The C++11 Memory Model

CDP

Based on “C++ Concurrency In Action” by Anthony Williams,
The C++11 Memory Model and GCC Wiki
and Herb Sutter’s atomic<> Weapons talk.

Created by Eran Gilad
Reminder: what is a memory model?

- The guarantees provided by the runtime environment to a multithreaded program, regarding the order of memory operations
- Each level of the environment might have a different memory model
  - CPU, virtual machine, compiler
- The correctness of parallel algorithms depends on the memory model
Why C++ needs a memory model

• Isn’t the CPU memory model enough?
• Until now, different code was required for every compiler, OS and CPU combination
• Threads are now part of the language (C++11)
  • Their behavior must be fully defined
  • The same code should run the same on different environments regardless of the CPU (Intel, AMD, ARM, ...) or the OS
• Having a standard guarantees portability and simplifies the programmer’s work
C++ in the past and now

• Given the following data race:
  • Thread 1: x = 10; y = 20;
  • Thread 2: if (y == 20) assert(x == 10);

• Pre-2011 C++
  • What’s a “Thread”?
  • Namely – behavior is unspecified (not even undefined)

• C++11
  • Undefined Behavior (don’t try the above code at home)
  • But now, C++11 introduces a new set of tools to get it right
Why Undefined Behavior matters

- Can the compiler convert the switch to a quick jump table, based on \( x \)'s value?

- Optimizations that are safe on sequential programs might not be safe on parallel ones.

- So, races are forbidden!

```cpp
if (x >= 0 && x < 3) {
    switch (x) {
        case 0: do0(); break;
        case 1: do1(); break;
        case 2: do2(); break;
    }
}
```

At this point, on some other thread:

\[ x = 8; \]

Switch jumps to unknown location. Your monitor catches fire.
Dangerous optimizations

• Optimizations are crucial for increasing performance, but might be dangerous

• Compiler optimizations:
  • Reordering to hide load latencies
  • Using registers to avoid repeating loads and stores

• CPU optimizations:
  • Loose cache coherence protocols
  • Out Of Order Execution

• Thumb rule: a thread must appear to execute serially to other threads
  • Optimizations must not be visible to other threads
Memory layout definitions

• Every variable of a scalar (simple) type occupies one memory location
• Bit fields are different
• One memory location must not be affected by writes to an adjacent memory location!

```c
struct s {
    char c[4];
    int i: 3, j: 4;
    struct in {
        double d;
    } id;
};
```

Can’t read and write all 4 bytes when changing just one
Same memory location
Considered one memory location even if hardware has no 64-bit atomic ops
# Threads in C++11 (std::mutex)

```cpp
#include <iostream>
#include <thread>

void func(mutex * m, int * x) {
    m->lock();
    std::cout << "Inside thread " << ++(*x) << std::endl;
    m->unlock();
}

int main() {
    mutex m; int x = 0;
    std::thread th(func, &m, &x); // or std::ref(m), std::ref(x)
    th.join();
    std::cout << "Outside thread " << x << std::endl;
    return 0;
}
```
Threads in C++11 (std::atomic)

• std::atomic provides both atomicity and ordering guarantees
• Default order is sequential consistency
  • Other orders are also possible
• The language offers specializations for basic types
  • Basic types: bool, numbers, pointers
  • Custom wrappers can be created by the programmer (i.e. template)
• A mutex is sometimes required to ensure atomic reads and writes
  • Most architectures provide atomic primitives for up to a dword
  • In case the variable is too large, a mutex must be used
Threads in C++11 (std::atomic definition)

template<typename T> struct atomic
{
    void store(T val, memory_order m = memory_order_seq_cst);
    T load(memory_order m = memory_order_seq_cst);
    // etc...
};

// and on some specializations of atomic:
T operator++();
T operator+=(T val);
bool compare_exchange_strong(T& expected, T desired,
    memory_order m = memory_order_seq_cst);
// etc...
Order Definitions: Sequenced Before

• The order imposed by the appearance of (most) statements in the code, for a single thread

Thread 1
read x, 1

write y, 2
Order Definitions:
Synchronized with

- The order imposed by an atomic read of a value that has been atomically written by some other thread

**Thread 1**
write y, 2

**Thread 2**
read y, 2
Order Definitions:
Inter-thread happens-before

- A combination of sequenced before and synchronized with

**Thread 1**
read x, 1
write y, 2

**Thread 2**
read y, 2
write z, 3
Order Definitions: Happens Before

- An event A **happens-before** an event B if:
  - A inter-thread happens-before B or
  - A is sequenced-before B

**Thread 1**

1. read x, 1
2. write y, 2

**Thread 2**

3. read y, 2
4. write z, 3
Using atomics
Memory Order: Sequential Consistency

• Strongest ordering, default *memory order*
• Atomic writes are globally ordered
• Compiler and processor optimizations are limited
  • No action can be reordered across an atomic action
  • Non-atomic actions can be safely reordered as long as the reorder doesn’t cross an atomic command
• Atomic writes flush non-atomic writes that are *sequenced-before* them
  • They are externally visible after the atomic action
Using atomics
Memory Order: Sequential Consistency (2)

atomic<int> x;
int y; // not atomic

void thread1() {
    y = 1;
x.store(2);
}

void thread2() {
    if (x.load() == 2) {
        assert(y == 1);
    }
}

The assert is guaranteed not to fail
This was the part you have to know in order to write correct code

Now, if you like to play with sharp tools...
Using atomics

Memory Order: Acquire Release

• Synchronized-with relation exists only between the releasing thread and the acquiring thread
• Other threads might see updates in a different order
• All writes before a release are visible after an acquire
  • Even if they were relaxed or non-atomic
• Similar to release consistency memory model
Using atomics
Memory Order: Acquire Release (2)

```c
#define rls memory_order_release /* save slide space... */
#define acq memory_order_acquire /* save slide space... */
atomic<int> x, y;

void thread1() {
  y.store(20, rls);
}

void thread2() {
  x.store(10, rls);
}

void thread3() {
  assert(y.load(acq) == 20 && x.load(acq) == 0);
}

void thread4() {
  assert(y.load(acq) == 0 && x.load(acq) == 10);
}
```

Both asserts might succeed
No order between writes to x and y
Using atomics

Memory Order: Relaxed Consistency

• Happens-before now only exists between actions to the same variables
• Similar to cache coherence
  • But can be used on systems with incoherent caches
• Imposes fewer limitations on the compiler
  • Atomic actions can now be reordered if they access different memory locations
• Less synchronization is required on the hardware side as well
Using atomics
Memory Order: Relaxed Consistency (2)

```cpp
#define rlx memory_order_relaxed /* save slide space... */

atomic<int> x, y;

void thread1() {
    y.store(20, rlx);  // W y, 20
    x.store(10, rlx);  // W x, 10
}

void thread2() {
    if (x.load(rlx) == 10) {
        assert(y.load(rlx) == 20);  // R y, 10
        y.store(10, rlx);  // W y, 10
    }
}

void thread3() {
    if (y.load(rlx) == 10) {
        assert(x.load(rlx) == 10);  // R x, 0
    }
}

Both asserts might fail
No order between operations on x and y
```
Using atomics
Memory Order: Relaxed Consistency (3)

• Let’s look at the trace of a run in which the assert failed

T1: \textbf{W y, 20}; \textbf{W x, 10};

T2: \textbf{R x, 10}; \textbf{R y, 20}; \textbf{W y, 10};

T3: \textbf{R y, 10}; \textbf{R x, 0};
Using atomics
Memory Order: Relaxed Consistency (4)

• Let’s look at the trace of a run in which the assert failed

T1: W y, 20; W x, 10;

T2: R x, 10; R y, 20; W y, 10;

T3: R y, 10; R x, 0;
Using atomics
Memory Order: Relaxed Consistency (5)

• Let’s look at the trace of a run in which the assert failed

T1: W y, 20; W x, 10;
T2: R x, 10; R y, 20; W y, 10;
T3: R y, 10; R x, 0;
volatile in multi-threaded C++

• Predates C++ multi-threading
• Serves different purpose
  • Intended for reading from memory that was written by a device
  • Does not guarantee atomic
  • Unspecified memory order
• However Java (and C#) volatile is very similar to C++ atomic
Summary and recommendations (1)

• Lock-free code is hard to write
  • Unless speed is crucial, better use mutexes

• Sequential consistency is the default
  • It is also the memory model for function that does not have memory-model as an input parameter

• Use the default sequential consistency and avoid data races. Really.
  • Acquire-release is hard
  • Relaxed is much harder
  • Mixing orders is insane (but allowed)
Summary and recommendations (2)

• The memory model can be passed as run-time parameter
  • But it is better to use it as a compilation constant to allow optimizations
• We only covered main orders
• C++11’s basic features are fully supported by all common compilers