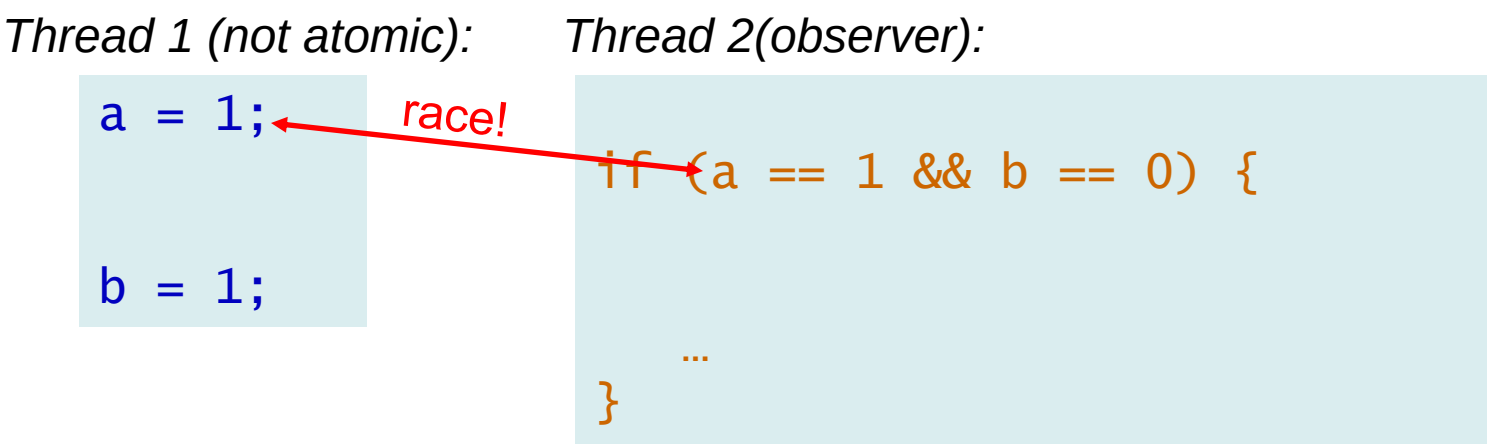
```
if (x)
  y = 1;
```

Thread 2

```
if (y)
  x = 1;
```

SC for DRF programming model advantages over SC

- Supports important hardware & compiler optimizations.
- DRF restriction ➔ Synchronization-free code sections appear to execute atomically, i.e. without visible interleaving.
  - If one didn't:



Basic Implementation model

- Very restricted reordering of memory operations around synchronization operations:
  - Compiler either understands these, or treats them as opaque, potentially updating any location.
  - Synchronization operations include instructions to limit or prevent hardware reordering (“memory fences”).
- Other reordering is invisible:
  - Only racy programs can tell.

Some variants

C++ draft (C++0x) C draft (C1x)	SC for DRF*, Data races are errors
Java	SC for DRF**, More details later.
Ada83+, Posix threads	SC for drf (sort of)
OpenMP, Fortran 2008	SC for drf (except atomics, sort of)
.Net	Getting there, we hope ☺

* Except explicitly specified memory ordering. ** Except some j.u.c.atomic.

Outline

- Emerging consensus:
  - Interleaving semantics (Sequential Consistency)
  - But only for data-race-free programs
- Brief discussion of consequences
  - Software requirements
  - Hardware requirements
- Major remaining problem:
  - Java can’t outlaw races.
  - We don’t know how to give meaning to data races.
  - Some speculative solutions.

Compilers must not introduce data races

- Single thread compilers currently may add data races: (PLDI 05)

```
struct {char a; char b} x;
```

```
x.a = 'z';
```

```
tmp = x;
tmp.a = 'z';
x = tmp;
```

  - `x.a = 1` in parallel with `x.b = 1` may fail to update x.b.
- … and much more interesting examples.
- Still broken in gcc in subtle cases.

Language spec challenge:

- *Some really awful code:*

Don't try this at home!!

Thread 1:

```
x = 42;
m.lock();
```

?

Thread 2:

```
while (m.trylock()==SUCCESS)
  m.unlock();
assert (x == 42);
```

- Disclaimer: Example requires tweaking to be pthreads-compliant.

- Can the assertion fail?
- Many implementations: Yes
- Traditional specs: No. C++0x: Yes
- `Trylock()` can effectively fail spuriously!

Outline

- Emerging consensus:
  - Interleaving semantics (Sequential Consistency)
  - But only for data-race-free programs
- Brief discussion of consequences
  - Software requirements
  - Hardware requirements
- Major remaining problem:
  - Java can't outlaw races.
  - We don't know how to give meaning to data races.
  - Some speculative solutions.

Byte store instructions

- `x.c = 'a';` may not visibly read and rewrite adjacent fields.
- Byte stores must be implemented with
  - Byte store instruction, or
  - Atomic read-modify-write.
    - Typically expensive on multiprocessors.
    - Often cheaply implementable on uniprocessors.

Sequential consistency must be enforceable

- Programs using only synchronization variables must be sequentially consistent.
- Compiler literature contains many papers on enforcing sequential consistency by adding fences. But:
  - Not really possible on Itanium.
  - Wasn't possible on X86 until the re-revision of the spec last year.
  - Took months of discussions with PowerPC architects to conclude it's (barely, sort of) possible there.
- The core issue is "write atomicity":

Can fences enforce SC?

Unclear that hardware fences can ensure sequential consistency. "IRIW" example:

x, y initially zero. Fences between every instruction pair!

Thread 1:	*Thread 2:*	*Thread 3:*	*Thread 4:*

`x = 1;`	`r1 = x; (1)`
	`fence;`
	`r2 = y; (0)`
		`y = 1;`	`r3 = y; (1)`
			`fence;`
			`r4 = x; (0)`

x set first!

y set first!

Fully fenced, not sequentially consistent. Does hardware allow it?

Why does it matter?

- Nobody cares about IRIW!?
- It's a pain to enforce on at least PowerPC.
- Many people (Sarita Adve, Doug Lea, Vijay Saraswat) spent about a year trying to relax SC requirement here.
- (Personal opinion) The results were incomprehensible, and broke more important code.
- No viable alternatives!

Acceptable hardware memory models

- More challenging requirements:
  1. Precise memory model specification
  2. Byte stores
  3. Cheap mechanism to enforce write atomicity
  4. Dirt cheap mechanism to enforce data dependency ordering(?) (Java final fields)
- Other than that, all standard approaches appear workable, but …

Replace fences completely? Synchronization variables on X86

- atomic store: ~1 cycle dozens of cycles
  - store (`mov`); `mfence`;
- atomic load: ~1 cycle
  - load (`mov`)
- Store implicitly ensures that prior memory operations become visible before store.
- Load implicitly ensures that subsequent memory operations become visible later.
- Sole reason for `mfence`: Order atomic store followed by *atomic* load.

Fence enforces all kinds of additional, unobservable orderings

- `s` is a synchronization variable:

```
x = 1;
s = 2;   // includes fence
r1 = y;
```

- Prevents reordering of `x = 1` and `r1 = y`;
  - final load delayed until assignment to `a` visible.
- But this ordering is invisible to non-racing threads
  - …and expensive to enforce?
- *We need a tiny fraction of* `mfence` *functionality.*

Outline

- Emerging consensus:
  - Interleaving semantics (Sequential Consistency)
  - But only for data-race-free programs
- Brief discussion of consequences
  - Software requirements
  - Hardware requirements
- Major remaining problem:
  - Java can't outlaw races.
  - We don't know how to give meaning to data races.
  - Some speculative solutions.

Data Races in Java

- C++0x leaves data race semantics undefined.
  - "catch fire" semantics
- Java supports sand-boxed code.
- Don't know how to prevent data-races in sand-boxed, malicious code.
- Java must provide some guarantees in the presence of data races.

Interesting data race outcome?

`x, y` initially null,

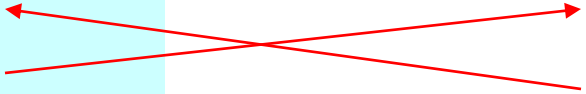
Loads may or may not see racing stores?

Thread 1:

```
r1 = x;
y = r1;
```

Thread 2:

```
r2 = y;
x = r2;
```



Outcome: `x = y = r1 = r2 =`
“<your bank password here>”

The Java Solution

Quotation from 17.4.8, Java Language Specification, 3rd edition, omitted, to avoid possible copyright questions. The important point is that this is a rather complex mathematical specification.

...

Complicated, but nice properties?

- Manson, Pugh, Adve: The Java Memory Model, POPL 05

Quotation from section 9.1.2 of above paper omitted, to avoid possible copyright questions. This asserts (Theorem 1) that non-conflicting operations may be reordered by a compiler.

Much nicer than prior attempts, but:

- Aspinall, Sevcik, “Java Memory Model Examples: Good, Bad, and Ugly”, VAMP 2007 (also ECOOP 2008 paper)

Quotation from above paper omitted, to avoid possible copyright questions. This ends in the statement:

“This falsifies Theorem 1 of [paper from previous slide].”

Note 1: This does not necessarily mean implementations are broken, or that we know how to do better. It does suggest this is too complicated.

Note 2: The underlying observation is due to Pietro Cenciarelli.

Why is this hard?

- Want
  - Constrained race semantics for essential security properties.
  - Unconstrained race semantics to support compiler and hardware optimizations.
  - Simplicity.
- No known good resolution.

Outline

- Emerging consensus:
  - Interleaving semantics (Sequential Consistency)
  - But only for data-race-free programs
- Brief discussion of consequences
  - Software requirements
  - Hardware requirements
- Major remaining problem:
  - Java can't outlaw races.
  - We don't know how to give meaning to data races.
  - Some speculative solutions.

A Different Approach

- Outlaw data races.
- Require violations to be detectable!
  - Even in malicious sand-boxed code.

- Possible approaches:
  - Statically prevent data races.
    - Tried repeatedly, ongoing work …
  - Dynamically detect *the relevant* data races.

Dynamic Race Detection

- Need to guarantee one of:
  - Program is data-race free and provides SC execution (done),
  - Program contains a data race and raises an exception, or
  - Program exhibits simple semantics anyway, e.g.
    - Sequentially consistent
    - Synchronization-free regions are atomic
- This is significantly cheaper than fully accurate data-race detection.
  - Track byte-level R/W information
  - Mostly in cache
  - As opposed to epoch number + thread id per byte

For more information:

- Boehm, “Threads Basics”, HPL TR 2009-259.
- Boehm, Adve, “Foundations of the C++ Concurrency Memory Model”, PLDI 08.
- Sevcık and Aspinall, “On Validity of Program Transformations in the Java Memory Model”, ECOOP 08.
- Sewell et al, “x86-TSO: A Rigorous and Usable Programmer’s Model for x86”, CACM, July 2010.
- S. V. Adve, Boehm, “Memory Models: A Case for Rethinking Parallel Languages and Hardware”, CACM, August 2010.
- Lucia, Strauss, Ceze, Qadeer, Boehm, "Conflict Exceptions: Providing Simple Parallel Language Semantics with Precise Hardware Exceptions, ISCA 2010.

Questions?

Backup slides

Introducing Races (2)

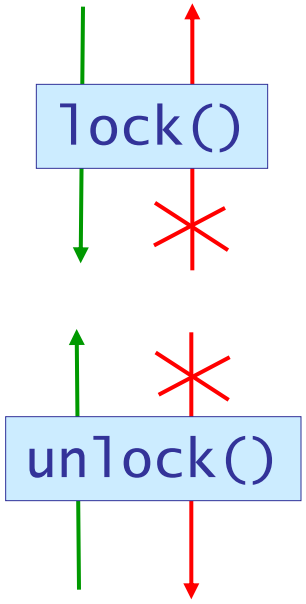
```
int count;     // global, possibly shared
…
for (p = q; p != 0; p = p -> next)
   if (p -> data > 0) ++count;
```

```
int count;     // global, possibly shared
…
reg = count;
for (p = q; p != 0; p = p -> next)
   if (p -> data > 0) ++reg;
count = reg;   // may spuriously assign to count
```

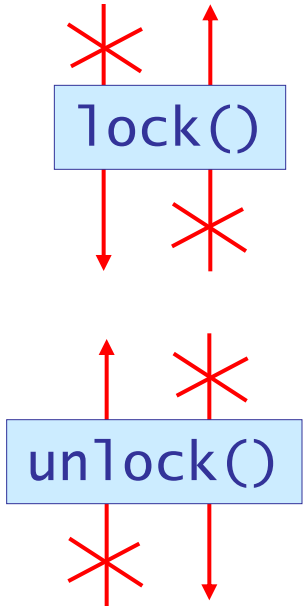
Trylock: Critical section reordering?

- Reordering of memory operations with respect to critical sections:

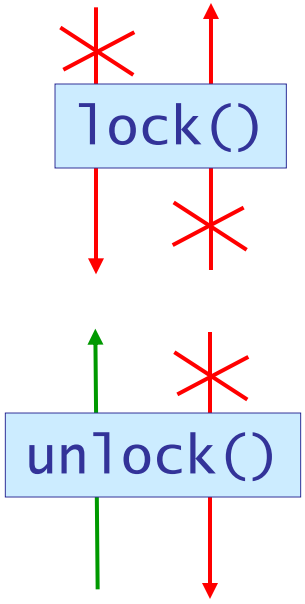
Expected (& Java):



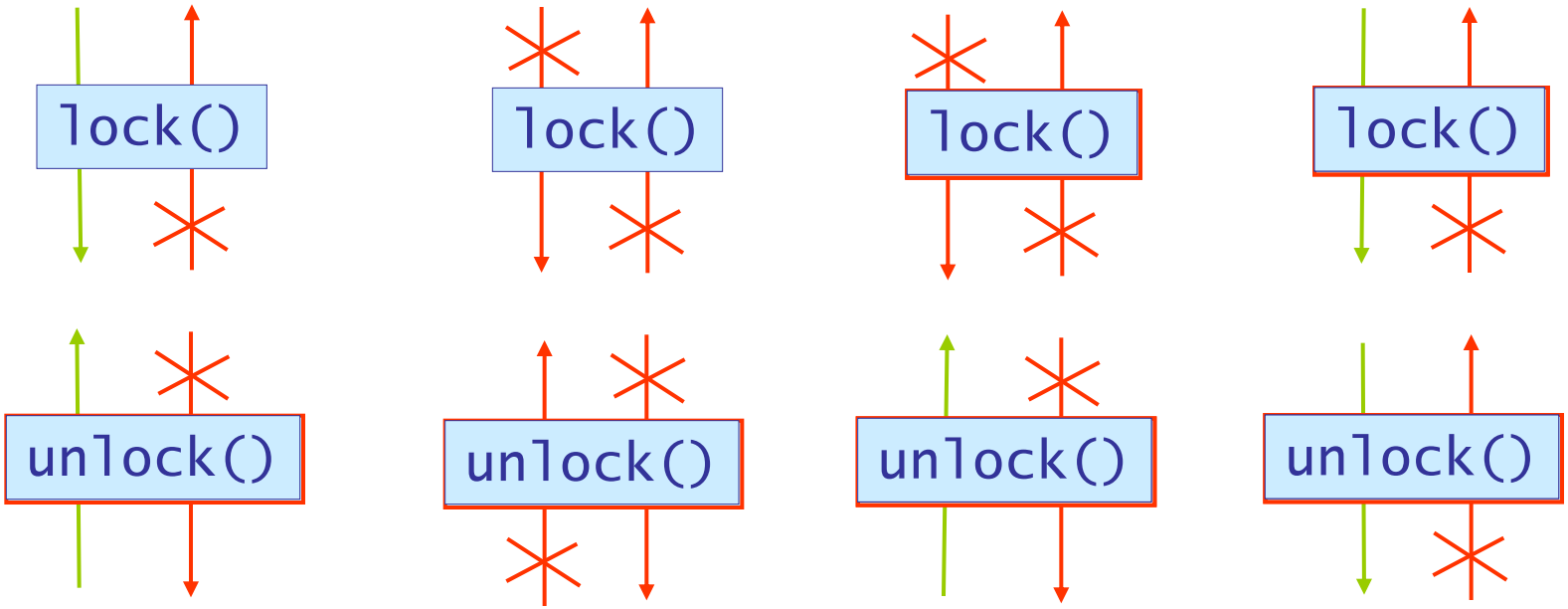
Naïve pthreads:



Optimized pthreads



Some open source pthread lock implementations (2006):



[technically incorrect]

NPTL

{Alpha, PowerPC}

{mutex, spin}

[Correct, slow]

NPTL

Itanium (&X86)

mutex

[Correct]

NPTL

{ Itanium, X86 }

spin

[Incorrect]

FreeBSD

Itanium

spin