C++ Memory Model

Valentin Ziegler
Fabio Fracassi
Meeting C++ Berlin, December 6th, 2014
The machine does not execute the code you wrote...
How your code is executed

<table>
<thead>
<tr>
<th>Code</th>
<th>Compiler</th>
<th>CPU</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo(int n, int m) {</td>
<td>mov esi,1</td>
<td>Pattern history table</td>
<td>Memory</td>
</tr>
<tr>
<td>int x=1;</td>
<td>test ecx,ecx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for (</td>
<td>jle 011F1301</td>
<td>000</td>
<td>004</td>
</tr>
<tr>
<td>int i=0;</td>
<td>lea ebx, [ebx]</td>
<td>001</td>
<td>005</td>
</tr>
<tr>
<td>i&lt;n;</td>
<td>mov eax,1</td>
<td>002</td>
<td>006</td>
</tr>
<tr>
<td>++i</td>
<td>sub eax,edx</td>
<td>003</td>
<td>007</td>
</tr>
<tr>
<td>}{</td>
<td>imul esi,eax</td>
<td>004</td>
<td>008</td>
</tr>
<tr>
<td>if (m&lt;0) {</td>
<td>dec ecx</td>
<td>005</td>
<td>009</td>
</tr>
<tr>
<td>x -= x*m;</td>
<td>jne 011F12F0</td>
<td>006</td>
<td>00A</td>
</tr>
<tr>
<td>}</td>
<td>...</td>
<td>007</td>
<td>00B</td>
</tr>
<tr>
<td>return x;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Register allocation
- Loop unswitching
- Branch prediction
- Out of order exec.
- Prefetching
- Buffering

Optimization
How your code is executed

Single thread execution model (C++03):

- Program will behave *as-if* it was yours: Result is the same as if operations were executed in the order specified by the program

- We can not observe optimizations performed by the system
Two threads of execution?

```c
bool f1 = false; bool f2 = false;
```

Thread #1

```c
f1 = true;
if (!f2) {
    // critical section
}
...
```

Thread #2

```c
f2 = true;
if (!f1) {
    // critical section
}
...
```

- Optimizations become observable
- Optimizations may break “naive” concurrent algorithms
Memory Model

- Describes the interactions of threads through memory and their shared use of data.

- Tells us if our program has well defined behavior.

- Constrains code generation for compiler
The C++ Memory model

C++ Memory Model Basics
Data Races, Sequential Consistency, Synchronization

Meddling with Memory Order
Relaxed Atomic Operations, and subtle Consequences
Data Race

**memory location** [intro.memory(1.7)/3]
- an object of scalar type or a maximal sequence of adjacent non-zero width bit-fields

**conflicting action** [intro.multithread(1.10)/4]
- two (or more) actions that access the same *memory location* and at least one of them is a write

**data race** [intro.multithread(1.10)/21]
- two *conflicting actions* in different threads and neither *happens before* the other.
Data Race

Memory location

An object of scalar type or a maximal sequence of adjacent non-zero width bit fields

Conflicting action

Two (or more) actions that access the same memory location and at least one of them is a write

Data race

Two conflicting actions in different threads and neither happens before the other

```c
int i;
char c;
int a:5,
    b:7;
X* p;
```
Data Race

memory location

an object of scalar type or a maximal sequence of adjacent non-zero width bit fields

conflicting action

two (or more) actions that access the same memory location and at least one of them is a write

data race

two conflicting actions in different threads and neither happens before the other.

<table>
<thead>
<tr>
<th></th>
<th>Thread #1</th>
<th>Thread #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i;</td>
<td>a = 23;</td>
<td>c = ‘@’;</td>
</tr>
<tr>
<td>char c;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int a:5,</td>
<td>X* x = p;</td>
<td>X* x = p;</td>
</tr>
<tr>
<td>b:7;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X* p;</td>
<td>int n = i</td>
<td>i = 42;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thread #1

X* x = p;

Thread #2

X* x = p;

int n = i

i = 42;
Data Race

Data race == undefined behavior!
Sequential Consistency

**sequential consistency** [Leslie Lamport, 1979]

the result of any execution is the same *as-if*

1. the operations of all threads are executed in some sequential order

2. the operations of each thread appear in this sequence in the order specified by their program
Sequential Consistency

Sequential consistency [Leslie Lamport, 1979]

The result of any execution is the same as if the operations of all threads are executed in some sequential order, and the operations of each thread appear in this sequence in the order specified by their program.
The C++ memory model

Here is the deal:

- We take care our program does not contain data races
- The system guarantees sequentially consistent execution

sequential consistency for data-race-free programs

SC-DRF
synchronize (the easy way)
Locks

Mutually exclusive execution of critical code blocks

```cpp
std::mutex mtx;
{
    mtx.lock();
    // access shared data here
    mtx.unlock();
}
```

Mutex provides inter-thread synchronization:

- `unlock()` synchronizes with calls to `lock()` on the same mutex object.
Locks

Mutually exclusive execution of critical code blocks

```cpp
std::mutex mtx;
{
  mtx.lock();
  // access shared data here
  mtx.unlock();
}
```

Mutex provides inter-thread **synchronization**: 

`unlock()` synchronizes with calls to `lock()` on the same mutex object.
Locks

Mutually exclusive execution of critical code blocks

```cpp
std::mutex mtx;
{
    std::lock_guard<std::mutex> lg(mtx);
    // access shared data here
    // lg destructor releases mtx
}
```

Mutex provides inter-thread synchronization:

- `unlock()` synchronizes with calls to `lock()` on the same mutex object.
Synchronize using Locks

```cpp
std::mutex mtx; bool bDataReady=false;
```

Thread #1

```cpp
{   mtx.lock();
    PrepareData();
    bDataReady=true;
    mtx.unlock();
}
```

Thread #2

```cpp
{   mtx.lock();
    if (bDataReady) {
        ConsumeData();
    }
    mtx.unlock();
}
```

“Simplistic view” on locking:
Critical code cannot run in both threads “simultaneously”
What about synchronization?

- The C++ standard identifies certain operations to be synchronization operations.

![](image)

- If \( A() \) synchronizes with \( B() \), then \( X() \) happens before \( Y() \).
Locks

**Mutual exclusive execution of critical code blocks**

```cpp
std::mutex mtx;
{
    mtx.lock();
    // access shared data here
    mtx.unlock();
}
```

**Mutex provides inter-thread synchronization:**

- `unlock()` synchronizes with calls to `lock()` on the **same** mutex object.
Synchronize using Locks

std::mutex mtx; bool bDataReady=false;

Thread #1
{
    mtx.lock();
    PrepareData();
    bDataReady=true;
    mtx.unlock();
}

Thread #2
{
    mtx.lock();
    if (bDataReady) {
        ConsumeData();
    }
    mtx.unlock();
    
    mtx.unlock() synchronizes with mtx.lock()
    PrepareData() happens before ConsumeData()

std::mutex mtx;  bool bDataReady=false;

Thread #1

PrepareData(); // once
{
    mtx.lock();
    bDataReady=true;
    mtx.unlock();
}

Proper synchronization, if PrepareData() is never executed again.

Thread #2

bool b;
{
    mtx.lock();
    b=bDataReady;
    mtx.unlock();
}
if (b) ConsumeData();
Clever?

std::mutex mtx;  bool bDataReady=false;

Thread #1

PrepareData();  // once
{
    mtx.lock();
    bDataReady=true;
    mtx.unlock();
}

Thread #2

if (!mtx.try_lock()){
// l33t optimization:
// thread 1 should
// be done with
// PrepareData :-)
    ConsumeData();
}
```c
std::atomic<>
```

- “Data race free” variable, e.g., `std::atomic<int>`
- (by default) provides inter-thread **synchronization**:
  ```c
  a store synchronizes with operations that load the stored value.
  ```

- (by default) sequential consistency
- Needs hardware support
  (not all platforms provide lock-free atomics)
std::atomic<>

In C++, this is spelled `std::atomic`, not `volatile`!

- a store synchronizes with operations that load the stored value.

- (by default) sequential consistency
- Needs hardware support
  (not all platforms provide lock-free atomics)
Synchronize using atomics

Thread #1

PrepareData(); // once
bDataReady.store(true);

Thread #2

Proper synchronization,
if PrepareData() is never executed again.

if(bDataReady.load()){
    ConsumeData()
}
Excursion: lock-free programming

template<
typename T> class lock_free_list {
    struct node{
        T data; node* next;
    };
    std::atomic<
node*> head;
public:
    void push(T const& data) {
        node* const newNode = new node(data);
        newNode->next = head.load();
        while(!head.compare_exchange_weak(newNode->next, newNode))
            ;
    }
};

Desired

Expected

Target
Are we there yet?

Avoid data races and you will be fine
  • Synchronize correctly
  • Implement lock-free data structures as described in your favorite computer science text book

The C++ memory model guarantees sequential consistency
  • as does the memory model of Java and C#
The C++ Memory model

C++ Memory Model Basics
Data Races, Sequential Consistency, Synchronization

Meddling with Memory Order
Relaxed Atomic Operations, and subtle Consequences
std::atomic<>

- (by default) provides inter-thread synchronization:
  
  a store synchronizes with operations that load the stored value.

- (by default) sequential consistency
Memory Order

Why only have one memory model when we can have a mix of 3 (and a half)?

<table>
<thead>
<tr>
<th>memory model</th>
<th>memory_order_*</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequentially consistent (SC)</td>
<td>seq_cst</td>
</tr>
<tr>
<td>acquire-release</td>
<td>acquire release</td>
</tr>
<tr>
<td>consume-release</td>
<td>consume release</td>
</tr>
<tr>
<td>relaxed</td>
<td>relaxed</td>
</tr>
</tbody>
</table>
Relaxed memory ordering

sequentially consistent (SC)
acquire-release
consume-release
relaxed

seq_cst
acquire release acq_rel
consume release acq_rel
relaxed
What is relaxed memory order

- Each memory location has a total **modification order** (however, this order cannot be observed directly)
- Memory operations performed by the **same thread** on the **same memory location** are not reordered with respect to the modification order.

```cpp
std::atomic<int> x;

x.store(42, memory_order_relaxed);
ox.load(memory_order_relaxed);

x.compare_exchange_weak(n, 42, memory_order_relaxed);
```
Relaxed load and store

Thread #1

x.store(1);
x.store(2);
x.load();
x.load();
x.store(4);

Thread #2

x.load();
x.store(3);

Modification Order
Relaxed load and store

Thread #1

Modification Order

x.store(1);
x.store(2);
x.load();
x.load();
x.store(4);

Thread #2

x.store(3);
x.load();
Relaxed load and store

Thread #1

x.store(1);
x.store(2);
x.load();
x.load();
x.store(4);

Thread #2

x.store(3);

x.load();

Modification Order
Relaxed load and store

Thread #1

- x.store(1);
- x.store(2);
- x.load();
- x.load();
- x.store(4);

Thread #2

- x.store(3);
- x.load();

Modification Order
Relaxed load and store

Modification Order

Thread #1

```
x.store(1);
x.store(2);
x.load();
x.load();
x.store(4);
```

Thread #2

```
x.store(3);
```

Thread #1's program:
- Line 0: `x.store(1)`
- Line 1: `x.store(2)`
- Line 2: `x.load()`
- Line 3: `x.load()`
- Line 4: `x.store(4)`
Relaxed load and store

Thread #1

x.store(1);
x.store(2);
x.load();
x.load();
x.store(4);

Thread #2

x.store(3);

Modification Order

0 1 2 3 4
Relaxed load and store

Thread #1

x.store(1);
x.store(2);
x.load();
x.load();
x.store(4);

Thread #2

x.store(3);
x.load();

Modification Order
Relaxed load and store

Safe?

```cpp
atomic<bool> f=false;
atomic<bool> g=false;
```

Thread #1:

```cpp
f.store(true, memory_order_relaxed);
g.store(true, memory_order_relaxed);
```

Thread #2:

```cpp
while(!g.load(memory_order_relaxed));
assert(f.load(memory_order_relaxed));
```
Relaxed load and store

Safe?

atomic<bool> f=false;
atomic<bool> g=false;

Thread #1:

f.store(true, memory_order_relaxed);
g.store(true, memory_order_relaxed);

Thread #2:

while(!g.load());
f.load();

while(!g.load(memory_order_relaxed));
assert(f.load(memory_order_relaxed));

assert(f.load(memory_order_relaxed));
Relaxed read-modify-write

lock-free ++x

int oldVal=x.load();
Sets oldVal to 0

x.compare_exchange_weak(
  oldVal,  // expected
  oldVal+1,  // desired
  memory_order_relaxed);
Returns false
Sets oldVal to 1

x.compare_exchange_weak(
  oldVal,  // expected
  oldVal+1,  // desired
  memory_order_relaxed);
Safe?  Yes.  Progress?

atomic<int> c = 0

Worker thread #1,#2, ...:

for (int i=0; i<100; ++i) {
    ...
    c.fetch_add(1, memory_order_relaxed);
}

Main thread:

start_n_threads();
join_n_threads();
assert(100*n == c);✔

Built-in for

int oldVal=c.load();
while(!c.compare_exchange_weak(
    oldVal,    // expected
    oldVal+1  // desired
    memory_order_relaxed
);
atomic<int> c = 0

Worker thread #1,#2, ...:

for (int i=0; i<100; ++i) {
  ...
  c.fetch_add(1, memory_order_relaxed);
}

Observing thread ("progress bar"):

for (int old_c = 0;;) {
  int c_now = c.load(memory_order_relaxed);
  assert(old_c <= c_now);
  old_c = c_now;
}

Main thread:

start_n_threads();
join_n_threads();
assert(100*n == c);
The acquire/release model

- sequentially consistent (SC)
- acquire-release
- relaxed
- consume-release
- seq_cst
- acquire
  release
- acq_rel
- consume
  release
- acq_rel
- relaxed
What does acquire/release mean

x.store(42, memory_order_release);
x.load(memory_order_acquire);

• a **store-release** operation **synchronizes** with all **load-acquire** operations reading the stored value.

• All Operations in the releasing thread **preceding** the store-release **happen-before** all operations **following** the load-acquire in the acquiring thread.
Store-release and load-acquire

```c
int n;  // (non-atomic 😊)
n = 23;
x.store(1);
while (!x.load());
int y = n;
assert(y == 23);
```
A note of caution!

“Synchronizes with” relation:
• Refers to operations at runtime.
• NOT about statements in the source code!

```
int n;
(none-atomic 😊)
```

```
Thread #1
n=23;
x.store(1);
```

```
x
0
1
```

```
Thread #2
y=n;
ex.load();
```

No synchronization. Data Race!
Is the answer 42? Yes.

```cpp
atomic<bool> f=false;
atomic<bool> g=false;
int n;

Thread #1:
```
n = 42;
f.store(true, memory_order_release);
```

Thread #2:
```
while(!f.load(memory_order_acquire));
g.store(true, memory_order_release);
```

Thread #3:
```
while(!g.load(memory_order_acquire));
assert(42 == n); ✓
```

The consume/release model

- sequentially consistent (SC)
- acquire-release
- relaxed
- consume-release
- seq_cst
- acquire
- release
- acq_rel
- consume
- release
- acq_rel
- relaxed
What does consume mean

```c
x.store(42, memory_order_release);
x.load(memory_order_consume);
```

• “Light version” of acquire/release

• All Operations in the releasing thread preceding the store-release happen-before an operation X in the consuming thread if X depends on the value loaded.
Who has the answer? \( x \rightarrow i \)

```
struct X { int i; }  
int n; 
std::atomic<X*> px;
```

Thread #1:
```
n = 42; 
auto x = new X; 
x->i = 42; 
px.store(x, memory_order_release);
```

Thread #2:
```
X* x; 
while(!x=px.load(memory_order_consume)); 
assert(42 == x->i);  
assert(42 == n);  

Data Race!
```

Typical examples for “X depends on the value loaded”

- X dereferences a pointer that has been loaded
- X is accessing array at index which has been loaded
Acquire/release provides very strong guarantees. Do we still need more? Who asked for sequential consistency?
Dekker’s algorithm revisited

```cpp
atomic<bool> f1=false;
atomic<bool> f2=false;

Thread #1:

f1.store(true, memory_order_release);
if (!f2.load(memory_order_acquire)) {
   // critical section
}

Thread #2:

f2.store(true, memory_order_release);
if (!f1.load(memory_order_acquire)) {
   // critical section
}
```
Dekker’s algorithm revisited

Thread #1

f1=true;

if (!f2.load()) {
  // critical
  // section
}

Thread #2

f2=true;

if (!f1.load()) {
  // critical
  // section
}

Oh noes!
Back to sanity sequential consistency

sequentially consistent (SC)  seq_cst

acquire-release  acquire release acq_rel
consume-release  consume release acq_rel
relaxed  relaxed
Dekker’s algorithm done right.

```cpp
atomic<bool> f1=false;
atomic<bool> f2=false;

Thread #1:

f1.store(true, memory_order_seq_cst);
if (!f2.load(memory_order_seq_cst)) {
    // critical section
}

Thread #2:

f2.store(true, memory_order_seq_cst);
if (!f1.load(memory_order_seq_cst)) {
    // critical section
}
```
Dekker’s algorithm done right.

- Global, total order of `load` and `store` operations
- At any given time, each memory location has only one value*

* assuming there are no data races
Use-cases for non-SC atomics

• target platform is ARM (<v8) or PowerPC

• operation counters

• some reference counters
  • but then you may use `std::shared_ptr`

• lazy initialization
  • but for this C++ also brings `std::call_once`

PROFILE FIRST before meddling with memory_order!
Wrap up

• Do not write Data Races!

• The C++ Memory Model gives reasonable guarantees to implement correct, yet performant algorithms.

• It allows us to deviate from sequential consistency if we need to.
hr@think-cell.com
Bibliography

• C++ Concurrency in Action – Anthony Williams – 2012
• Atomic Weapons – Herb Sutter – 2012
• ISO C++ Working Draft N3337 – 2012
• How to make a Multiprocessor Computer that correctly executes Multiprocess Programs – Leslie Lamport – 1979
std::atomic<> on x86/x64

<table>
<thead>
<tr>
<th>Load</th>
<th>Store</th>
<th>Compare_exchange</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory_order_seq_cst</td>
<td>MOV prevent compiler optimizations</td>
<td>(LOCK) XCHG prevent compiler optimizations</td>
</tr>
<tr>
<td>memory_order_acquire memory_order_release memory_order_acq_rel</td>
<td>MOV prevent compiler optimizations</td>
<td>MOV prevent compiler optimizations</td>
</tr>
<tr>
<td>memory_order_relaxed</td>
<td>MOV</td>
<td>MOV</td>
</tr>
</tbody>
</table>

[http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html]
std::atomic<> on ARMv7

<table>
<thead>
<tr>
<th>Memory Order</th>
<th>Load</th>
<th>Store</th>
<th>Compare Exchange</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory_order_seq_cst</td>
<td>ldr ; dmb</td>
<td>dmb ; str</td>
<td>dmb ; LOOP ; isb</td>
</tr>
<tr>
<td></td>
<td>prevent compiler optimizations</td>
<td>prevent compiler optimizations</td>
<td>prevent compiler optimizations</td>
</tr>
<tr>
<td>memory_order_acquire</td>
<td>ldr ; dmb</td>
<td>dmb ; str</td>
<td>(dmb;) LOOP (;isb)</td>
</tr>
<tr>
<td></td>
<td>prevent compiler optimizations</td>
<td>prevent compiler optimizations</td>
<td>prevent compiler optimizations</td>
</tr>
<tr>
<td>memory_order_release</td>
<td>ldr ; dmb</td>
<td>dmb ; str</td>
<td>LOOP</td>
</tr>
<tr>
<td></td>
<td>prevent compiler optimizations</td>
<td>prevent compiler optimizations</td>
<td></td>
</tr>
<tr>
<td>memory_order_acq_rel</td>
<td>ldr</td>
<td>str</td>
<td>LOOP</td>
</tr>
</tbody>
</table>

Subroutine LOOP :=

_loop:
   ldrex roldval, [rptr];
   mov rres, 0; teq roldval, rold;
   strexeq rres, rnewval, [rptr];
   teq rres, 0; bne _loop

[ http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html ]