CS-453 (project)

Memory ordering

Sébastien Rouault

Distributed Computing Laboratory

September 25, 2018

Ordering?

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 = ·
```

Ordering?

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 = ·
```

Ordering?

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
```

Ordering?

Two threads

```
// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1;
b = 1;
```

```
// Thread B

if (b == 1) {
      print(a, b);
      // a = 1, b = 1
      // a = 1, b = 0
      // a = 0, b = 1
      // a = 0, b = 0
}
```

□

Ordering?

Two threads

```
// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1;
b = 1;
```

```
// Thread B

if (b == 1) {
      print(a, b);
      // a = 1, b = 1   ☑
      // a = 1, b = 0
      // a = 0, b = 1
      // a = 0, b = 0
}
```

Ordering?

Two threads

```
// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1;
b = 1;
```

```
// Thread B

if (b == 1) {
      print(a, b);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1
      // a = 0, b = 0
}
```

Ordering?

Two threads

```
// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1;
b = 1;
```

```
// Thread B

if (b == 1) {
      print(a, b);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1
      // a = 0, b = 0
}
```

Ordering?

Two threads

```
// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1;
b = 1;
```

```
// Thread B

if (b == 1) {
      print(a, b);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1   ☐
      // a = 0, b = 0
}
```

Ordering?

Two threads

```
// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1;
b = 1;
```

```
// Thread B

if (b == 1) {
      print(a, b);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1   ☑
      // a = 0, b = 0
}
```

Ordering?

Two threads

```
// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1;
b = 1;
```

```
// Thread B

if (b == 1) {
      print(a, b);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1   ☑
      // a = 0, b = 0   ☐
}
```

Ordering?

Two threads

```
// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1;
b = 1;
```

```
// Thread B

if (b == 1) {
      print(a, b);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1   ☑
      // a = 0, b = 0   ☐
}
```

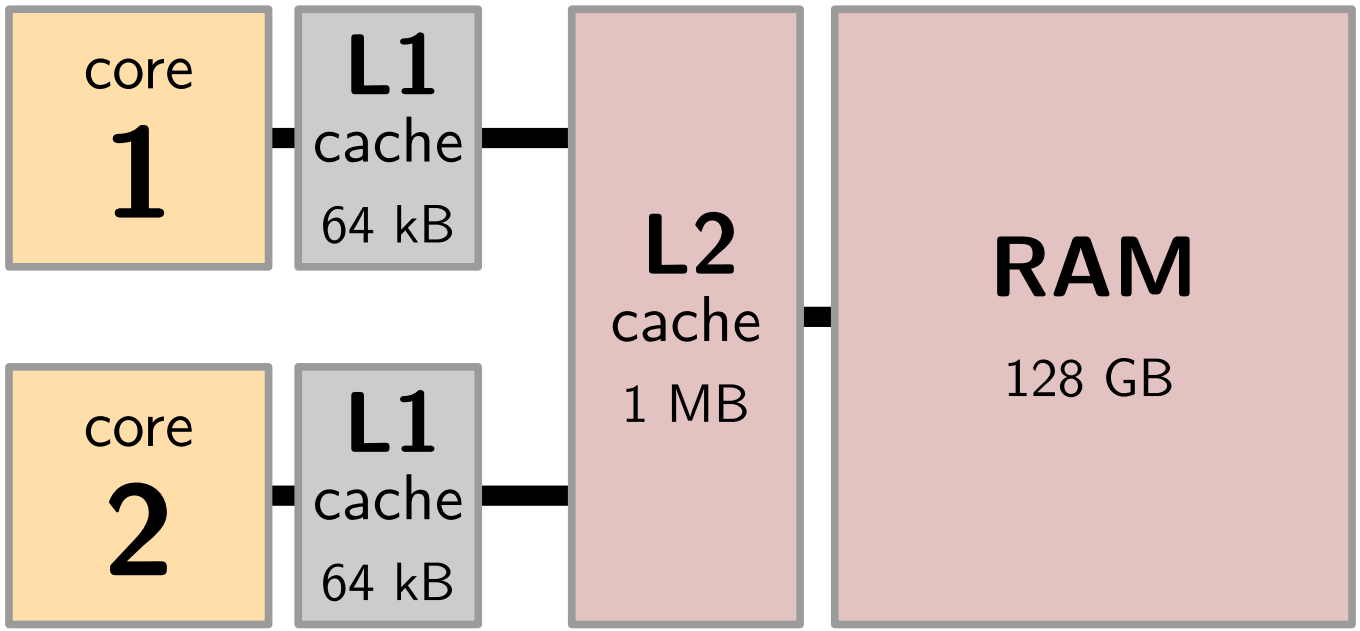
But why complicated? ☹

But why complicated? ☹

Compiler reordering

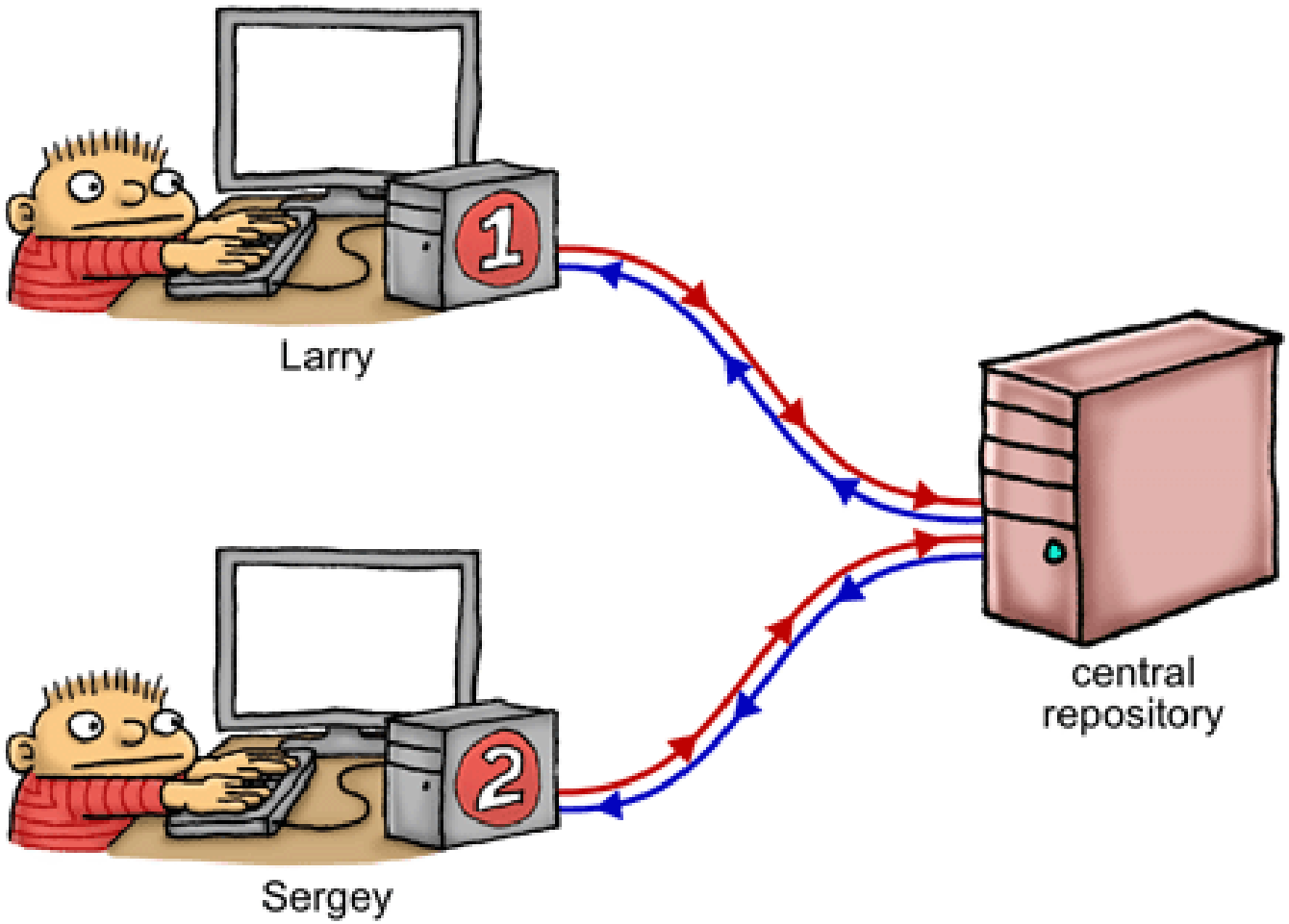
But why complicated? ☹

Hardware reordering



But why complicated? ☹

Hardware reordering



How to tame the dragon?

Memory fences!

How to tame the dragon?

Memory fences!

Acquire **Release**

How to tame the dragon?

Usage (C11)

```
#include <stdatomic.h>

_Atomic int a;
atomic_init(&a, 0);

// Read example
int b = atomic_load_explicit(&v,
      memory_order_acquire);

// Write example
atomic_store_explicit(&v, 1,
      memory_order_release);
```

How to tame the dragon?

Usage (C++11)

```
#include <atomic>
using namespace std;

atomic<int> a = 0;

// Read example
auto b = a.load(memory_order_acquire);

// Write example
a.store(1, memory_order_release);
```

Order(ing)!

Back to the threads

```
// Global var.

#include <atomic>
using namespace std;

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

// Thread A

a.store(1, relaxed);
b.store(1, release);
```

```
// Thread B

auto v = b.load(acquire);
if (v == 1) {
      print(a.load(relaxed), v);
      // a = 1, b = 1   □
      // a = 1, b = 0
      // a = 0, b = 1
      // a = 0, b = 0
}
```

Order(ing)!

Back to the threads

```
// Global var.

#include <atomic>
using namespace std;

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

// Thread A

a.store(1, relaxed);
b.store(1, release);
```

```
// Thread B

auto v = b.load(acquire);
if (v == 1) {
      print(a.load(relaxed), v);
      // a = 1, b = 1   ☑
      // a = 1, b = 0
      // a = 0, b = 1
      // a = 0, b = 0
}
```

Order(ing)!

Back to the threads

```
// Global var.

#include <atomic>
using namespace std;

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

// Thread A

a.store(1, relaxed);
b.store(1, release);
```

```
// Thread B

auto v = b.load(acquire);
if (v == 1) {
      print(a.load(relaxed), v);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1
      // a = 0, b = 0
}
```

Order(ing)!

Back to the threads

```
// Global var.

#include <atomic>
using namespace std;

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

// Thread A

a.store(1, relaxed);
b.store(1, release);
```

```
// Thread B

auto v = b.load(acquire);
if (v == 1) {
      print(a.load(relaxed), v);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1
      // a = 0, b = 0
}
```

Order(ing)!

Back to the threads

```
// Global var.

#include <atomic>
using namespace std;

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

// Thread A

a.store(1, relaxed);
b.store(1, release);
```

```
// Thread B

auto v = b.load(acquire);
if (v == 1) {
      print(a.load(relaxed), v);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1   ☐
      // a = 0, b = 0
}
```

Order(ing)!

Back to the threads

```
// Global var.

#include <atomic>
using namespace std;

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

// Thread A

a.store(1, relaxed);
b.store(1, release);
```

```
// Thread B

auto v = b.load(acquire);
if (v == 1) {
      print(a.load(relaxed), v);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1   ☐
      // a = 0, b = 0
}
```

Order(ing)!

Back to the threads

```
// Global var.

#include <atomic>
using namespace std;

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

// Thread A

a.store(1, relaxed);
b.store(1, release);
```

```
// Thread B

auto v = b.load(acquire);
if (v == 1) {
      print(a.load(relaxed), v);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1   ☐
      // a = 0, b = 0   ☐
}
```

Order(ing)!

Back to the threads

```
// Global var.

#include <atomic>
using namespace std;

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

// Thread A

a.store(1, relaxed);
b.store(1, release);
```

```
// Thread B

auto v = b.load(acquire);
if (v == 1) {
      print(a.load(relaxed), v);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1   ☐
      // a = 0, b = 0   ☐
}
```

Order(ing)!

Back to the threads

```
// Global var.

#include <atomic>
using namespace std;

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

// Thread A

a = 1;
b.store(1, release);
```

```
// Thread B

auto v = b.load(acquire);
if (v == 1) {
      print(a, v);
      // a = 1, b = 1   ☑
      // a = 1, b = 0   ☐
      // a = 0, b = 1   ☐
      // a = 0, b = 0   ☐
}
```

Conclusion

Simple concepts, but *powerful & complicated* language

- https://en.cppreference.com/w/c/atomic
- https://en.cppreference.com/w/cpp/atomic
- https://preshing.com/20130922/acquire-and-release-fences

Next time, exciting stuffs!

- More atomic primitives (fetch & add, compare & swap)
- Workshop: "writing my own lock"