

# The C++0x concurrency memory model and some of its implications

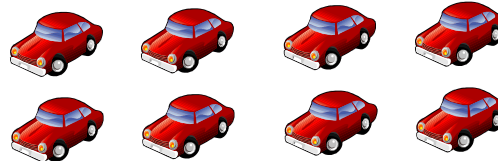
*Hans Boehm*, 

# Acknowledgements & Disclaimer

- This is an ongoing standardization effort.
- Many others contributed, including:
  - Sarita Adve (UIUC, also did much of the underlying work long ago)
  - Herb Sutter (Microsoft)
  - Clark Nelson, Bratin Saha (Intel)
  - Doug Lea (SUNY Oswego)
  - Paul McKenney (IBM)
  - Lawrence Crowl, Jeremy Manson (Google)
  - Bill Pugh (U. of Maryland)
  - ...
- Some things may still change ...

# Why threads?

- Future processor performance enhancements:
  - mostly from multi-core processors
  - far less from single-core performance



- Programmability is often a limiting factor.
- For parallelizing single non-numerical apps:
  - **Threads and locks dominate.**
    - Because they're there. And the alternatives are still research or less general.
    - Most applications on your desktop are probably already multithreaded

# Threads basics

- Multithreaded execution can be viewed as interleaving:

- “Dekker’s” example (everything initially zero):

*Thread 1*

`x = 1;`

`r1 = y;`

*Thread 2*

`y = 1;`

`r2 = x;`

- Might be executed as:

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

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

\*Provided certain esoteric library calls are avoided.

# The problem

- Canonical example (everything initially zero):

*Thread 1*

`x = 1;`

`r1 = y;`

*Thread 2*

`y = 1;`

`r2 = x;`

- No interleaving can result in `r1 = r2 = 0`.
- Most hardware allows this (write buffers).
- Standard compiler transformations also cause this.

# The Basic Solution

- Prohibit *data races*.
- Defined as follows:
  - Two memory operations *conflict* if they access the same memory location and at least one is a store operation.
  - An execution (interleaving) contains a *data race* if two conflicting operations corresponding to different threads are adjacent (maybe executed concurrently).
  - $x = 1; y = 1; r1 = y; r2 = x;$  has data race.

# The Implementation Promise

If your program does not allow a data race, it will behave as though program steps are simply interleaved.

(Sequential consistency for data-race-free programs.)

# Some variants

C++0x	Basic promise, Undefined race semantics, + non-SC escapes
C1x	Same as C++0x
Java	Basic promise, Complex race semantics + few non-SC escapes
Ada83+, Posix threads	Basic promise (more or less, sort of)
.Net, OpenMP, Fortran	Getting there, we hope 😊

# How do we avoid data races?

- Locks:
  - No `lock(l)` can appear in the interleaving unless prior `lock(l)` and `unlock(l)` calls from other threads balance.
- Atomic variables:
  - Allow concurrent access. “Exempt” from data races.
  - Called `volatile` in Java, *but not in C/C++*.
  - Dekker’s example “works” with atomics.

# 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)

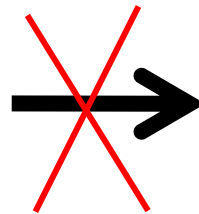
# Basic characteristics

- Compiler/hardware can continue to reorder accesses
  - Compiler doesn't know about threads.
  - Only a racing thread can tell the difference.
- Careful treatment of synchronization operations.

# Standard treatment of synchronization

- `Pthread_mutex_lock()` etc. are opaque to compiler.
  - Viewed as potentially modifying any location
  - Memory operations cannot be moved past them.
- `Pthread_mutex_lock()` etc. contain “sufficient fences” to prevent hardware reordering.
- Together these prevent:

```
lock();  
    tmp = x;  
    x = tmp + 1;  
unlock();
```



```
tmp = x;  
lock();  
    x = tmp + 1;  
unlock();
```

# So isn't this all obvious?

- Perhaps it should have been.
- But a few things went wrong in the past ...

# What went wrong? (1)

- Single thread compilers can 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.

# Compiler introduced races: (a)

- Part of the problem:
  - Need to state how much memory is touched as the result of an assignment.
  - Can an assignment to a data member overwrite adjacent data members?
  - What's a “memory location”?
- C++0x answer:
  - Scalar object or contiguous bit-field sequence.

# Compiler introduced races (b)

- May the compiler introduce speculative stores?
- Can the compiler vectorize

```
for (i = 1; i < N; ++i)  
    if (a[i] != 1) a[i] = 2;
```

as

```
for (i = 1; i < N; ++i)  
    a[i] = ((a[i] != 1)? 2 : a[i]);
```

- No!
- But most compilers do things along these lines.

# Very different examples as well:

- Posix rules arguably allow:

```
[g is global]
for(...) {
    if(mt) lock();
    use/update g;
    if(mt) unlock();
}
```

- lock calls "synchronize memory"



```
r = g;
for(...) {
    if(mt) {
        g = r; lock(); r = g;
    }
    use/update r instead of g;
    if(mt) {
        g = r; unlock(); r = g;
    }
}
g = r;
```

# What went wrong? (2)

- It's unrealistic to avoid data races without some “escape” mechanism. (PLDI05)
  - E.g. for simple atomic counters, flags, ...
- PLDI 2007 had a paper (Narayanasami et al.) on distinguishing benign and harmful races.
- C++0x provides `atomic<T>` with a comprehensive set of operations.

# What went wrong? (3)

- Uncertainty about details.
- How is a data race defined?
  - C++0x: With respect to interleaving semantics.
  - Generated debate on Posix mailing list.
    - E.g.  $x, y$  initially zero:  
*Thread 1: if (x) y = 1;*  
*Thread 2: if (y) x = 1;*
- If there are no data races, what behavior can the programmer expect?
  - C++0x: Sequential Consistency.
  - Posix is not very clear.

# What went wrong? (4)

Unclear that hardware can ensure sequential consistency. “IRIW” example:

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

*Thread 1:*

```
x = 1;
```

*Thread 2:*

```
r1 = x; (1)
```

```
fence;
```

```
r2 = y; (0)
```

*Thread 3:*

```
y = 1;
```

*Thread 4:*

```
r3 = y; (1)
```

```
fence;
```

```
r4 = x; (0)
```

*x* set first!

*y* set first!

We had long discussions on whether this is allowed, with what fences, on several architectures.

# What went wrong? (5)

- `TryLock()` semantics had weird consequences:
  - More later.

# C++ specifics

- Sequential consistency for data-race-free programs (except loopholes), as in Java,
- But a few things are different:
  - Undefined semantics for data races.
    - Java can't do this for security reasons.
    - Allows compiler to assume *no asynchronous changes*.
  - Weaker `tryLock()` semantics.
  - Extensive, but very explicit, loopholes.
  - Clearer statement on libraries.

# Undefined semantics for data races

- Java can't do this for security reasons.
- Allows compiler to assume *no asynchronous changes*.
  - Seems common, but hard to exercise
- OK to store/load integers and pointers a byte at a time.
- Avoids fences in constructors: Data race  
→ bad method table pointers are OK.

# “No asynchronous changes” compiler assumption consequences:

```
unsigned x;  
  
If (x < 3) {  
    ... // async x change  
    switch(x) {  
        case 0: ...  
        case 1: ...  
        case 2: ...  
    }  
}
```

27 April 2009

- Assume switch statement compiled as branch table.
- May assume **x** is in range.
- Asynchronous change to **x** causes wild branch.
  - Not just wrong value.

# Trylock restricts `lock()` reordering:

- Some really awful code:

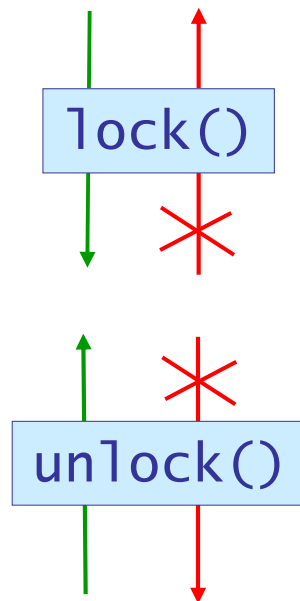
<i>Thread 1:</i>	<i>Thread 2:</i>	<i>Don't try this at home!!</i>
<pre>x = 42; lock();</pre>	<pre>while (trylock() == SUCCESS)     unlock(); assert (x == 42);</pre>	

- Can't move `x = 42` into critical section!
- Disclaimer: Example requires tweaking to be pthreads-compliant.

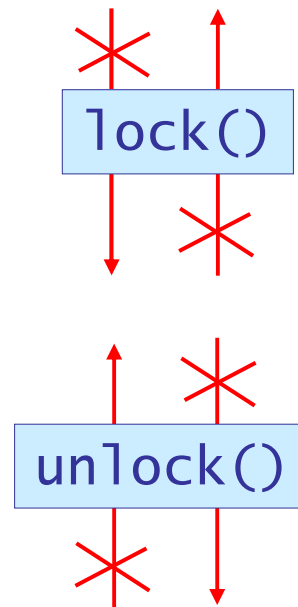
# Trylock: Critical section reordering?

- Reordering of memory operations with respect to critical sections:

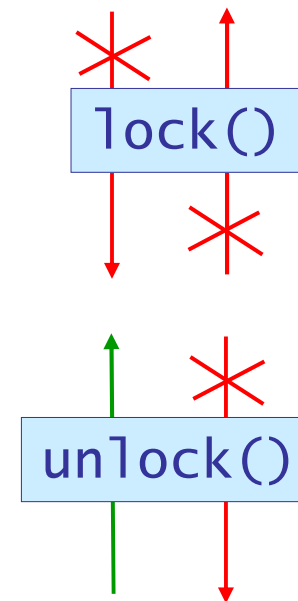
Expected (& Java):



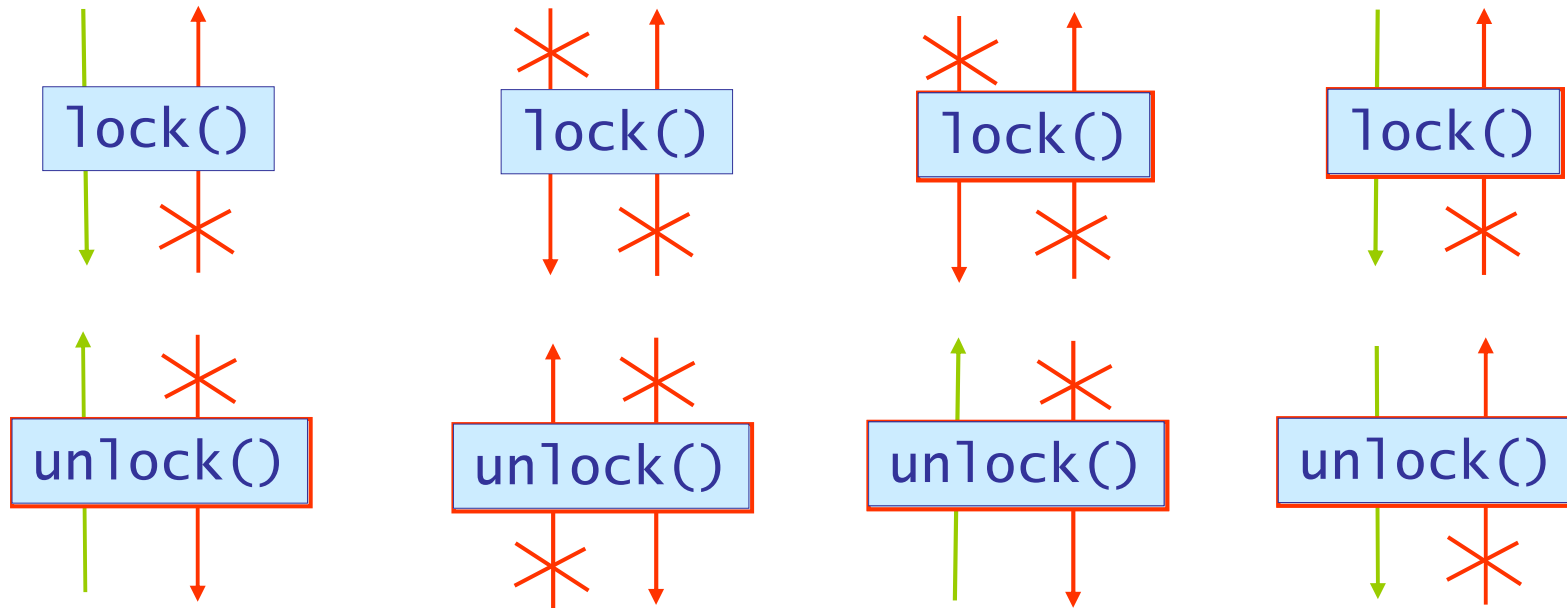
Naïve pthreads:



Optimized pthreads



# Some open source pthread lock implementations (2006):



[technically incorrect]

NPTL

{Alpha, PowerPC}

{mutex, spin}

27 April 2009

[Correct, slow]

NPTL

Itanium (&X86)

mutex

[Correct]

NPTL

{ Itanium, X86 }

spin

[Incorrect]

FreeBSD

Itanium

spin

27

# C++0x solution:

- `TryLock()` and the like may fail spuriously.
- The problematic example has a data race and is thus incorrect.
- Code movement *into* critical sections is allowed, as in Java.

# Escapes from sequential consistency

- Data-race-free programs are sequentially consistent unless:
  - They use operations on atomics with an explicit `memory_order...` argument.
  - Provides for “relaxed” and “acquire/release” ordering and explicit fences.
  - Tricky: Initial example won’t work.
  - Hard to hide in libraries.
  - Significantly complicates specification in standard.
  - Occasionally necessary for current hardware.

# Libraries

- By default:
- Library objects (e.g. containers) behave like scalars:
  - Concurrent reads are allowed.
  - Concurrent accesses to different objects are allowed.
  - Other concurrent accesses introduce data races, and must be avoided with client locking.

# Immediate Consequences

- Hardware
  - Enforce sequential consistency
  - Byte store granularity
- Compiler back-ends
  - Must fix speculative store problems

# Architecture, the good news (mid 2008):

- Intel and AMD published much better descriptions (summer 2007, summer 2008).
- Can get sequential consistency including for “IRIW” example.
- Other major architectures also appear OK, some with performance issues for sequential consistency.

# Most common reason for speculative stores

```
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
```

# More speculative consequences

- Current hardware ISAs appear suboptimal.
- Suggests an approach to atomic section/transactional memory semantics.

# Implementing sequentially consistent atomics 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

- `a` is atomic:

```
x = 1;
```

```
a = 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.
- *We need a tiny fraction of `mfence` functionality.*

# Better solution?

- Preserve atomic operations to the machine instruction level?
- Also enables hardware data race detection?
- Not a new proposal.
  - But we finally have the language foundations in place.

# What does this imply about transactional memory?

- Transactional memory removes need to specify locks:

```
{ lock_guard _(mtx); S; }
```



```
__tm_atomic { S; }
```

- Big advantage:
  - No lock-based deadlocks.
  - Avoids need to enforce lock ordering.

# A simple semantics for atomic blocks

- Can be modeled as acquiring a single global lock.
- Allows simple interleaving based semantics
  - Unlike traditional software transactional memory implementations.
  - Shared and private data structures can continue to share source code.
- Increasing consensus on this approach, but
  - Performance issues remain.
  - Do atomic sections provide more? (abort on failure?)