Memory Model for Multithreaded C++

Hans-J. Boehm
HP Labs

Other participants:
Andrei Alexandrescu, Peter Buhr, Kevlin Henney, Ben Hutchings, Doug Lea, Maged Michael, Bill Pugh

Motivation

- Multithreaded programming is critically important:
  - For dealing with multiple event streams.
    - The old reason.
  - Because everything will be a multiprocessor,
    - and there is often no other way to utilize it.

Background

- The status quo:
  - C++ standard doesn't mention threads.
  - Threads added via libraries (e.g. pthreads).
  - Compiler mostly unaware of threads.
  - Synchronization calls treated as opaque.
  - *Almost* works if there are no unprotected concurrent accesses (*data races*).
    - Compiler transformations safe for single thread.
    - Safe in locked regions.
    - Safe without shared variables.

Kinds of failures

- Compiler introduces *data races*, i.e. concurrent writes.
  - Overwriting of adjacent fields, array elements or variables.
  - Speculative memory references.
    - Speculative register promotion in loops.
      - Common optimization.
      - Completely unsafe with threads.
      - Rare, but unpredictable failures.
    - Optimizations that assume termination.

Example 1

struct {char a; char b; char c; char d;} x;
    -- initialized to zeroes

Thread 1	Thread 2
++x.a;	++x.d;
++x.b;	
++x.c;	
// x = x + 0x10101;	

x.d can still be zero!

Example 1, contd.

- Language standards allow this implementation.
  - Deal only with single-threaded semantics.
  - In the case of bit-fields, it's unavoidable.
  - Pthreads allows it even for independent global variables.
- Thread (pthread) standard intentionally doesn't address issue.
  - Leave transformations between locking primitives up to sequential compiler.
- Result: Unexpected concurrent writes (races).
  - No way to protect against them.
  - Adjacent memory overwrites are not the only problem.

Example 2

- **Before:**

```
for (...) {
  if (mt) lock();
  use x;
  if (mt) unlock();
}
```

- **After:**

```
r = x;
for (...) {
  if (mt) {
    x = r;
    lock();
    r = x;
  }
  use r;
  if (mt) ...
}
x = r;
```

Example 3

- **Before:**

```
for (x = y;
     x != 0;
     x = x->next)
   c++;
z = 1;
```

- **After:**

```
z = 1;
for (x = y;
     x != 0;
     x = x->next)
   c++;
```

Uncommon on this form. But common compiler analyses assume this is legal.

The solution

- Language standard has to either
  - Define precisely what language constructs mean in the presence of threads, or
  - Define precisely when races may occur, and disallow them.
  - Question: Which one?

The Java solution

- Java supports "sandboxed" execution of untrusted code.
- Cannot leave semantics of data races undefined.
  - Cannot prevent data races in malicious code.
  - Secure code must guard against them.
- Even type-safety requires a lot of this.
- Not currently an issue for C++ (?)

Currently preferred solution

- Define precisely when data races occur.
  - A C++ program contains a data race if a naive sequentially consistent execution contains a data race.
  - A store to a bit field is treated as a store to all adjacent bit fields.
  - No other implicit stores are allowed.
  - Concurrent modifications using "special" atomic primitives are not a race.
  - Race-free programs have sequentially consistent semantics; o.w. undefined.

Some consequences

- Multiprocessor architectures not supporting efficient atomic byte stores will perform badly. (There aren't any?)
- Uniprocessors may need to use restartable atomic sequences.
- Speculative register promotion across function calls is disallowed.
- Combination of field writes is largely disallowed.
- Movement across potentially nonterminating loops is mostly disallowed.

Example 4: Wrong, but common

Double-checked locking:

```
bool is_initialized;
if (!is_initialized) {
   lock();
   if (!is_initialized) {
      <initialize x>;
      is_initialized = true;
   }
   unlock(); }
<use x>;
```

Secondary issue: `volatile`

- Should `volatile` references qualify as "special atomic" primitives?
- I.e. should we allow races involving only `volatile` accesses?
- Pro:
  - Makes `volatile` useful for threads.
  - Makes it easier to fix existing code
    - "double checked locking" pattern.
  - Gives real meaning to volatile.
    - Currently many variations, even on IA64.
  - Consistency with Java.

Volatile, contd.

- Cons:
  - Assignments to a volatile often more expensive than strictly needed.
    - DCL initialization path.
  - Too strong for some existing applications.
    - At least those with explicit memory barriers.

Secondary issue: Function scope statics

- Current problem:
  - `int f() { static foo x(17); ... }`
  - Introduces hidden "is initialized" flag.
  - Implicit potential race on flag.
- Options:
  - Compiler adds synchronization.
    - Unexpected overhead.
    - Usually useless? Details messy.
  - Leave synchronization to programmer.
    - Subtle correctness problems.
  - Deprecate? Alternatives?

Other issues

- We need an atomic operations library.
  - Not all architectures support e.g. CAS.
    - Emulation or feature tests? Both?
- Asynchronous signal support?
- Can/should we standardize thread library itself.
  - Compromise:
    - Standardize only a high level facility.
      - e.g. futures.