CS-453 (project)
Memory ordering

Sébastien Rouault

Distributed Computing Laboratory

September 24, 2019
Order?

A single thread

// Single thread

```c
int a = 0;
int b = 0;
print(a, b); // a = 0, b = 0

a = 1;
print(a, b); // a = ., b = .

b = 1;
print(a, b); // a = ., b = .
```
But why complicated?

C11/C++11’s solutions

Order!

A single thread

// Single thread

```c
int a = 0;
int b = 0;
print(a, b); // a = 0, b = 0

a = 1;
print(a, b); // a = 1, b = 0

b = 1;
print(a, b); // a = , b = .
```
Order?
A single thread

// Single thread

```c
int a = 0;
int b = 0;
print(a, b); // a = 0, b = 0
a = 1;
print(a, b); // a = 1, b = 0
b = 1;
print(a, b); // a = 1, b = 1
```
Order?

Two threads

// Global var.
int a = 0;
int b = 0;

// Thread A
a = 1; // write
b = 1; // write

// Thread B
auto v = b; // read
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1  □
    // a = 1, v = 0
    // a = 0, v = 1
    // a = 0, v = 0
}
Order?

Two threads

```
// Global var.
int a = 0;
int b = 0;

// Thread A
a = 1; // write
b = 1; // write

// Thread B
auto v = b; // read
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1 ✓
    // a = 1, v = 0
    // a = 0, v = 1
    // a = 0, v = 0
}
```
Order?

Two threads

```
// Global var.
int a = 0;
int b = 0;

// Thread A
a = 1; // write
b = 1; // write

// Thread B
auto v = b; // read
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1 ✓
    // a = 1, v = 0 □
    // a = 0, v = 1
    // a = 0, v = 0
}
```
Order?

But why complicated?

C++11’s solutions

Order!

Two threads

```cpp
// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1; // write
b = 1; // write

// Thread B

auto v = b; // read
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1 ✓
    // a = 1, v = 0 □
    // a = 0, v = 1
    // a = 0, v = 0
}
```
Order?

Two threads

// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1; // write
b = 1; // write

// Thread B

auto v = b; // read
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1 ✓
    // a = 1, v = 0 □
    // a = 0, v = 1 □
    // a = 0, v = 0
}
Order?

Two threads

// Global var.

```cpp
int a = 0;
int b = 0;
```

// Thread A

```cpp
a = 1; // write
b = 1; // write
```

// Thread B

```cpp
auto v = b; // read
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1 ✓
    // a = 1, v = 0 □
    // a = 0, v = 1 ✓
    // a = 0, v = 0
}
```
Order?

Two threads

// Global var.
int a = 0;
int b = 0;

// Thread A
a = 1; // write
b = 1; // write

// Thread B
auto v = b; // read
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1 ✓
    // a = 1, v = 0 □
    // a = 0, v = 1 ✓
    // a = 0, v = 0 □
}

Order?

Two threads

// Global var.

```cpp
int a = 0;
int b = 0;
```

// Thread A

```cpp
a = 1; // write
b = 1; // write
```

// Thread B

```cpp
auto v = b; // read
if (v == 1) {

    print(a, v); // read
    // a = 1, v = 1 ✓
    // a = 1, v = 0 □
    // a = 0, v = 1 ✓
    // a = 0, v = 0 □
}
```
But why complicated? 😞
Order? ☹

But why complicated?

Compiler/hardware reordering

Memory consistency model?

Unrelated R/W (& R/R, W/W) could be carried out-of-order.

(More of that in other courses, e.g., CS-471.)
But why complicated? 😞

It even gets a bit worse...

```cpp
// Global var.
int a = 0;
int b = 0;

// Thread A
a = 1; // write
b = 1; // write

// Thread B
auto v = b; // read U.B.!!
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1  ✔
    // a = 1, v = 0  □
    // a = 0, v = 1  ✔
    // a = 0, v = 0  □
}
```
But why complicated? 😞

Main takeaway

C++11/C++11 do **not** ensure “by default” that reads/writes are carried/observed in program order by different threads
C11/C++11’s solutions 😊
C11/C++11’s solutions 😊

Atomic variables

```c
#include <atomic>

std::atomic<T> foo = T{};
```

With $T$ being:

- Trivially copyable
- Copy and move constructible
- Copy and move assignable
C11/C++11’s solutions 😊

Thread fences

Specifies constraints on the ordering of memory accesses

```
#include <atomic>

std::atomic_thread_fence(std::memory_order_/*...*/);

std::atomic<T> foo = T{};
foo.load(std::memory_order_/*...*/);
foo.store(T{}, std::memory_order_/*...*/);
```
C11/C++11’s solutions 😊

Thread fences

- (consume)
- acquire
- seq_cst
- relaxed
- release
- acq_rel

Least constraining
Most constraining
C11/C++11’s solutions 😊

Thread fences

Least constraining

(consume) acquire release acq_rel

relaxed

seq_cst

Most constraining

atomic_thread_fence(memory_order_relaxed);

read

write

atomic_thread_fence(memory_order_relaxed);

read

write
C11/C++11’s solutions

Thread fences

Order?
But why complicated?
C11/C++11’s solutions
Order!

Least constraining

(read) acquire

seq_cst

Most constraining

atomic_thread_fence(memory_order_acquire);

read
write

atomic_thread_fence(memory_order_acquire);
read
write

Thread fences

Least constraining

Most constraining
Order? But why complicated? C11/C++11’s solutions

Thread fences

Least constraining

Most constraining

(consume) acquire release seq_cst

relaxed acq_rel

atomic_thread_fence(memory_order_release);

read
write
read
write
C11/C++11's solutions 😊

Thread fences

Least constraining

- read
- write

Most constraining

- acquire
- release

atomic_thread_fence(memory_order_acq_rel);
C11/C++11’s solutions 😊

Thread fences

(consume) acquire release acq_rel

read
write

atomic_thread_fence(memory_order_seq_cst);

Most constraining

Least constraining

seq_cst

read
write

total order

relaxed
C11/C++11’s solutions 😊

Thread experiment

// Global
a = 0;
b = 0;
c = 0;
d = 0;

// Threads {0,1}
print(a,c);
print(b,d);

// Threads {2,3}
print(a,c);
print(b,d);
print(a,c);
print(b,d);

// Global
a = 0;
b = 0;
c = 0;
d = 0;
print(a,c);
print(b,d);
print(a,c);
print(b,d);
print(a,c);
print(b,d);

// Threads {0,1}
print(a,c);
print(b,d);
print(a,c);
print(b,d);
print(a,c);
print(b,d);

// Threads {2,3}
print(a,c);
print(b,d);
print(a,c);
print(b,d);
print(a,c);
print(b,d);
## C11/C++11's solutions 😊

### Thread experiment

```cpp
// Global
a = 0;
b = 0;
c = 0;
d = 0;

// Threads {0,1}
print(a,c);  // relaxed
print(b,d);  // relaxed

// Threads {2,3}
print(a,c);  // relaxed
print(b,d);  // relaxed

a = 1;
b = 1;
c = 1;
d = 1;
```

---

**Values of (a,b,c,d) that could be read by threads 0 and 2**

<table>
<thead>
<tr>
<th>Thread 0:</th>
<th>0100</th>
<th>1100</th>
<th>1100</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 2:</td>
<td>0001</td>
<td>1111</td>
<td>0011</td>
<td>0110</td>
</tr>
</tbody>
</table>
C11/C++11’s solutions 😊

Thread experiment

// Global
a = 0;
b = 0;
c = 0;
d = 0;
print(a,c);
print(b,d);
print(a,c);
print(b,d);

// Threads {0,1}
acquire
relaxed
acquire
relaxed

// Threads {2,3}
acquire
relaxed
acquire
relaxed

---

Values of \((a,b,c,d)\) that could be read by threads 0 and 2

<table>
<thead>
<tr>
<th>Thread 0:</th>
<th>0100</th>
<th>1100</th>
<th>1100</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 2:</td>
<td>0001</td>
<td>1111</td>
<td>0011</td>
<td>0110</td>
</tr>
</tbody>
</table>
C11/C++11’s solutions 😊

Thread experiment

// Global
a = 0;
b = 0;
c = 0;
d = 0;

// Threads {0,1}
print(a, c);
print(b, d);
b = 1;

// Threads {2,3}
print(a, c);
print(b, d);
c = 1;
d = 1;

Values of (a,b,c,d) that could be read by threads 0 and 2

Thread 0: 0100 1100 1100 1001
Thread 2: 0001 1111 0011 0110
C++11's solutions

Thread experiment

// Global
a = 0;
b = 0;
c = 0;
d = 0;

// Threads {0,1}
print(a,c);
print(b,d);

// Threads {2,3}
print(a,c);
print(b,d);

// Threads {0,1}
print(a,c);

// Threads {2,3}
print(b,d);

d = 1;
c = 1;

Values of (a,b,c,d) that could be read by threads 0 and 2

Thread 0: 0100 1100 1100 1001
Thread 2: 0001 1111 0011 0110
Order!

Original code

```
// Global var.
int a = 0;
int b = 0;

// Thread A
a = 1; // write
b = 1; // write

// Thread B
auto v = b; // read
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1  ✓
    // a = 1, v = 0  □
    // a = 0, v = 1  ✓
    // a = 0, v = 0  □
}
```
Order!
Corrected code

```
// Global var.

#include <atomic>

int a = 0;
std::atomic<int> b = 0;

// Thread A

a = 1; // write
b = 1; // atomic write

// Thread B

auto v = b; // atomic read
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1 ✓
    // a = 1, v = 0 □
    // a = 0, v = 1 □
    // a = 0, v = 0 □
}
```
Order!

Corrected code

```
// Global var.
#include <atomic>

int a = 0;
std::atomic<int> b = 0;

// Thread A
a = 1; // write
b.store(1, release);

// Thread B
auto v = b.load(acquire);
if (v == 1) {
    print(a, v); // read
    // a = 1, v = 1 ✓
    // a = 1, v = 0 □
    // a = 0, v = 1 □
    // a = 0, v = 0 □
}
```
Order!
I want to know more

Here you go

- [https://preshing.com/...](https://preshing.com/...)
  - 20120612/an-introduction-to-lock-free-programming
  - 20120913/acquire-and-release-semantics

  - atomic{,/memory_order}
  - language/memory_model

- Memory Barriers: a Hardware View for Software Hackers, Paul E. McKenney

Next time

- *Read–Modify–Write* atomic primitives (e.g. compare & swap)
- Workshop: “Writing my own lock”