# CS-453 (project)
# Memory ordering

Sébastien Rouault

Distributed Computing Laboratory

September 24, 2019

# Order?

## A single thread

```
// Single thread

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 = ·
```

# Order?

## A single thread

```
// Single thread

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

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?

## 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?

## 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    ☐
}
```

# But why complicated? ☹

# But why complicated? ☹

## Compiler/hardware reordering

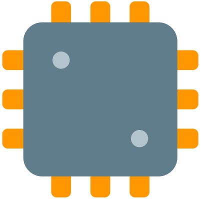
```
a = 1;
b = 1;
```

⇕

```
b = 1;
a = 1;
```

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...

```
// 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

| | |
|---|---|
| C11/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

```
#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

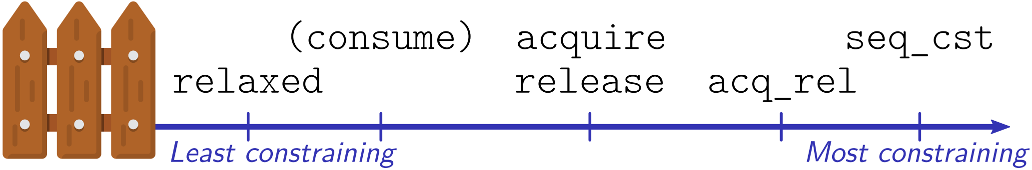
```
#include <atomic>

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

std::atomic<T> foo = T{};
foo.load(std::memory_order_/*...*/);
foo.store(T{}, std::memory_order_/*...*/);
```

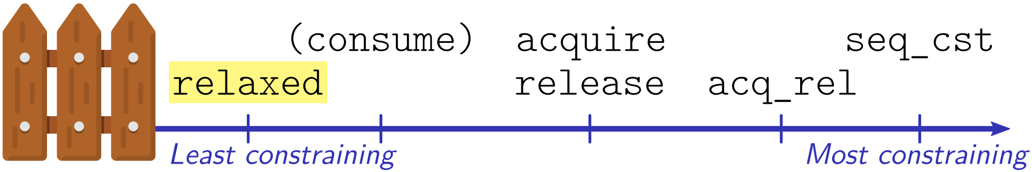
# C11/C++11's solutions ☺

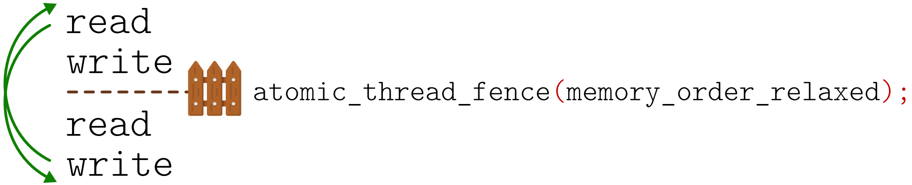
## Thread fences



# C11/C++11's solutions ☺

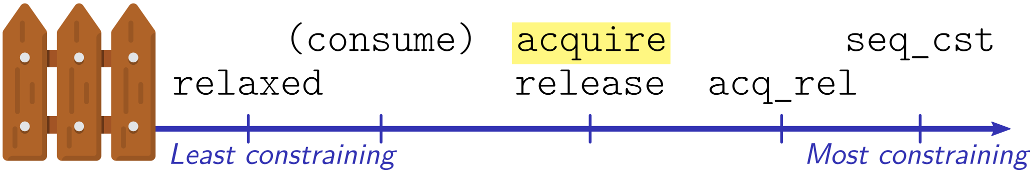
## Thread fences

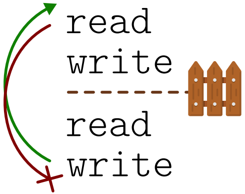


# C11/C++11's solutions ☺

## Thread fences

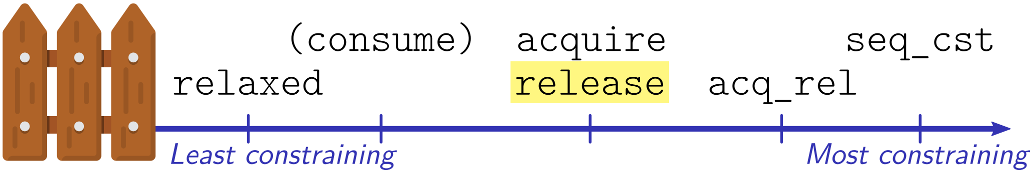


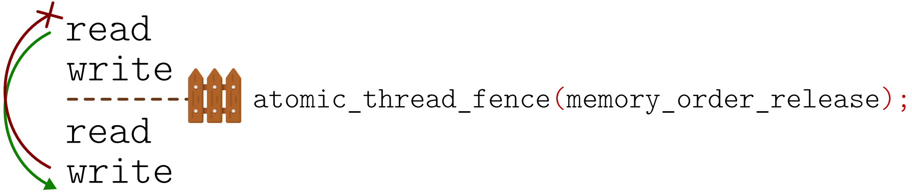




```
atomic_thread_fence(memory_order_acquire);
```

# C11/C++11's solutions ☺

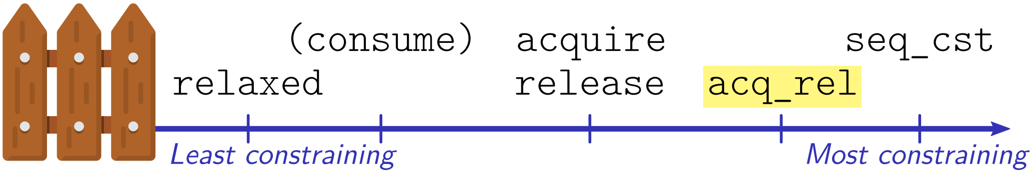
## Thread fences

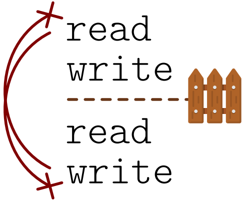


# C11/C++11's solutions ☺

## Thread fences

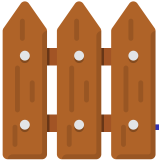



```
atomic_thread_fence(memory_order_acq_rel);
```

# C11/C++11's solutions ☺

## Thread fences



relaxed — (consume) — acquire / release — acq_rel — seq_cst

*Least constraining* → *Most constraining*

read
write
 `atomic_thread_fence(memory_order_seq_cst);`
read
write

total order

# 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;
a = 1;
```

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

# C11/C++11's solutions ☺

## Thread experiment

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

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

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

# C11/C++11's solutions ☺

## Thread experiment

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

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

```
// Threads {2,3}
print(a,c);
-------------------- acquire
print(b,d);
d = 1;
-------------------- relaxed
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 |

# C11/C++11's solutions ☺

## Thread experiment

```
// Global        // Threads {0,1}           // Threads {2,3}
a = 0;           print(a,c);                print(a,c);
b = 0;           ----------- acquire        ----------- acquire
c = 0;           print(b,d);                print(b,d);
d = 0;           b = 1;                     d = 1;
                 ----------- release        ----------- release
                 a = 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~~ |

# C11/C++11's solutions ☺

## Thread experiment

```
// Global        // Threads {0,1}            // Threads {2,3}
a = 0;           print(a,c);                 print(a,c);
b = 0;           ----------- seq_cst         ----------- seq_cst
c = 0;           print(b,d);                 print(b,d);
d = 0;           b = 1;                      d = 1;
                 ----------- seq_cst         ----------- seq_cst
                 a = 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/...
  ... 20120612/an-introduction-to-lock-free-programming
  ... 20120913/acquire-and-release-semantics
- https://en.cppreference.com/w/{c,cpp}/...
  ... 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"