Threads and Shared Variables in C++0x

Hans-J. Boehm

HP Labs

Disclaimers:

- This describes the work of many people.
- Major contributors to work on the memory model and atomic operations: Sarita Adve, Lawrence Crowl, Paul McKenney, Clark Nelson, Herb Sutter, …
- The threads API is almost entirely the work of others; I'm likely to have gotten some small things wrong.
- C++0x is a misnomer. It's likely to be C++11.

Outline

- Overview
- Threads API
- Basic memory model
- A note on detached threads
- Basic atomic objects
- Performance consequences
  - and how to avoid them

What are threads?

- Multiple instruction streams (programs) that share memory.
- Static variables, and everything they point to, are shared between them.
- Each thread has its own stack and thread-local variables.

Why threads?

- A convenient way to process multiple event streams.
- The dominant way to take advantage of multiple cores for a single application.

Naive threads programming model (Sequential Consistency)

- Threads behave as though their operations 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;
```

Threads in C++0x

- Threads are finally part of the language! (C1x, too)
- Threads API
  - Thread creation, synchronization, …
  - Evolved from `Boost.Thread`.
- Memory model
  - What exactly do shared variables mean?
    - Not quite the naïve sequential consistency model.
  - When does thread *a* see an update by thread *b*?
  - When is it OK to simultaneously access variables from different threads?
- Atomic operations
- `thread_local` variables, parallel constructor execution, thread-safe function-local statics

Outline

- Overview
- Threads API
- Basic memory model
- A note on detached threads
- Basic atomic objects
- Performance consequences
  - and how to avoid them

Threads API: Thread creation

```
class thread {
   public:
         class id;
         // movable, not copyable
         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();
         // + native handles, swap(), …
};
```

Thread creation example:

```
int fib(int n) {
   if (n <= 1) return n;
   int fib1, fib2;
   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 Boost threads gotcha: Detached threads are hazardous!

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

- What if parent call to `fib` throws?
  - In Boost, if `fib2` computation throws, thread `t` is *detached*.
    Thread t contiinues to run independentaly.
  - Thread t will still write to `fib1`, which will be long gone.
- In C++0x, destroying a joinable thread calls `terminate()`!
- Always join!
- More on `detach()` later …

A safer way to write parallel `fib()`

```
int fib(int n) {
  if (n <= 1) return n;
  int fib2;
  auto fib1 =
     async([=]{return fib(n-1);});
  fib2 = fib(n-2);
  return fib1.get() + fib2;
}
```

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
                                tmp = x;   // 17
x = tmp + 1; // 18
                                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 restrict interleavings

Thread 1

```
m.lock();
r1 = x;
x = r1+1;
m.unlock();
```

Thread 2

```
m.lock();
r2 = x;
x = r2+1;
m.unlock();
```

– can only be executed as

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

or

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

since second `m.lock()` must follow first `m.unlock()`

C++0x Mutexes

```
class mutex {
   public: mutex();
   ~mutex();
   mutex(const mutex&) = delete;
   mutex& operator=(const mutex&) = delete;
   void lock();
   bool try_lock();     // may fail even if lock available!
   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<mutex> _(m);
   x = x + 1;
}
```

- Lock is released in destructor.
- `unique_lock<>` is a generalization of `lock_guard<>`.

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.

Outline

- Overview
- Threads API
- Basic memory model
- A note on detached threads
- Basic atomic objects
- Performance consequences
  - and how to avoid them

Let's look back more carefully at shared variables

- So far threads are executed as though thread steps were just interleaved.
  - *Sequential consistency*
- But this provides expensive guarantees that reasonable code can't take advantage of.

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;
```

- Compilers like to perform loads early.
- Hardware likes to buffer stores.

Sensitive to memory access granularity

Thread 1

```
x = 300;
```

Thread 2

```
x = 100;
```

- If memory is accessed a byte at a time, this may be executed as:

```
x_high = 0;
x_high = 1;    // x = 256
x_low =  44;   // x = 300;
x_low = 100;   // x = 356;
```

And this is at too low a level …

- 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 by different threads 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 `atomic` (synchronization) variables
    - (wait a few slides…)
- In C++0x, there are ways to explicitly relax the sequential consistency guarantee.

Dekker’s example, again:

- (everything initially zero):

```
Thread 1                    Thread 2
x = 1;                      y = 1;
r1 = y; // reads 0          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.

Data races → undefined behavior: Very strange things may happen

```
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.

A note on data race definition

- 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;
```

Another note on data race definition

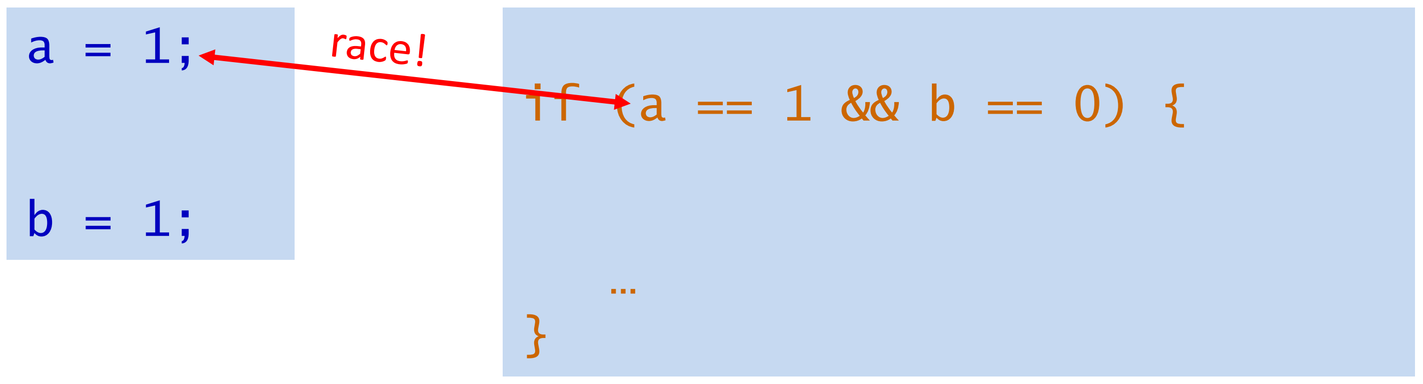
- We define it in terms of scalar accesses, but …
- Container libraries should ensure that

  Container accesses don't race ➔

  No races on memory locations
- This means
  - Accesses to hidden shared state (caches, allocation) must be locked by implementation.
  - User must lock for container-level races.
- This is often the correct library thread-safety condition.

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:



Thread 1 (not atomic):

```
a = 1;


b = 1;
```

Thread 2(observer):

```

if (a == 1 && b == 0) {


    …
}
```

Basic Implementation model

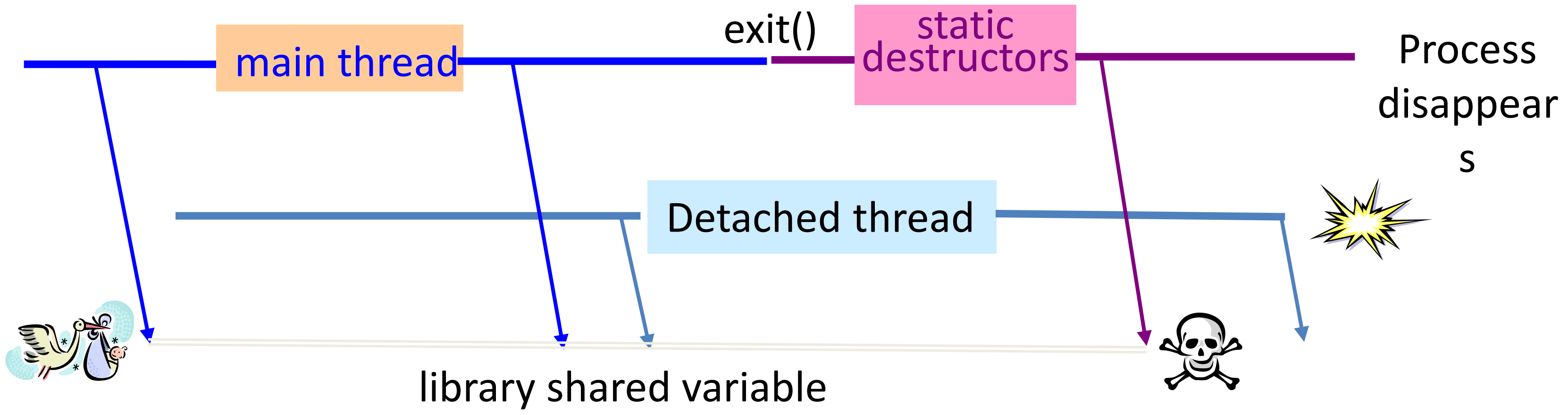
- Very restricted reordering of memory operations around synchronization operations:
  - Compiler either understands these, or treats them as opaque, potentially updating any location.
  - Synchronization operations include instructions to limit or prevent hardware reordering (“memory fences”).
- Other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

Outline

- Overview
- Threads API
- Basic memory model
- A note on detached threads
- Basic atomic objects
- Performance consequences
  - and how to avoid them

A note on detached threads:

- C++ static destructors can cause problems:



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

Options for detached threads

- Wait for them to terminate, possibly after some sort of shutdown request.
  - Unfortunately, there is no thread cancellation.
  - But then why detach?
- Exit without calling static destructors (`quick_exit()`)
- Just don't call `detach()`. (My personal favorite.)

Outline

- Overview
- Threads API
- Basic memory model
- A note on detached threads
- Basic atomic objects
- Performance consequences
  - and how to avoid them

Synchronization variables

- C++0x: `atomic<int>, atomic_int`
- C1x: `_Atomic(int), _Atomic int, atomic_int`
- *not* C++ `volatile`!
- Java: `volatile`, `java.util.concurrent.atomic`.
- C# : none, though `volatile` is closest.
- 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 are "escapes" in C++0x.
- Dekker's algorithm "just works" with synchronization variables.

C++0x atomics

```
template< T > struct atomic {
  // Greatly simplified, for now
  constexpr atomic( T ) noexcept;
  atomic( const atomic& ) = delete;
  atomic& operator =( const atomic& ) = delete;
  void store( T ) noexcept;
  T load( ) noexcept;
  T operator =( T ) noexcept;   // similar to store()
  T operator T () noexcept;    // equivalent to load()
  T exchange( T ) noexcept;
  bool compare_exchange_weak( T&, T) noexcept;
  bool compare_exchange_strong( T&, T) noexcept;
  bool is_lock_free() const noexcept;
};
```

C++0x atomics, contd

- Integral, pointer specializations add atomic increment operators.
- Atomic to atomic assignment intentionally not supported.
  - But it is in C1x!

Outline

- Overview
- Threads API
- Basic memory model
- A note on detached threads
- Basic atomic objects
- Performance consequences
  - and how to avoid them

Performance impact of DRF with sequentially consistent atomics

- Some optimization restrictions (compiler and hardware).
  - But those should have been there all along.
  - (and maybe some of them were?)
- Sequentially consistent atomic operations must
  - Ensure that these operations appear to be executed in order ➔ fences on all major current architectures.
    - Possible with a fence for every store on (revised) X86.
  - Ensure that ordinary memory operations are not visibly reordered w.r.t. atomic operations.
    - Free on X86, sometimes requires more fences
- Fence instructions are typically expensive.

New compiler restrictions

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

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

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

$\longrightarrow$

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

  - `x.a = 1` in parallel with `x.b = 1` may fail to update x.b.
- Still broken in gcc in subtle cases involving bit-fields.

43

Some restrictions are a bit more annoying:

- Compiler may not introduce “speculative” stores:

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

$$\cancel{\Downarrow}$$

```
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
```

Also some hardware restrictions

- Multiprocessors need fast byte stores.
- Should be able to implement sequential consistency without locks, e.g. by adding fences.
  - You might have thought this was obvious …
  - Took years to confirm for X86, PowerPC!

Performance costs

- Compiler restrictions typically minor cost
  - Assuming sane optimizations to start with.
- Fence costs for sequentially consistent atomics are potentially much larger.
  - C++0x also allows non-SC atomics.
    - and even explicit memory fences.
  - Double-edged sword:
    - Faster. Especially on some non-X86 architectures.
    - Really hard to use correctly.
    - We don't generally know how to hide library uses.
    - Initially controversial. Maybe deprecate after hardware adjusts?

C++0x explicitly ordered (low-level) atomics

- Pairs of atomic operations cannot form a data race.
- Operations that do not specify `memory_order_seq_cst` (the default) are not guaranteed to execute in a single total order.
- A `memory_order_release` store still guarantees memory visibility to `memory_order_acquire` load that reads the value.

```
atomic<bool> flag;
```

Thread 1:

```
   data = 42;
   flag.store(true, memory_order_release);
```

Thread 2:

```
   if (flag.load(memory_order_acquire)){
     assert (data == 42)
```

Dekker’s with C++0x low-level atomics

atomic<int> x, y;

Thread 1:

```
x.store(1,memory_order_release);
r1 = y.load(memory_order_acquire);
```

Thread 2:

```
y.store(1,memory_order_release);
r2 = x.load(memory_order_acquire);
```

- `r1` = `r2` = 0 *is possible outcome*.
- No acquire operations reads release result ➔ no constraints.
- Same as `memory_order_relaxed`.
- Allows ordinary `MOV` on X86, much cheaper on PowerPC.

Other memory_order options

- A `memory_order_relaxed` operation also drops acquire/release visibility requirement.
- But operations *on a single variable* still behave as though they were interleaved (cache coherent).
- A `memory_order_consume` operation behaves like `memory_order_acquire`, but only with respect to subsequent data-dependent operations.

Safe uses for low-level atomics

- Use `memory_order_relaxed` if no concurrent access to an atomic is possible.
- Use `memory_order_relaxed` to atomically update variables (e.g. increment counters) that are only read with synchronization.
- Use `memory_order_release` / `memory_order_acquire`, when it's OK to ignore the update, at least for some time (?)

C++0x fine-tuned double-checked locking

```
atomic<bool> x_init;

if (!x_init.load(memory_order_acquire) {
   l.lock();
   if (!x_init.load(memory_order_relaxed) {
      initialize x;
      x_init.store(true, memory_order_release);
   }
   l.unlock();
}
use x;
```

Summary

- C++0x provides APIs to program at three levels:
  1. Threads + locks + condition variables.
     - Traditional threads programming.
  2. (1) + atomic operations.
     - Allows improved performance, occasionally simplification.
     - Easy (e.g. counters) are straightforward. General lock-free programming is 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.

$\Big\}$ Sequentially consistent (data-race-free)

Questions?

Backup slides

Language spec challenge:

- *Some really awful code:*

Thread 1:

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

?

Thread 2:

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

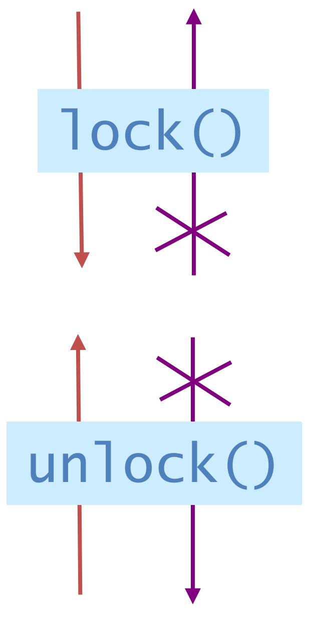
Don't try this at home!!

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

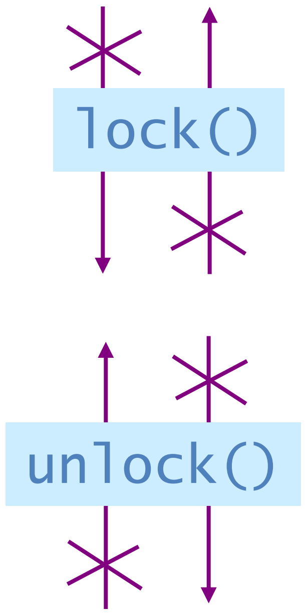
- Can the assertion fail?
- Many implementations: Yes
- Traditional specs: No. C++0x: Yes
- `Trylock()` can effectively fail spuriously!

55

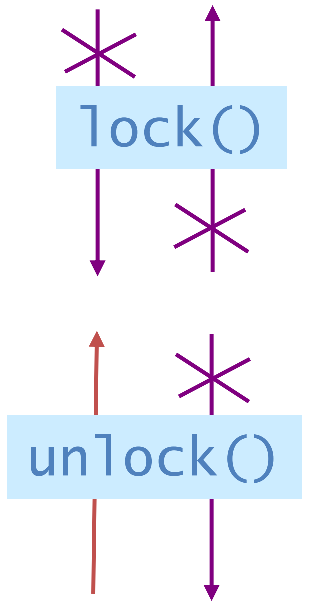
Some open source pthread lock implementations (2006):



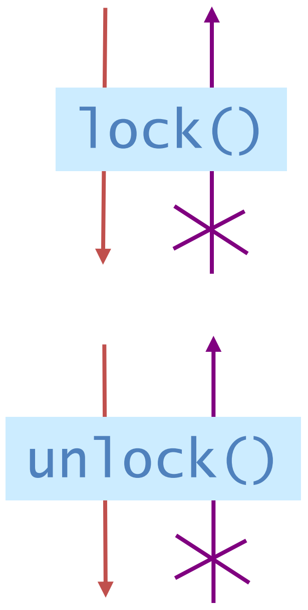
[technically incorrect]
NPTL
{Alpha, PowerPC}
{mutex, spin}



[Correct, slow]
NPTL
Itanium (&X86)
mutex



[Correct]
NPTL
{ Itanium, X86 }
spin



[Incorrect]
FreeBSD
Itanium
spin

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)
```

Thread 4:
```
r3 = y; (1)
fence;
r4 = x; (0)
```

x set first!

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.