Memory Model for Multithreaded C++

N1738=04-0178

Andrei Alexandrescu
Hans Boehm
Kevlin Henney
Doug Lea
Bill Pugh
Maged Michael

Agenda

- Myth and reality: is threading a library issue?
- Introduction to memory model
- Conclusions

Threads: library?

- Myth: Threads can be implemented as a C++ library without changing the language
- Fact: Threads affect the very core of code generation and execution
  - Endless battle between optimizations and correct multithreaded behavior
- Fact: Threads can be implemented without changing the ***syntax*** of the language
  - It's the ***semantics*** that need changed

Current execution model

- 1.9/1: "… conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below."
- 1.9/6: "The observable behavior of the abstract machine is its sequence of reads and writes to `volatile` data and calls to library I/O functions."
  - Implicit single-threading
  - No relationship between operations on `volatile` and non- `volatile` data
  - No "global effects" possible

Locks

- Classic lock semantics cannot be defined within the current language:

Mutex m; /* ... */

{ Lock lock(m); /* access data */ }

- Nonvolatile reads and writes can be moved across the lock (cf. language definition)
- Need to express: All data (volatile and not) operations inside the locked region must start after the Lock's ctor and be committed by the Lock's dtor
  - e.g., no register promotions!

Reality Check

- The language can’t express such semantics
- Overly pessimistic (disables many valid opt’s)
- No help for user-space locking
- Such an observation doesn’t help the plethora of widely used lock-less mechanisms
- No help for lock-free and wait-free techniques either

Example

```
const char* sym; double price;
if (sym == 0) { price = 27.9; sym = “msft”; } // writer
if (sym != 0) { p = price; s = sym; sym = 0; } // reader
```

- Writers write prices and set symbols
- Readers read them and reset the symbols
- Simple synchronization device
  - It should be allowed
  - Relies on memory ordering: what if price is updated after the symbol?

Down to the core

- Consider:

a = 5;

b = 6;

- The sequence in which they actually are updated is up to the implementation
- Inter-thread communication routinely depends on proper sequencing of such operations
- This is *not* a theoretical issue

Make everything volatile?

- Possible approach: make all data that is ever manipulated by multiple threads volatile
- Manipulated even though not shared!
- Severe pessimization for the sake of a few hot spots
- A volatile write costs ~50% of an uncontended lock operation
- Note: pthreads is defined such that it never relies on volatile because of its insufficiently strong semantics

Lock-Free programming

- CAS primitive (belongs to std):

```
bool cas(int* p, int expected, int newval) {
 if (*p != expected) return false;
 *p = newval;
 return true;
}
```

- It's been proven that any shared data structure can be implemented with CAS alone
- A flurry of research and development

Lock-Free advantages

- Fast (up to 4 times faster than mutexes)
- Readers don’t get in each other’s way
- Graceful degradation under contention
- Single-variable lock-free operations much faster than lock-based
- Async signal safety
- Immunity to priority inversion
- Tolerance to thread death

Lock-free disadvantages

- Can’t control priorities => can increase contention gratuitously
- Hard to write
- Complex data structures are easier to implement with locks
  - Use locks for 98% of your code
  - Use 2% CAS to increase performance by 98%

- Conclusion: we need both

Approach

- The J word:
  - Java defines a mathematical memory model
    - Fixes bugs in its old informal spec
  - Development took years
    - Heavily reviewed and scrutinized
  - Most of it is language-independent and can be reused for C++
- Shorten development time dramatically

Atomicity

- Certain operations on primitive data must be guaranteed to be atomic
- Still leave leeway to implementations
- Possibly: define **int_atomic_t** (at least N bits integral type)
- (Non-member) pointer operations should be atomic
- Floating-point operations needn't be atomic

Memory modeling

- “Happens-before” relation **–hb>**
  - Partial ordering of memory operations
- **Program order:** classic “as-if” for one thread
- **Monitor:** Unlocking –hb> Locking
- **Volatile:** Write –hb> Read
- **Thread start:** start() –hb> thread actions
- **Thread termination:** thread actions –hb> join()

Looking forward

- Once the memory model is complete, semantics of library primitives can be defined on top of it
- Development of memory model separate from development of libraries

Conclusions

- Language changes necessary
  - No syntax changes needed
  - Subtle changes in semantics
  - Backwards compatible
- Pure library additions to come
- Issue: shall we reuse/redeem volatile or not?