



Threads in C++0x: An Update

Hans-J. Boehm
HP Labs



Disclaimers:

- This is a standards committee effort. Many people helped.
- Others who helped in identifying/fixing problems: Sarita Adve, Lawrence Crowl, Howard Hinnant, Doug Lea, Paul McKenney, Jeremy Manson, Clark Nelson, Bill Pugh, Bratin Saha, Herb Sutter, ...
- Most of what I describe here is now in the working paper – but it may still change – and small pieces are likely to change.
- The threads API is almost entirely the work of others; I'm likely to have gotten some small things wrong.
- C++0x may well be a misnomer. 2010?

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 still dominate.**
 - Because they're there. And the alternatives are still research or less general.

Threads: The Problem

- A lot of performance critical code is written using a combination of:
 - ISO C++ (or ISO C) ← single threaded language
 - Threads library (Pthreads or Windows threads)
- But
 - Code using threads is not portable.
 - Some aspects of thread behavior are really a language, not library issue.
 - Programming rules unclear, hard to teach.
 - Occasionally misbehaving code
 - even if everyone follows the rules.
 - Atomic operations are necessary, but not well supported.

C++0x adds

- Threads API
 - Allows threads use in portable C++ code.
- Concurrency memory model
 - When do changes to objects become visible to other threads?
 - Contract between programmer and compiler/hardware.
- Atomics library
 - Shared variables without locks.
 - Usually for performance.
 - Sometimes to avoid deadlocks.
- Threads fully supported in the language
 - Higher level concurrency libraries expected to be added later.

This talk: go over these in order, explaining some of the underlying issues.

Memory model and atomics are somewhat inseparable.

Threads API (N2497)

- Based on Boost, with some improvements:
 - Thread creation
 - `Quick_exit()` and process shutdown
 - Mutexes, locks
 - Condition variables
- Some other relevant language changes:
 - Lambda expressions (anonymous function objects)
 - Function local statics are thread-safe
 - Thread-local storage

Threads API: Thread creation

```
class thread {  
    public:  
        class id;  
        // omitting native handles, swap(), ...  
        // movable, not copyable  
        template <class F> explicit thread(F f);  
        template <class F, class ...Args>  
            thread(F&& f, Args&&... args);  
        bool joinable() const;  
        void join();  
        void detach();  
        id get_id() const;  
        ...  
        static unsigned hardware_concurrency();  
};
```

Thread creation example:

```
int fib(int n) {
    if (n <= 1) return n;
    int fib1, fib2;
    std::thread t([=, &fib1]{fib1 = fib(n-1);});
    fib2 = fib(n-2);
    t.join();
    return fib1 + fib2;
}
```

Disclaimers:

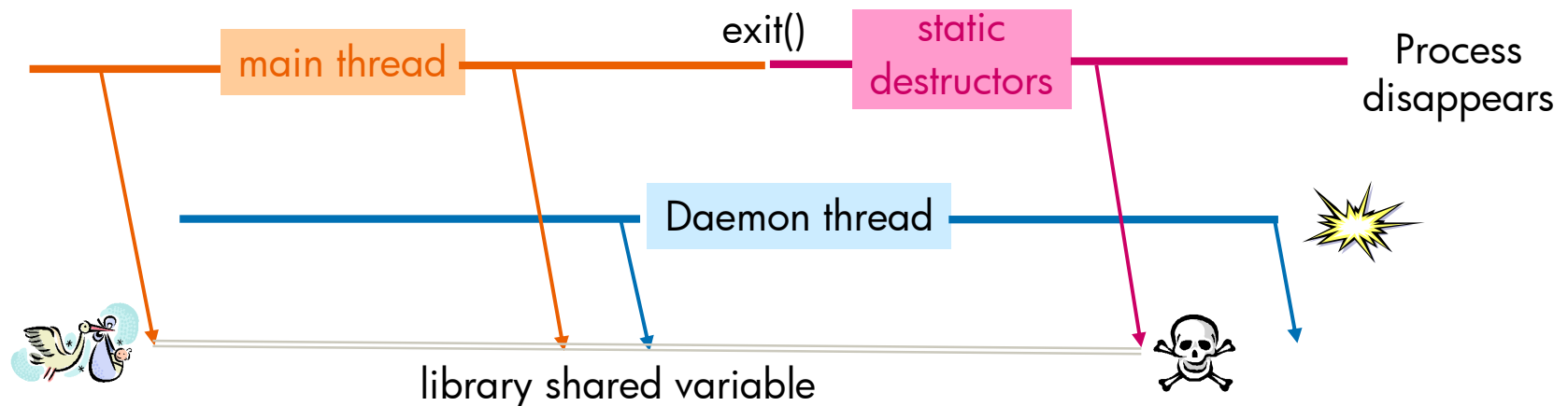
- Untested code!
- Don't really do this! It creates too many threads.
- Runs in exponential time. There is a $\log(n)$ algorithm.
 - Except that it overflows for interesting inputs.

Potential gotcha: Watch object lifetimes!

- The preceding code cannot handle exception thrown by `fib`!
 - If `fib2` computation throws, thread `t` is *detached*.
 - Thread `t` will still write to `fib1`, which will be long gone.
- Such runaway threads may also make library calls after main exits.
 - See next slide.
- Always make sure that threads are shut down when expected.
 - Thread join should be done even for exception.

Process shutdown:

- C++ static destructors can cause problems:



- Even standard library is unsafe to use after `exit()`
 - except that threads may return after `main()` calls `exit()`

Process shutdown: `quick_exit()`

- We need either:
 - A way to shut down threads before `exit` from `main()`
 - A way to kill a process without making libraries unusable,
 - i.e. without running static destructors.
- The former is hard:
 - Need to shut down threads blocked on I/O
 - Tried and failed to get something that played with Posix cancellation.
- `quick_exit()` gives us the latter.
 - Something between `exit()` and `_exit()`

Mutual Exclusion

- Real multi-threaded programs usually need to access shared data from multiple threads.
- For example, incrementing a counter in multiple threads:

- `x = x + 1;`

- Unsafe if run from multiple threads:

```
tmp = x; // 17
```

```
x = tmp + 1; // 18
```

```
tmp = x; // 17
```

```
x = tmp + 1; // 18
```

Mutual Exclusion (contd)

- Standard solution:
 - Limit shared variable access to one thread at a time, using locks.
 - Only one thread can be holding lock at a time.

Mutexes

```
class mutex {  
    public: mutex();  
    ~mutex();  
    mutex(const mutex&) = delete;  
    mutex& operator=(const mutex&) = delete;  
    void lock();  
    bool try_lock();  
    void unlock();  
    ...  
};
```

- Class `recursive_mutex` is similar:
 - allows *same* thread to acquire mutex mutiple times.

Counter with a mutex

```
mutex m;
```

```
void increment() {  
    m.lock();  
    x = x + 1;  
    m.unlock();  
}
```

- Lock not released if critical section throws.

Lock_guard

```
template <class Mutex>
class lock_guard {
public:
    typedef Mutex mutex_type;
    explicit lock_guard(mutex_type& m);
    lock_guard(mutex_type& m, adopt_lock_t);
    ~lock_guard();
    lock_guard(lock_guard const&) = delete;
    lock_guard& operator=(lock_guard const&) = delete;
private:
    mutex_type& pm; // for exposition only
};
```

Counter with a lock_guard

```
mutex m;
```

```
void increment() {  
    lock_guard _(m);  
    x = x + 1;  
}
```

- Lock is released in destructor.

Condition variables: Waiting on shared state to change

```
class condition_variable {  
    public: ...  
        void notify_one();  
        void notify_all();  
        void wait(unique_lock<mutex>& lock);  
        template <class Predicate>  
            void wait(unique_lock<mutex>& lock, Predicate pred);  
        template <class Duration>  
            bool timed_wait(unique_lock<mutex>& lock,  
                           const Duration& rel_time);  
};
```

- class `condition_variable_any` deals with arbitrary mutex types.

Concurrency Memory Model (N2429)

- What does it mean to share memory?
- 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).
- Standard compiler transformations also cause this.
- Programmers tend to initially assume *sequential consistency*, i.e. program executes by interleaving thread actions → not allowed

Existing(?) 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.

Canonical example, again:

- (everything initially zero):

Thread 1

`x = 1;`

`r1 = y; // reads 0`

Thread 2

`y = 1;`

`r2 = x; // reads 0`

- This has a data race:
 - `x` and `y` can be simultaneously read and updated.
- Has undefined behavior.
- Unless `x` and `y` are declared to have `atomic` type.
 - In which case the compiler has to do what it takes to preclude this outcome.

C++0x approach

- Data races are (still?) disallowed!
 - No notion of a “benign” data race.
 - And this time we’re not kidding.
 - We give you the tools to avoid them:
 - Library provides `atomic<T>` type.
 - Effectively you need to identify data races.
- If you avoid data races (and some low-level constructs)
- We guarantee sequential consistency.
 - Programs behave as though steps are interleaved.
- Compiler may not introduce visible data races!
- Consistent with Java approach, though weaker.

This has compiler consequences:

- 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
```

It also has hardware consequences

- Hardware must be able to provide sequential consistency if all shared variables are declared as `atomic<T>`.
- Hardware usually provides “memory fence” instructions to prevent memory operation reordering.
- At hardware level:

Thread 1

`x = 1;`

`fence;`

`r1 = y;`

Thread 2

`y = 1;`

`fence;`

`r2 = x;`

`r1` and `r2` cannot both be zero.

But it's not clear fences are enough!

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!

This was not clearly disallowed by public X86 hardware manuals.

Intel, AMD provided new descriptions (summer 07) that made it possible to avoid this.

Atomic operations may have to be compiled differently.

A note on “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.

- C++0x:

Declare `x_init` as `atomic<bool>` for same effect.

- C/Pthreads:

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

Remains true, but it may be useful as a pre-C++0x work-around.

- 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.”

Mistake? I believe this will disappear.

But we don't have a complete solution:

- We've been sloppy about defining "data race".
- And it matters.

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;
```

Struct field update (contd 1):

```
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;

Struct field update (contd 2):

```
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;

Struct field update (contd 3):

```
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;

Struct field update (contd 4):

```
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;

Struct field update (contd 5):

```
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;

Struct field update (contd 6):

```
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;

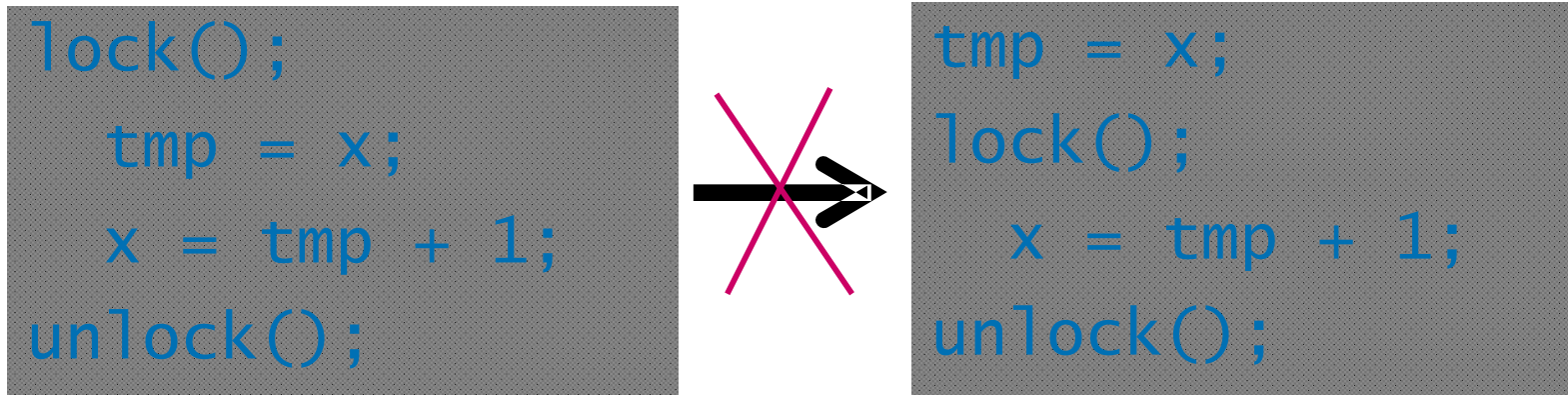
- This behavior is currently allowed and common.
 - No two fields can safely be independently updated.

C++0x working paper solution:

- Each update affects a “memory location”.
 - Scalar value, or contiguous sequence of bit-fields.
- Read accesses access a memory location.
- Two accesses conflict if
 - They access the same memory location,
 - At least one updates the location.
 - They are not both accesses to an `atomic` variable or other synchronization object.
- A data race occurs if two conflicting accesses can occur simultaneously.
- Without a race, we require sequential consistency.
- There is no data race in the preceding example →
 - 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.

Also affects thread library implementations:

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

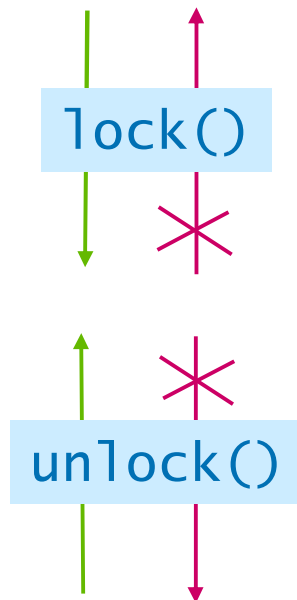


- This is normally done in two ways:
 - Compiler treats synchronization functions as opaque
 - i.e. as though they might change `x`
 - Synchronization routines contain expensive fence instructions
 - Prevents hardware reordering.

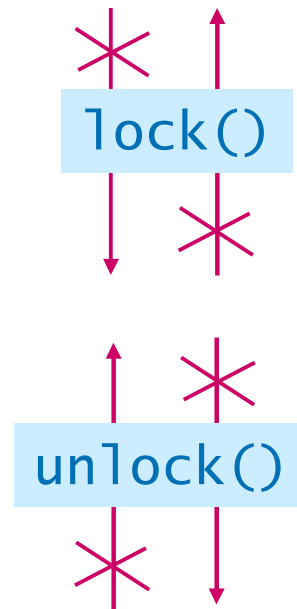
But: Unclear what reordering is allowed:

- Reordering of memory operations with respect to critical sections:

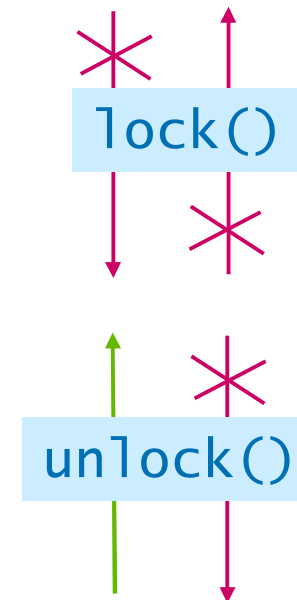
Expected (& Java):



Naïve pthreads:



Optimized pthreads



Pthreads doesn't allow expected reordering because:

- Some really awful code would break:

Thread 1:

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

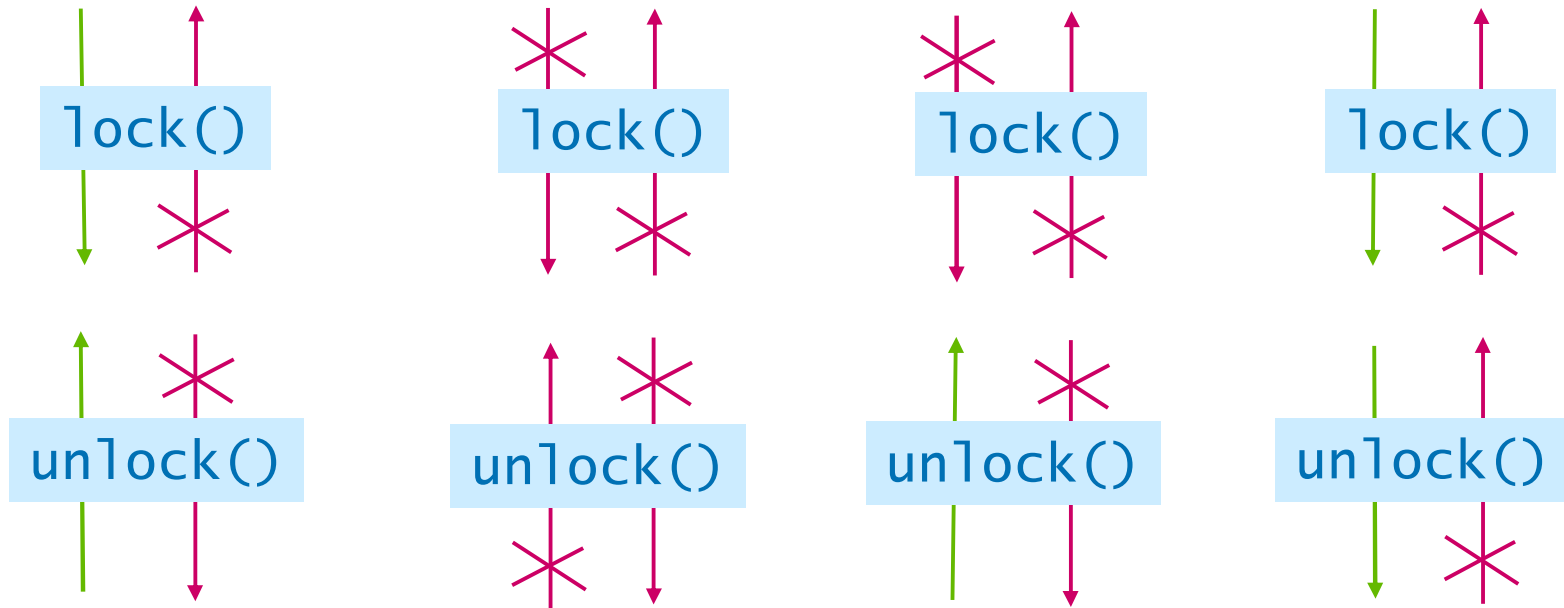
Thread 2:

Don't do this!!

```
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 low-level atomic operations.

Atomics library (N2427)

- The problem:
 - Would like to implement, for example, counters, without locks.

```
atomic<int> x;  
  
void increment() {  
    ++x;  
}
```

instead of

```
int x;  
mutex m;  
  
void increment() {  
    lock_guard _(m);  
    x = x + 1;  
}
```

- Advantages:
 - Sometimes enables much better performance (last year's talk)
 - No space for locks, sometimes simpler.
 - Potentially safe for use with signal handlers, across processes.

Many other atomics applications

- Very common example: double-checked locking:

```
T x;
atomic_bool x_init(false);
mutex m;

if (!x_init) {
    lock_guard _(m);
    if (!x_init) {
        x = ...
        x_init = true;
    }
}
use x;
```

- Note: Atomics are still tricky. Only a single memory operation at a time is atomic!

Atomics needs

- Want shared variables
 - that can be concurrently updated without introducing “data race”,
 - that are atomically updated and read
 - “half updated” states are not visible,
 - that are implemented without lock overhead whenever the hardware allows,
 - that provide access to hardware atomic read-modify-write (fetch-and-add, xchg, cmpxchg) instructions whenever possible.

Atomics, ca. 2007:

- Various compilers provide various nonstandard extensions.
 - Gcc and others provide `__sync...` operations.
 - Windows provides `Interlocked...` operations.
 - Various attempts at more general libraries
 - e.g. our `atomic_ops` library.
- Semantics nonstandard, often confused and incomplete.
 - Generally no atomic loads. But also no guarantee that ordinary accesses are atomic. Or that `volatile` works.
 - Memory ordering implications are unclear.
 - E.g., gcc has `__sync_synchronize()` “full memory barrier”,
 - but it erroneously generates no-op on X86 (& perhaps has to).

C++0x atomics

```
template< T > struct atomic {  
    // Simplified, some details in flux ...  
    explicit atomic( T );  
    atomic( const atomic& ) = delete;  
    atomic& operator =( const atomic& ) = delete;  
    void store( T ) volatile;  
    T load( ) volatile;  
    T operator =( T ) volatile;    // similar to store()  
    T operator T ( ) volatile;  
        // equivalent to load(), currently not in WP  
    T swap( T ) volatile;  
    bool compare_swap( T&, T) volatile;  
    bool is_lock_free() const volatile;  
};
```

C++0x atomics, contd.

- `compare_swap(expected, desired)` atomically performs

```
if (load() == expected) {
    store(desired);
    return true;
} else {
    expected = load();
    return false;
}
```
- Allows atomic updates $x = f(x)$ as

```
tmp = x;
do {
    new_val = f(tmp);
} while (!x.compare_swap(tmp, new_val));
```

Implementation

- Most modern hardware has instructions to implement the atomic operations
 - for small types
 - and bit-wise comparison, assignment (which we require)
- For other types, hardware, atomic operations must be emulated with locks.
 - Sometimes this isn't good enough:
 - across processes, in signal/interrupt handlers.
 - `is_lock_free()` returns `false` if locks are used, and operations may block.

Other atomics facilities

- Specializations for integral types, pointers
 - Provide atomic increment, decrement (`++`, `--`, `+=`, `-=`)
 - Note: `x++` is very different from `x = x + 1` !
 - Unlike Java `volatile`s, where both are probably wrong!
- Non-template (C-like) atomic types
 - Template specializations inherit from these
- C-like stand-alone (`atomic_`) function interfaces.

“Low-level” atomic operations

- Most atomic operations actually provide an optional memory ordering argument.
 - Removes sequential consistency guarantee
 - which often allows really weird behavior!
 - Reintroduces weird outcomes in previous examples.
 - But sometimes gives significantly better performance.
 - For example:

```
if (!x_init.load(memory_order_acquire) {  
    lock_guard _(m);  
    if (!x_init.load(memory_order_relaxed)) {  
        x = ...  
        x_init.store(true, memory_order_release);  
    }  
}  
use x;
```

is likely to be appreciably faster on PowerPC.

Summary

- C++0x provides APIs to program at three levels:
 1. Threads + locks + condition variables.
 - Gives you control over size of critical sections.
 2. (1) + atomic operations.
 - Allows improved performance, occasionally simplification.
 - Once you try to avoid locks, and get past the easy cases, this can get very hard.
 3. (2) `_` low-level (explicitly ordered) atomics
 - You need to understand more of the memory model (1.10) than I've presented here.
 - Experts only.
 - And the experts usually get it wrong.

Sequentially
consistent
(data-race-free)



Status

- API, memory model, atomics, are in C++0x working paper.
- Other pieces are still moving through process:
 - thread-local storage
 - some low-level atomics refinements
- Probably not the long pole in getting C++0x out the door.
 - “concepts” (typed templates) probably have that honor.
- Memory model, atomics under consideration by C committee.
- Reasonably consistent with the Java memory model:
 - Aside from `volatile` vs. `atomic<T>`.
- Working with OpenMP committee towards consistency.



