Memory Model for C++: Status update

Hans-J. Boehm

HP Labs

Hans.Boehm@hp.com

WG21/N1911=J16/05-0171

2005-10-20

With help from Bill Pugh, Doug Lea, Peter Dimov, Alexander Terekhov, ...

Goals of this talk

- Outline where we have been.
- What are the difficulties?
- Tradeoffs for atomic operations
  - & why those are fundamental
- Current status

A note on assumptions

- In spite of N1834, we concentrate on threads.
- I believe these reflect the most common approach to concurrency, though there are others:
  - Message passing (e.g. MPI): Different issues.
  - Partially shared address space:
    - Sometimes useful, partially addressed.
    - Pointers and virtual functions broken.
    - Share many of the same issues.

Approach, from last time: (still a bit tentative)

- "Pthreads-like" memory model.
  - Data race: A store to a memory location concurrent with another load or store to a memory location.
  - Data races have undefined semantics.
  - Otherwise: Sequential consistency.
- Careful and restrictive definition of "data race" and "memory location".
  - Only bit-fields share a "memory location."
  - Data races defined for seq. consistent exec.

Reasons for this approach

- We can get away with it, kind of.
  - No type-safety required.
- Remain consistent with current practice.
- Java-like approach disallows some compiler optimizations:
  - Register "rematerialization".
  - Code hoisting (sometimes).
- Requires memory barriers on object construction to ensure vtable visibility.
- Avoid (?) complex causality treatment.
- Avoid atomicity constraints.

The Problem: Atomic Operations Library.

- Some low level code requires data races for performance.
- Common example: "double-checked locking"

```
if (!x_initialized) {
    lock();
    if (!x_initialized) x = ...;
    x_initialized = true;
    unlock();
}
 ... x ...
```

- Incorrect as is: Data race!

Double-checked locking:
Why it has to be illegal as is.

- Compiler/hardware may reorder

```
if (!x_initialized) {
    lock();  // Not real syntax
    if (!x_initialized) x = ...;
    x_initialized = true;
    unlock();
  }
  ... x ...
```

- E.g., compiler may load x early after discovering that it misses cache.
- Some architectures allow reordering.

The solution: atomic operations

- Loads and stores of `x_initialized` must be done specially:
  - Tell compiler (and programmer) that a race is involved.
  - Ensure atomicity.
  - Specify ordering constraints.
- Use either a special `volatile` variant, or calls to a standard atomic operations library.
  - We are concentrating on the library for now.

Double-checked locking: Correct, with atomic operations

- Use atomic operations (not real syntax):

```
if (!load_acquire(x_initialized)) {
    lock();
    if (!x_initialized) x = ...;
    store_release(x_initialized, true);
    unlock();
}
  ... x ...
```

- Store_release ensures that preceding stores are visible to a load_acquire reading variable in another thread.

A controversial part: Memory ordering constraints:

- Different hardware can cheaply enforce different types of ordering constraints.
  - Argues for many different supported variants:
  - E.g. *order load with respect to later operations "control-dependent" on it*.
- But:
  - These often don't make sense at source level.
  - Sometimes they constrain separate compilation.
  - Synchronization operations that allow reordering complicate semantics.
  - More variety complicates semantics more.

Atomic operation semantics

Variables x, y, and z initially 0

- **Thread 1:**

```
store_unordered(x, 1);
r1 = load_unordered(y);
if (r1 == 0) z = 17;
```

- **Thread 2:**

```
store_unordered(y, 1);
r2 = load_unordered(x);
if (r2 == 0) z = 42;
```

Does this have a data race?

- Simultaneous accesses through atomics don't count.
- No race on z under sequentially consistent interpretation.
- But simultaneous accesses are really possible.
- This must have undefined semantics in order to preserve the compilers optimization ability.

Current approach

- Definition of data race assumes
  - Sequential consistency for ordinary memory accesses.
  - Java-like semantics for atomic operations.
  - (this is technically tricky.)

Causality

- **Problem: This brings back the complexity of Java memory model.**

Initially x = y = 0

Thread 1:

```
store(x, load(y));
z[x] = 17;
```

Thread 2:

```
store(y, load(x));
z[42] = 23;
```

- **Solutions under consideration:**
  - Simply say "no speculation on atomics" (vague)
  - Try for simpler model that overconstrains optimization of atomics.

Issues related to atomics:

- Fine control vs. ease of use?
  - How many ordering constraints?
  - Do we want higher level facilities, like Lawrence Crowl's proposal?
    - In addition to or instead of lower level package?
- Templatized w.r.t. location type?
  - atomic<T> vs atomic_ptr or both?
- Operations parameterized w.r.t. ordering?
  - load_acquire vs. load<acquire> vs. load(acquire, ...)
- Emulated operations & feature tests.
  - Don't have compare-and-swap everywhere.

Current status

- Web page at
  `http://www.hpl.hp.com/personal/Hans_Boehm/c++mm`
- Includes (still informal) proposal
- Needs further scrutiny
- Very preliminary atomic operations library interface
  - Want more C compatibility.
- Would like opinions on:
  - Atomics interface.
  - Required precision of atomics memory model.