Threads and Shared Variables in C++11 and elsewhere

Hans-J. Boehm

HP Labs
Credits:

• This describes work done by many people, mostly as part of the ISO C++ standards committee. Other major contributors include: Lawrence Crowl (Sun/Google), Clark Nelson (Intel), Herb Sutter (Microsoft), Paul McKenney (IBM).

• Some of it is heavily based on earlier academic research, notably by Sarita Adve

• ... and that doesn’t include the many people who worked on other parts of the language, such as the threads API itself.
Outline

• Overview
• C++11 Threads API (very briefly)
• C++11/C11 Memory model
• Understanding data races
• Atomic objects
• A word about Java
• Conclusions
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?

• Controversial:
  – “Threads are evil” gets around 20K Google hits, but
• Threads are a convenient way to process multiple event streams, and
• 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**
\[
\begin{align*}
x &= 1; \\
z &= 3;
\end{align*}
\]

**Thread 2**
\[
\begin{align*}
y &= 2; \\
r1 &= x;
\end{align*}
\]

- might be executed as
\[
\begin{align*}
x &= 1; \\
y &= 2; \\
r1 &= x; \\
z &= 3;
\end{align*}
\]
Pre-C++11/C11 Threads in C & C++

• Single-threaded language + Threads API
  – e.g. Posix, Windows

• Exact meaning of shared variables unclear:

```c
char x, y;  // Class members
```

```
Thread 1
x = 1;
```

```
Thread 2
y = 1;
```

Are `x` and `y` set to 1 when both finish?
– Posix: Implementation defined. Windows: ???

• + Much more complicated ways for things to go wrong (very rarely)
• No consistent story to teach programmers.

20 April 2012
Threads in C++11

• Threads are finally part of the language! (C11, too)
• Threads API
  – Thread creation, synchronization, ...
  – Evolved from Boost.Thread.
• Memory model
  – Carefully defines shared variable behavior.
    • Still not quite the naïve sequential consistency model.
• Atomic operations
• Condition variables, call_once, thread_local variables, parallel constructor execution, thread-safe function-local statics
Outline

• Overview
• C++11 Threads API (very briefly)
• C++11/C11 Memory model
• Understanding data races
• Atomic objects
• A word about Java
• Conclusions
Thread creation example:

```c
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:
- `fib(n-2)` throws ➔ no join ➔ bad things happen
- Don’t really do this! It creates too many threads.
- Easier to use `async()`, which returns a `future`.
- Runs in exponential time. There is an $O(\log(n))$ algorithm.
  - Except that they all overflow for interesting inputs.
Thread creation rules

• Always call `join()`!
  – Language provides `detach()`, `quick_exit()`, but ...

• Destroying an unjoined thread invokes `terminate()`.
  – Makes exceptions in parent much safer.

• Program terminates when main thread returns.
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:
  \[
  \text{tmp} = x; \quad // \quad 17 \\
  x = \text{tmp} + 1; \quad // \quad 18 \\
  \]
  \[
  \text{tmp} = x; \quad // \quad 17 \\
  x = \text{tmp} + 1; \quad // \quad 18 \\
  \]
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()
Counter with C++11 mutex

mutex m;

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

• Lock not released if critical section throws.
Counter with a `lock_guard`

```cpp
mutex m;

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

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

• Overview
• C++11 Threads API (very briefly)
• C++11/C11 Memory model
• Understanding data races
• Atomic objects
• A word about Java
• Conclusions
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 $r_1 = r_2 = 0$:

  Thread 1
  \[
  \begin{align*}
  x &= 1; \\
  r_1 &= y;
  \end{align*}
  \]

  Thread 2
  \[
  \begin{align*}
  y &= 1; \\
  r_2 &= x;
  \end{align*}
  \]

• 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:

  \[ \begin{align*}
  \text{x\_high} & \text{ = 0;} \\
  \text{x\_high} & \text{ = 1; } // x = 256 \\
  \text{x\_low} & \text{ = 44;} // x = 300; \\
  \text{x\_low} & \text{ = 100;} // x = 356;
  \end{align*} \]
And this is at too low a level ...

• 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) Data race definition

• 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) A useful restriction

• Sequential consistency only for data-race-free programs!
• Catch-fire semantics for data races!
• Data races are prevented by
  – mutexes (or atomic sections) to restrict interleaving
  – declaring atomic (synchronization) variables
    • (wait a few slides...)
• In C++11, there are ways to explicitly relax the sequential consistency guarantee.
Dekker’s example, again:

• (everything initially zero):

  Thread 1                             Thread 2
  \[ x = 1; \quad y = 1; \]
  \[ r1 = y; \quad \text{// reads 0} \quad r2 = x; \quad \text{// reads 0} \]

• 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 \text{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

```c
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.
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:

\[
\begin{align*}
\text{Thread 1 (not atomic):} & & \text{Thread 2(observer):} \\
\text{Thread 1:} & \quad a = 1; \\
& \quad b = 1; \\
& \quad \text{race!} \\
\text{Thread 2:} & \quad \text{if } (a == 1 \&\& b == 0) \{ \\
& \quad \quad \text{...} \\
& \quad \} \\
\end{align*}
\]
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
• C++11 Threads API (very briefly)
• C++11/C11 Memory model
• Understanding data races
• Atomic objects
• A word about Java
• Conclusions
Understanding data races

• To show that a program behaves correctly:
  1. Demonstrate there are no data races
     • assuming sequential consistency
  2. Demonstrate that it behaves correctly
     • Assuming sequential consistency, and
     • Assuming synchronization-free-regions are indivisible

• Some examples:
• Assume \texttt{x} and \texttt{done} are initially 0/false.
• Consider:

\begin{align*}
\text{Thread 1} & \quad \text{Thread 2} \\
\text{x} &= 42; & \text{while} \ (!\text{done}) \{} & \\
\text{done} &= \text{true}; & \text{assert}(\text{x} == 42); &
\end{align*}

\textbf{Data race on done.}

Frequently breaks repeatably in practice.
Lazy initialization and DCL

• Assume \( x \) and \( \text{initd} \) are initially 0/false.
• Consider:

\[
\begin{align*}
\text{Thread 1} & \quad \text{Thread 2} \\
\text{if (!initd) { } } & \quad \text{if (!initd) { } } \\
\quad \text{lock\_guard<mutex> _(m);} & \quad \text{lock\_guard<mutex> _(m);} \\
\quad x = 42; & \quad x = 42; \\
\quad \text{initd} = \text{true;} & \quad \text{initd} = \text{true;} \\
\} & \quad \} \\
\text{read } x; & \quad \text{read } x; \\
\end{align*}
\]

Data race on \( \text{initd} \).
Often works in practice, but not reliable.
• Assume \( x \) and \( y \) are initially zero.

• Consider:

\[
\begin{align*}
\text{Thread 1} & \\
\quad & \text{if (x)} \\
\quad & \quad y = 1;
\end{align*}
\[
\begin{align*}
\text{Thread 2} & \\
\quad & \text{if (y)} \\
\quad & \quad x = 1;
\end{align*}
\]

No data race.

But that was unclear before C++11.
• struct { char a; char b; } x;

• Consider:

  Thread 1
  x.a = 1;

  Thread 2
  x.b = 1;

No data race in C++11 or C11.

But there may be one under older Posix rules.
• `struct { int a:8; int b:8; } x;`

• Consider:
  
  `Thread 1`
  
  `x.a = 1;`

  `Thread 2`
  
  `x.b = 1;`

Data race!
• struct { char a; int b:11; } x;

• Consider:

  
  Thread 1
  
  \[ x.a = 1; \]
  
  Thread 2
  
  \[ x.b = 1; \]

No data race.

But existing compilers commonly introduce a data race.
• `list<int> x;`

• Consider:
  
  \[
  \begin{align*}
  \text{Thread 1} & \quad \text{Thread 2} \\
  x.\text{push}\_\text{front}(1); & \quad x.\text{pop}\_\text{front}();
  \end{align*}
  \]

Data Race.

Data races are defined for scalar accesses. Default rule for libraries:

Race on scalars $\leftrightarrow$ Race on object
• `list<int> x; mutex m;`

• Consider:

  ```cpp
  Main Thread
  occasionally
  { lock_guard<mutex> _(m);
    x.push_front(1);
  }
  
  Thread 2
  for(;;) {
    lock_guard<mutex> _(m);
    if(!x.empty()) …
  }
  ```

Data Race.

*Thread 2 races with x’s destructor.*

*(That’s why `thread::detach()` is discouraged.)*
• Consider:

\[
\begin{align*}
\text{Thread 1} & : & \left\{ \begin{array}{l}
\text{for (;} ; ; \text{) {}} \\
x = 1;
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
\text{Thread 2} & : & x = 2;
\end{align*}
\]

No data race

but undefined behavior anyway.

Infinite loops that perform neither IO nor synchronization have undefined behavior!
• `int x; mutex m;`

• Consider:

  Thread 1
  
  ```c
  x = 42;
  m.lock();
  ```

  Thread 2
  
  ```c
  while (m.try_lock())
  m.unlock();
  assert(x == 42);
  ```

Data Race.

`try_lock()` may fail spuriously.

(Reality is complicated. This simple rule works.)
Outline

• Overview
• C++11 Threads API (very briefly)
• C++11/C11 Memory model
• Understanding data races
• Atomic objects
• A word about Java
• Conclusions
Atomic objects

• Pthreads programs
  – Should not have data races
  – Frequently have intentional data races

• Problem:
  – Lock-based synchronization often perceived as too heavy weight.

• C++11/C11 solution: atomic objects
  – Allow concurrent access
    • Do not participate in data races.
  – By default preserve simple sequentially consistent behavior

20 April 2012
A note on naming

- Roughly similar (synchronization variables):
  - C++11 `atomic<t>`, `atomic_??`
  - C11 `_Atomic(t)`, `atomic_??`
  - Java `volatile` (or `j.u.c.atomic`)

- Related, but profoundly different:
  - C# `volatile`
  - OpenMP 3.1 `atomic`

- Unrelated (at least officially):
  - C & C++ `volatile`

20 April 2012
C++0x atomics

template< T > struct atomic {
   // GREATLY simplified
   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 C11!
Counter with atomic object

atomic<int> x;

void increment() {
    X++;  // not x = x + 1
}

Dekker’s example, version 2

```cpp
atomic<int> x, y;  // initially zero

Thread 1
x = 1;
r1 = y;

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

- No data races.
- **Disallows** \( r_1 = r_2 = 0 \).
- Compiler and hardware do whatever it takes.
Done flag, version 2

```cpp
int x; // initially zero
atomic<bool> done; // initially false
```

```
Thread 1
x = 42;
done = true;
```

```
Thread 2
while (!done) {} 
assert(x == 42);
```

- No data races. Works.
- Compiler and hardware do whatever it takes.
Lazy initialization version 2

atomic<bool> initd; // initially false.
int x;

Thread 1
if (!initd) {
    lock_guard<mutex> _(m);
    x = 42;
    initd = true;
}
read x;

Thread 2
if (!initd) {
    lock_guard<mutex> _(m);
    x = 42;
    initd = true;
}
read x;

No data race.
Problem:

- “Do whatever it takes” (ensuring SC) can be expensive.
  - At least on some current hardware.
  - Though less so on modern x86 hardware.
  - And the cost appears to be decreasing.

“Solution”:

- Allow programmers to explicit relax sequential consistency.
- Programs no longer behave as though threads were simply interleaved.
- Much more complicated & bug-prone.
  - Complexity is hard to localize.
- Sometimes significantly faster on current hardware.
atomic<bool> done;
int x;

Thread 1:
   x = 42;
   done.store(true, memory_order_release);

Thread 2:
   while (!done.load(memory_order_acquire)) {}
   assert (x == 42);

Details not covered here.
Outline

• Overview
• C++11 Threads API (very briefly)
• C++11/C11 Memory model
• Understanding data races
• Atomic objects
• A word about Java
• Conclusions
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.
This is hard!

• **Want**
  – Constrained race semantics for essential security properties.
  – Unconstrained race semantics to support compiler and hardware optimizations.
  – Simplicity.

• **No known good solution.**
Java 2005 “Solution”

See

http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.8
• The good news:
  – Sequential consistency for data-race-free programs.
• The bad news:
  – Data race semantics are broken.
    • Main theorem about compiler optimizations is wrong. (Aspinall&Sevcik 2007)
    • Other issues.
    • Too complicated ...
  – We don’t really know how to fix it.
• Avoid data races!
  – Also in Java.
Conclusions

• C++11 and C11 are (finally!) multithreaded languages.
• Shared variables are (finally!) well-defined.
  – No matter which API you use!
• Atomics make it easier to write data-race-free programs.
• C++ “Catch fire” data-races are a useful compromise:
  – Programs should be data-race-free anyway.
  – We don’t know how to define data race semantics.
  – Unlike Java, we have (mostly) sound shared-variable semantics.
  – But behavior of buggy programs is completely unconstrained.
    • Unhelpful for debugging
    • Like C arrays and bounds checking ...
• Java data races aren’t really defined either.
• Data races are evil!
Questions?

- For more information:
  - Boehm, Adve, You Don't Know Jack About Shared Variables or Memory Models, Communications of the ACM, Feb 2012.

Easily understandable

Mathematically rigorous

C++ specific
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;
}
New compiler restrictions

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

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

x.a = 'z';
```

```c
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.
Some restrictions are a bit more annoying:

- Compiler may not introduce “speculative” stores:

```c
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
```
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!
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

```cpp
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;
```