



Getting C++ Threads Right

Hans-J. Boehm
HP Labs



Why threads?

- Future processor performance enhancements:
 - mostly from multi-core processors
 - far less from single-core performance
- Programmability is often a limiting factor.
- Lots of research on programming models:
 - Transactional memory
 - Better message passing models
 - More flexible data parallel models
- But for parallelizing single non-numerical apps:
 - **Threads and locks dominate.**
 - Because they're there. And the alternatives are still research or less general.

Threads (contd.)

- Threads and locks have a bad reputation.
 - Many horror stories of intermittent failures.
 - General feeling that things are too complex.
 - “Locks don’t compose.”
 - “Threads are Evil.”
- My claim:
 - Not all of this is rooted in profound problems.
 - Sometimes, we just got it wrong the first time around.
 - It just wasn’t so important until now.
 - And most of it is fixable.

Disclaimers:

- Not all of the insights are mine.
- Others who helped in identifying/fixing problems: Sarita Adve, Lawrence Crowl, Doug Lea, Paul McKenney, Jeremy Manson, Clark Nelson, Bill Pugh, Bratin Saha, Herb Sutter, ...
- Some of the insights are cheap shots based on 20/20 hindsight.
- Some of the solutions are obvious.
 - That's part of the point.

Our role

- Identify the problems.
- Resolve them whenever possible:
 - Some technical solutions are nontrivial.
 - C++ standards committee effort.
 - Plan: Provide for proper thread support in C++0X.
 - Note: X = 9 is the goal, X = 0xa is quite possible.
 - C++ working paper currently includes:
 - Threads “memory model” defining shared variable values seen by a thread.
 - Atomic types and operations library.
 - Threads API (almost).

Our role (2)

- Resolve problems whenever possible:
 - Talking to C and Posix committees.
 - C appears likely to adopt memory model and atomics library.
 - Talk to others, e.g. processor vendors, to address issues.
 - Things are getting better,
 - e.g. new Intel & AMD X86 memory model specifications.
 - Trying to spread the word to resolve past misconceptions.

Rest of talk:

- List some of the problems.
- Explain expected resolutions to the extent they're known.
 - For language issues, most of these are in the C++0x working paper, but not yet in a formal standard.
- Try to convince you that
 - There are problems here,
 - But they are solvable.

Starting with the hardware at the bottom:

- Everything from threads implementations to user code depends on memory consistency/ordering.

- Canonical example (everything initially zero):

Thread 1

`x = 1;`

`r1 = y; // reads 0`

Thread 2

`y = 1;`

`r2 = x; // reads 0`

Most hardware allows this (write buffers).

- Programmers tend to initially assume sequential consistency, i.e. program executes by interleaving thread actions è not allowed
- Determined by compiler, thread libraries, and hardware.
- Everything relies on understanding of hardware properties.

Hardware at the bottom (contd.):

- This is not fundamentally a problem:
 - Provided we understand hardware rules, and can use them to implement a usable programming model.
- Widely held beliefs (?):
 - Weaker memory consistency (e.g. allowing this) is fine, since:
 - We only pay for ordering (special fence instructions) when needed.
 - Should be cheaper.
 - Fence instructions get us sequential consistency (acts like thread actions are simply interleaved) exactly when we need it.

Memory ordering and fences, reality, ca. 2006:

- Some architectures, **underspecified memory ordering** è
 - **Confusion**. (Mutually reinforced by higher layers?)
 - Interesting consequences. For X86:
 - gcc `__sync_synchronize()` “full memory barrier” erroneously generates no-op (& perhaps has to).
 - P4 **lfence**, **sfence** instructions appeared to be no-ops in most user code. But the spec didn't say that.

Memory ordering and fences, pre-2006 reality:

- Performance of fences and synchronization was often neglected.
 - More than 100 cycles, best case on P4. (Improving ...)
 - Encourages
 - clever synchronization avoidance techniques = bugs.
 - Can easily be much more expensive than sequential consistency everywhere. (PA-RISC)
- Memory models in which it appeared that fences could not enforce sequential consistency
 - and hence Java memory model is not really implementable.

Fences and sequential consistency

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

Thread 1:

$x = 1;$

Thread 2:

$y = 1;$

Thread 3:

$r1 = x; (1)$

fence;

$r2 = y; (0)$

x set first!

Thread 4:

$r3 = y; (1)$

fence;

$r4 = x; (0)$

y set first!

Architecture, the good news (late 2007):

- Intel and AMD published much better descriptions.
- Can get sequential consistency for examples like the preceding one.
 - But stores need to be implemented with `xchg`.
 - Many JVMs probably still need to be fixed.
- Other vendors are paying attention, and mostly also have good stories.

Languages and compilers

- Programming rules have been unclear.
 - Java memory model “fixed” in 2005.
 - But some ongoing issues.
 - .NET and OpenMP rules could be clearer.
 - We’ll focus mostly on C/C++ with thread library here.

General C/C++ programming rule:

- No simultaneous access to ordinary shared variables if one access is a write. I.e. **no data races**.
 - E.g. Posix.
 - Dates back to at least Ada83.
- Solves many problems:
 - Can't tell whether compiler reorders ordinary memory operations.
 - If you could tell, observing thread would be racing with updating thread.
 - Can't tell whether hardware reorders memory operations (so long as locks etc. are handled)
 - C/C++ compilers may rely on the absence of asynchronous changes. This may have weird effects ...

Possible effect of “no asynchronous changes” compiler assumption:

```
unsigned x;

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

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

C and C++ threads realities

- Common attitude that data races aren't so bad.
 - Frequently used idioms rely on data races:
 - Approximate counters sometimes without locking on update, and read asynchronously. (Works badly even if you get lucky.)
 - Double-checked locking: Lazy initialization that reads flag outside of critical section.
 - Or nonportable atomic ("Interlocked", "__sync") operations:
 - E.g. reference counting.
 - Not well-defined, and read accesses generally appear as data races to compiler.
 - This is can result in crashes, reads of half-updated values, uninitialized data, etc.
 - But locks are expensive enough that this is often impractical to avoid.

And there's "volatile"

- 2005 Java:

store to `volatile x_init`:

```
x = ...; x_init = true;
```

guarantees that `x` becomes visible before `x_init`.

Implementation may require fence **before** `volatile` store.

- OpenMP 2.5 (and 3.0 draft):

"... a reference that modifies the value of an object with a `volatile`-qualified type behaves as if there were a flush operation on that object **at the next** sequence point."

- C/Pthreads:

[Dave Butenhof:] "`volatile` ... provide[s] no help whatsoever in making code 'thread safe'"

C++0x WP solution:

- Concurrent access to special atomic objects, e.g. `atomic<int>` is allowed.
- Other than that, this is really just a communication issue.
- There are still no “benign data races” in C++ programs.
- The C++ `volatile` qualifier continues to have nothing to do with threads.
- Important note: Java `volatile` • C++ `atomic`.
 - Unfortunate, but existing C++ `volatile` uses are incompatible with Java semantics.

But there are more technical problems:

- **Compilers may** generate code that (unpredictably?) **adds** data **races**.
 - Can happen when small struct fields are updated.
 - Details follow
 - Can also happen in more complicated cases if optimization results in spurious write of old value.
 - And very occasionally, this does happen.
 - And current standards allow it.

Struct field update:

```
struct {char a; int b: 5; int c: 11; char d;} x;
```

- Is it safe to protect `c` and `d` with separate locks?

Thread1:

```
x.c = 1;
```

Thread2:

```
x.d = 1;
```

implemented as

Thread1:

```
tmp = x;
```

```
tmp.c = 1;
```

```
x = tmp;
```

Thread2:

```
x.d = 1;
```

Example issue 1 (contd):

```
struct {char a; int b: 5; int c: 11; char d;} x;
```

- Is it safe to protect **c** and **d** with separate locks?

Thread1:

```
tmp = x;  
tmp.c = 1;  
x = tmp;
```

Thread2:

```
x.d = 1;
```

x: a: 0; b: 0; c: 0; d: 0;

Example issue 1 (contd):

```
struct {char a; int b: 5; int c: 11; char d;} x;
```

- Is it safe to protect **c** and **d** with separate locks?

Thread1:

```
➤ tmp = x;  
tmp.c = 1;  
x = tmp;
```

Thread2:

```
x.d = 1;
```

x: a: 0; b: 0; c: 0; d: 0;

tmp: a: 0; b: 0; c: 0; d: 0;

Example issue 1 (contd):

```
struct {char a; int b: 5; int c: 11; char d;} x;
```

- Is it safe to protect **c** and **d** with separate locks?

Thread1:

```
tmp = x;
```

```
➤ tmp.c = 1;
```

```
x = tmp;
```

Thread2:

```
x.d = 1;
```

x: a: 0; b: 0; c: 0; d: 0;

tmp: a: 0; b: 0; c: 1; d: 0;

Example issue 1 (contd):

```
struct {char a; int b: 5; int c: 11; char d;} x;
```

- Is it safe to protect **c** and **d** with separate locks?

Thread1:

```
tmp = x;
```

```
➤ tmp.c = 1;
```

```
x = tmp;
```

Thread2:

```
➤ x.d = 1;
```

x: a: 0; b: 0; c: 0; d: 1;

tmp: a: 0; b: 0; c: 1; d: 0;

Example issue 1 (contd):

```
struct {char a; int b: 5; int c: 11; char d;} x;
```

- Is it safe to protect **c** and **d** with separate locks?

Thread1:

```
tmp = x;
```

```
➤ tmp.c = 1;
```

```
x = tmp;
```

Thread2:

```
x.d = 1;
```

x: a: 0; b: 0; c: 0; d: 1;

tmp: a: 0; b: 0; c: 1; d: 0;

Example issue 1 (contd):

```
struct {char a; int b: 5; int c: 11; char d;} x;
```

- Is it safe to protect `c` and `d` with separate locks?

Thread1:

```
tmp = x;  
tmp.c = 1;  
x = tmp;
```

Thread2:

```
x.d = 1;
```



x: a: 0; b: 0; c: 1; d: 0;

- This behavior is currently allowed and common.

§No two fields can safely be independently updated.

C++0x working paper solution:

- Subject to “no data races” rule:
 - Each update affects a “memory location”.
 - Scalar value, or contiguous sequence of bit-fields.
 - Define exactly which assignments can be “seen” by each reference to a memory location.
 - For ordinary (non-atomic) references, there must be exactly one, for atomics there can be several.
 - A reference to `x.d` after completion of both threads must see a value of 1.
 - The preceding implementation of bit-field assignments is incorrect.
 - Assignments `x.b` and `x.c` bit-fields may still interfere.

More subtle implication of C++0x rule:

- Compiler may not introduce “speculative” stores:

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



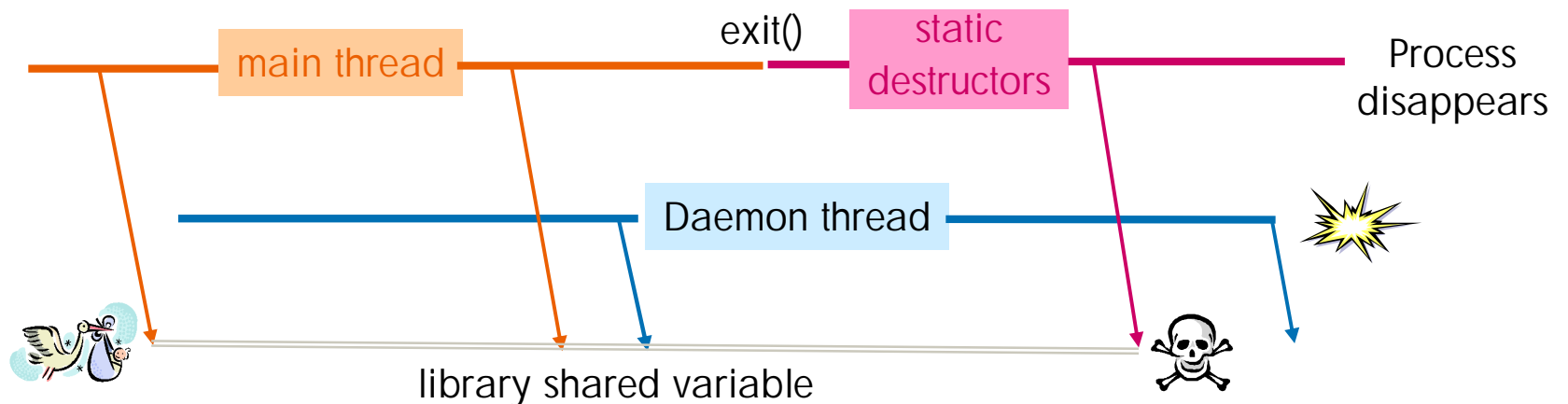
```
int count;      // global, possibly shared
...
reg = count;
for (p = q; p != 0; p = p -> next)
    if (p -> data > 0) ++reg;
count = reg;    // may spuriously assign to count
```

Consequences

- Outlaws some useful optimizations, but
- Gives the programmer a simple and consistent story, and
- Prevents really mysterious compiler-introduced program bugs.
 - Which have occasionally been observed in practice.
- And the outlawed optimizations can often be replaced by others.

Other language features:

- So far, we just needed assignments to break things.
- C++ destructors are interesting:



C++0x treatment of static destructors

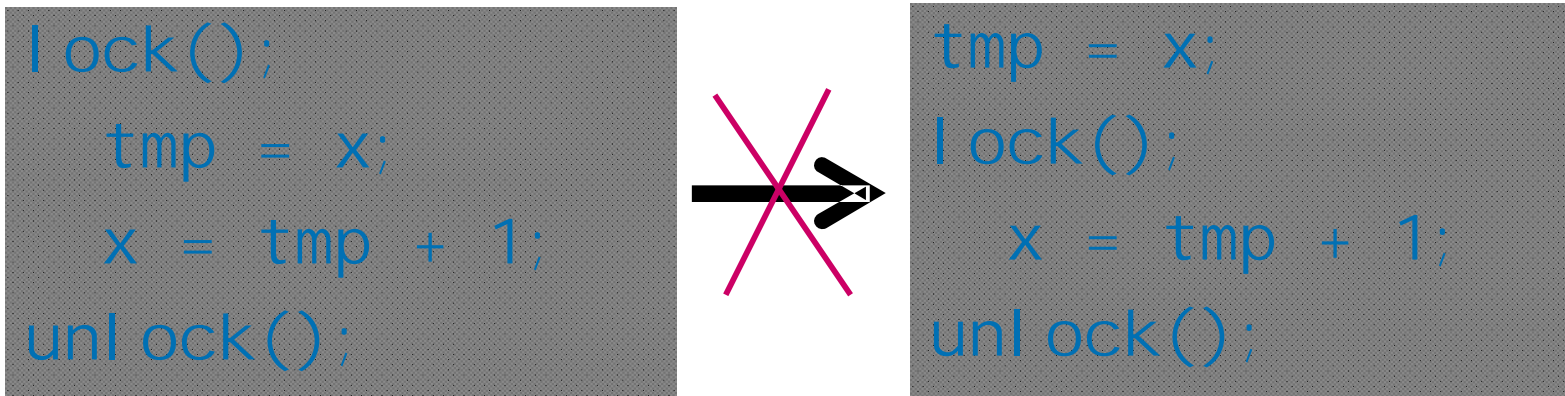
- Only partially satisfactory solution: Either
 - Shut down all threads before process exit.
 - (Hard if they are waiting on IO.)

or

- Execute only special cleanups before exit (not destructors).

And thread libraries:

- We have to limit reordering of memory operations with respect to synchronization operations:



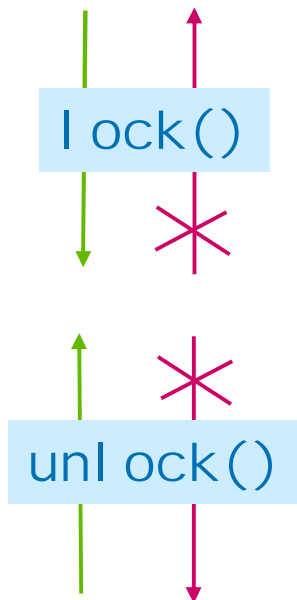
- This is normally done in two ways:
 - Compiler treats synchronization functions as opaque
 - i.e. as though they might change `x`
 - Synchronization routines contain expensive fence instructions
 - Prevents hardware reordering.

But:

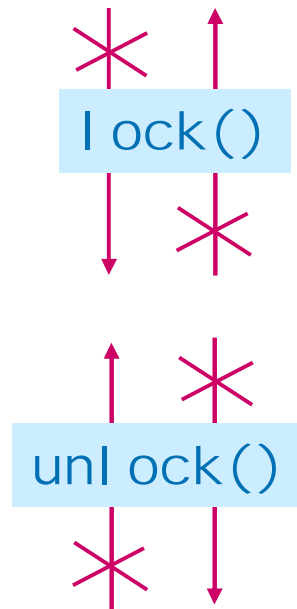
Unclear what reordering is allowed:

- Reordering of memory operations with respect to critical sections:

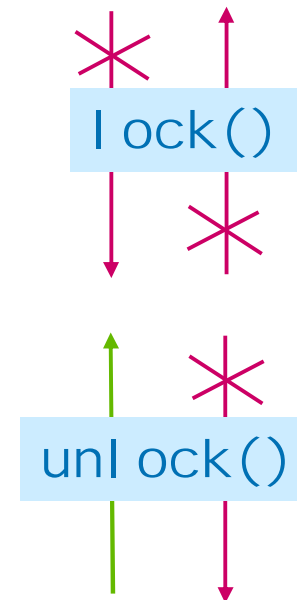
Expected (& Java):



Naïve pthreads:



Optimized pthreads



Pthreads doesn't allow expected reordering because:

- Some really awful code would break:

Thread 1: Thread 2: **Don't do this!!**

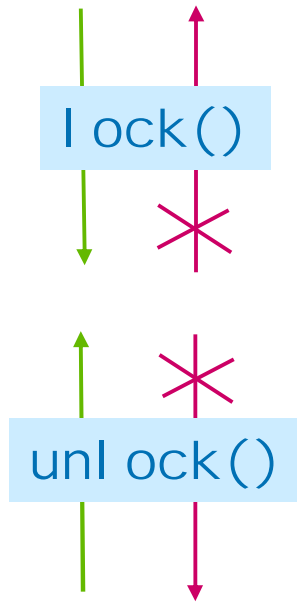
```
x = 42;
lock();
```

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

- Reordering thread 1 statements is wrong.
 - This is a movement into critical section.
- But this issue wasn't recognized until recently.
- Disclaimer: Example requires tweaking to be pthreads-compliant.



Some open source pthread lock implementations (2006):

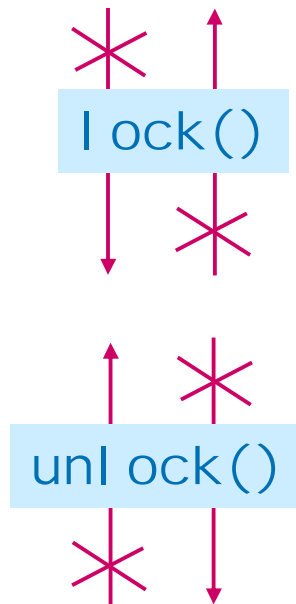


[technically incorrect]

NPTL

{Alpha, PowerPC}

{mutex, spin}

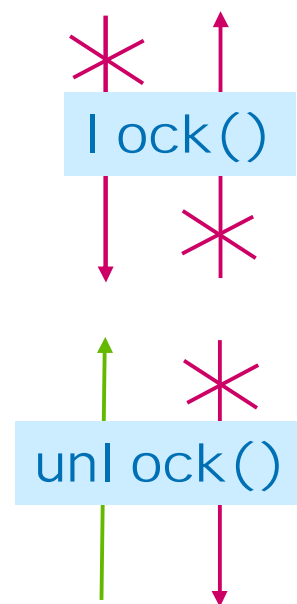


[Correct, slow]

NPTL

Itanium (&X86)

mutex

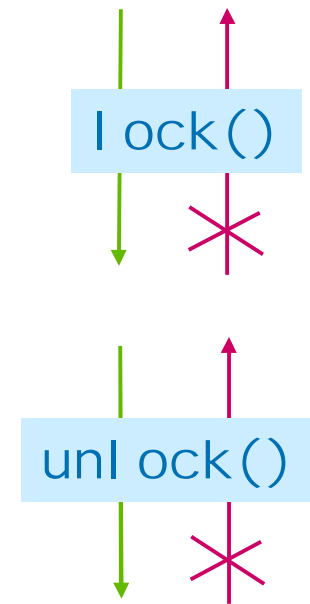


[Correct]

NPTL

{ Itanium, X86 }

spin



[Incorrect]

FreeBSD

Itanium

spin

C++0x solution:

- Movement into critical section is allowed.
- `Trylock()` may fail even if lock is available!
 - Problematic examples are now clearly incorrect.
 - Which was our goal.
 - Library vendors are informally discouraged from taking advantage of this.
 - Worsens performance.
- Allows the standard to guarantee that memory operation reordering is invisible for data-race-free programs.
 - That don't use some low-level library facilities.
 - Such data-race-free programs behave as though thread steps are simply interleaved (sequentially consistent).

And libraries in general:

- General wisdom about locks in e.g. general purpose container libraries:
 - **Lock in the client!**
 - Only client knows about sharing, the right granularity.
 - Unexpected library-based locking causes:
 - Performance problems.
 - Deadlocks.
- But:
 - Original Java collections like **Vector** are synchronized.
 - including individual element access!
 - As is Posix **putc**.
 - But not C++ STL container implementations.

C++0x solution:

- Mostly follow the de facto STL convention.
- Details TBD
- We're not done yet ...

Where does this leave us?

- Threads as an afterthought are dubious.
- It's time to get the remaining problems fixed.
- Maybe there really is a better general purpose parallel programming model ...
 - But that's hard to evaluate without getting threads and locks right first.
 - Especially since we will probably have threads and locks underneath anyway.



