CS3211 Tutorial 2

Atomics in C++

(AY 25/26 Semester 2)

February 5, 2026

(Compiled by Benson, thanks to all past and current TAs!)

Join our telegram group!
https://t.me/+MqCIT5ObVgxiMzRl

Contents

Why Atomics?

Memory Orders

Contents

Why Atomics?

Memory Orders

Data Races and Undefined Behaviour

Thread 1:

```
void test1() {
  a = 1;
  b = 1;
  flag = 1;
}
```

Thread 2:

```
void test2() {
  std::cout << "init a: " << a << "\n";
  std::cout << "init b: " << b << "\n";

  while(flag == 0) {}

  std::cout << "final a: " << a << "\n";
  std::cout << "final b: " << b << "\n";
}
```

Link to fsmbolt

Mutex vs. Atomics

Mutexes:

```
1  call     __gthrw_pthread_mutex_lock(pthread_mutex_t*)
2  test     eax, eax
3  jne      .L20
4  mov      edi, OFFSET FLAT:m
5  add      DWORD PTR counter[rip], 1
6  call     __gthrw_pthread_mutex_unlock(pthread_mutex_t*)
```

Atomics:

```
1  lock add        DWORD PTR atomic_counter[rip], 1
```

Link to fsmbolt

Contents

Why Atomics?

Memory Orders

Sequentially Consistent: Our Mental Model

Thread 1:

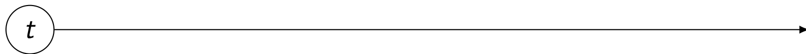
```
x.store(1, stdmo::seq_cst);
x.store(2, stdmo::seq_cst);
```

Thread 2:

```
y.store(3, stdmo::seq_cst);
x.store(4, stdmo::seq_cst);
```

Thread 3:

```
x.load(stdmo::seq_cst);
x.load(stdmo::seq_cst);
y.load(stdmo::seq_cst);
```



Why relaxing guarantees?

Sequential consistency (Link to fsmbolt):

```
1  xchg    edx, DWORD PTR t1_ctr[rip]
```

- XCHG: If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

Acquire-Release (Link to fsmbolt) / Relaxed (Link to fsmbolt):

```
1  mov     DWORD PTR t1_ctr[rip], eax
```

Relaxed Ordering

Thread 1:

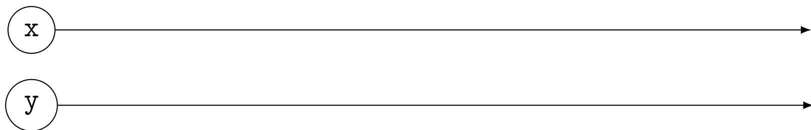
```
x.store(1, stdmo::relaxed);
x.store(2, stdmo::relaxed);
```

Thread 2:

```
y.store(3, stdmo::relaxed);
x.store(4, stdmo::relaxed);
```

Threads 3 & 4:

```
x.load(stdmo::relaxed);
x.load(stdmo::relaxed);
y.load(stdmo::relaxed);
```



Modification Order: All modifications to any particular atomic variable occur in a *total order* that is specific to this one atomic variable.

“Happens-Before”

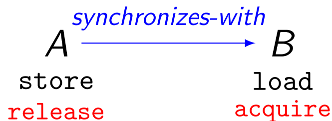
“Happens-Before”

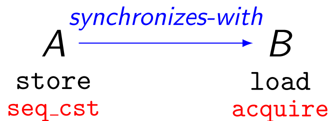
“Happens-Before”

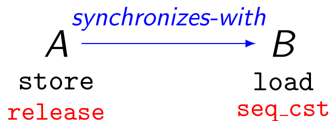
- *sequenced-before*: Program order in the **same thread**.

```
x.store(1, stdmo::relaxed);
x.store(2, stdmo::relaxed);
```

- *synchronized-with*: A suitably-tagged atomic read operation **successfully reads** from a suitably-tagged atomic write operation (or a subsequent one).
  - Depends on the exact execution!







“Happens-Before”

(Simply) happens before: Explained by chaining *sequence-before* and *sychronized-with*.

Thread 1:

```
x.store(1, stdmo::relaxed); //a
y.store(2, stdmo::release); //b
```

Thread 2:

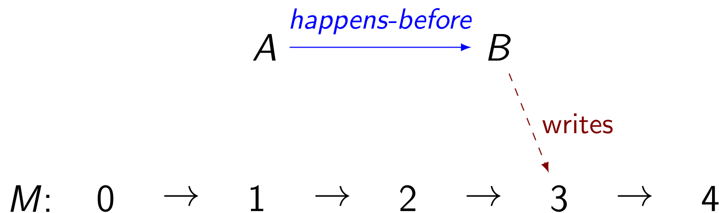
```
y.load(stdmo::acquire);     //x
x.load(stdmo::relaxed);     //y
```

Does (a) simply happen before (y)?

“Happens-Before” and Modification Order

Write-Write Coherence:

If write A on M *happens-before* another write B on M, then A appears earlier than B in the modification order of M.



What about reads? – Coherence Order

Thread 1:

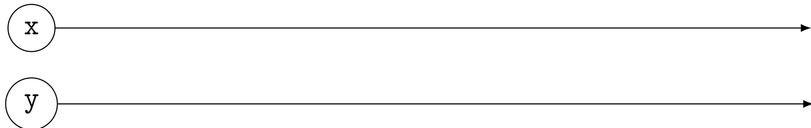
```
x.store(1, stdmo::relaxed);
x.store(2, stdmo::relaxed);
```

Thread 2:

```
y.store(3, stdmo::relaxed);
x.store(4, stdmo::relaxed);
```

Threads 3 & 4:

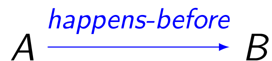
```
x.load(stdmo::relaxed);
x.load(stdmo::relaxed);
y.load(stdmo::relaxed);
```



Coherence order: The read operations will always read the value written by the **preceding write**.

What about reads? – Coherence Order

What is Read-Read Coherence? Read-Write Coherence? Write-Read Coherence?



M: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$

What about reads? – Coherence Order

Thread 1:

```
x.store(1, stdmo::relaxed);
x.store(3, stdmo::relaxed);
x.store(5, stdmo::relaxed);
x.store(7, stdmo::relaxed);
x.store(9, stdmo::relaxed);
```

Thread 2:

```
x.store(0, stdmo::relaxed);
x.store(2, stdmo::relaxed);
x.store(4, stdmo::relaxed);
x.store(6, stdmo::relaxed);
x.store(8, stdmo::relaxed);
```

Threads 3 & 4:

```
// read 10 times
for (i = 0; i <= 9; i++) {
    x.load(stdmo::relaxed);
}
```

Is it possible to observe the following outputs?

A. Thread 3: 1 1 3 2 6 5 9 9 9 9
   Thread 4: 0 1 4 6 8 5 9 9 9 9

B. Thread 3: 1 2 6 6 3 7 9 9 9 9
   Thread 4: 1 1 3 4 8 9 9 9 9 9

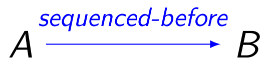
A is possible with modification order $0 \to 1 \to 3 \to 2 \to 4 \to 6 \to 8 \to 5 \to 7 \to 9$. B is not possible.

Strongly Happens-Before

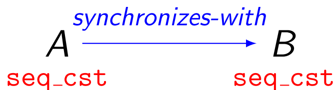
Specifies which operations **see the effects** of which other operations.

- Happens-before is NOT enough!

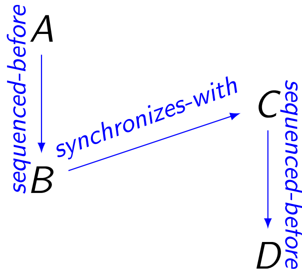
Program Order



Sequentially consistent operations



Non-sequentially consistent operations



Extra Reading: Why do we have strongly happens-before?

Code Snippet A

Thread 1:

```
1  x.store(1, stdmo::relaxed);   //a
2  y.store(2, stdmo::release);   //b
```

Thread 2:

```
1  while (y.load(stdmo::acquire) != 2) { }  //p
2  cout << x.load(stdmo::relaxed);          //q
```

- What if `x` is not atomic?
- What if we remove the `while` loop?

Code Snippet B

Thread 1:

```
1  x.store(1, stdmo::relaxed);   //a
2  y.store(2, stdmo::release);   //b
3  x.store(3, stdmo::relaxed);   //c
4  z.store(4, stdmo::release);   //d
5  x.store(5, stdmo::relaxed);   //e
```

Thread 2:

```
1  while (y.load(stdmo::acquire) != 2) { }  //p
2  cout << x.load(stdmo::relaxed);          //q
3  while (z.load(stdmo::acquire) != 4) { }  //r
4  cout << x.load(stdmo::relaxed);          //s
```

- What if the first cout prints a 5?
- What if x is not atomic?

Code Snippet C

Thread 1:

```
1  x.store(1, stdmo::relaxed);   //a
2  y.store(2, stdmo::release);   //b
```

Thread 2:

```
1  x.store(3, stdmo::relaxed);   //p
2  z.store(4, stdmo::release);   //q
3  x.store(5, stdmo::relaxed);   //r
```

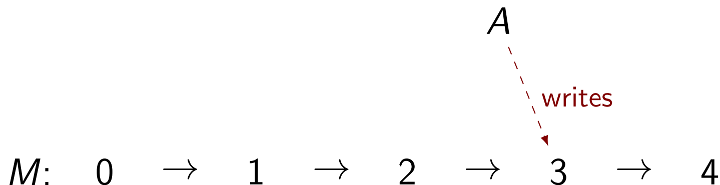
Thread 3:

```
1  while (y.load(stdmo::acquire) != 2) { }  //t
2  cout << x.load(stdmo::relaxed);          //u
3  while (z.load(stdmo::acquire) != 4) { }  //v
4  cout << x.load(stdmo::relaxed);          //w
```

- What if the first cout prints a 5?
- What if the second cout prints a 1?

Read-Modify-Write (RMW) Operations

RMW operations are **indivisible**, i.e. the read and the write must come together in the coherence order.



Examples: `fetch_add`, `fetch_sub`, `exchange`, `compare_exchange_weak`, `compare_exchange_strong`.

Code Snippet D

Thread 1:

```
1  // Assume (a) reads 0 and writes 1
2  x.fetch_add(1, stdmo::relaxed);  //a
```

Thread 2:

```
1  // Can (p) read 0 and write 1?
2  x.fetch_add(1, stdmo::relaxed);  //p
```

Code Snippet E

Thread 1:

```
1  x.fetch_add(1, stdmo::relaxed);   //a
2  x.fetch_add(1, stdmo::relaxed);   //b
```

Thread 2:

```
1  x.fetch_add(1, stdmo::relaxed);   //p
2  x.fetch_add(1, stdmo::relaxed);   //q
```

Code Snippet F

Is it possible for `(a)` to read the value 1 and write 2 to `x`, and for `(p)` to read the value 1 and write 2 to `y`, all in the same execution?

Thread 1:

```
1  x.fetch_add(1, stdmo::relaxed);   //a
2  y.fetch_add(1, stdmo::relaxed);   //b
```

Thread 2:

```
1  y.fetch_add(1, stdmo::relaxed);   //p
2  x.fetch_add(1, stdmo::relaxed);   //q
```

Extra Exercise 1

Extra Slide

Thread 1:

```
x.store(true, stdmo::seq_cst);
```

Thread 2:

```
y.store(true, stdmo::seq_cst);
```

Thread 3:

```
while (!x.load(stdmo::seq_cst)) {}
if (y.load(stdmo::seq_cst)
  z.fetch_add(1, stdmo::seq_cst);
```

Thread 4:

```
while (!y.load(stdmo::seq_cst)) {}
if (x.load(stdmo::seq_cst)
  z.fetch_add(1, stdmo::seq_cst);
```

Question: Can the final value of z be 0?

- What if we use acquire/release semantics instead? How about relaxed memory ordering?
- What if z is a non-atomic variable?

Extra Exercise 2

Extra Slide

Thread 1:

```
1 x.fetch_add(1, stdmo::relaxed); // a
2 y.store(2, stdmo::release);     // b
```

Thread 2:

```
1 x.fetch_add(1, stdmo::relaxed); // p
2 y.store(2, stdmo::release)?     // q
```

Thread 3:

```
1 while (y.load(stdmo::acquire) != 2) {} //x
2 x.load(stdmo::relaxed);                //y
```

Question:

- Is there a *synchronizes-with* relation for both (b)-(x) and (q)-(x)? Or is it just one of them?

Extra Exercise 3

Extra Slide

Thread 1:

```
1  x.store(1, stdmo::seq_cst);   // a
2  y.store(1, stdmo::release);   // b
```

Thread 2:

```
1  r1 = y.fetch_add(1, stdmo::seq_cst); // p
2  r2 = y.load(stdmo::relaxed);         // q
```

Thread 3:

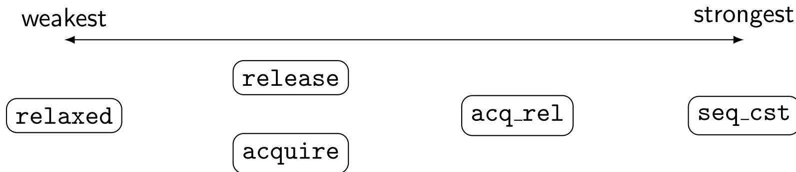
```
1  y.store(3, stdmo::seq_cst);    // x
2  r3 = x.load(stdmo::seq_cst);   // y
```

Remember that `fetch_add` returns the old value.

Question: Is it possible that `r1 = 1`, `r2 = 3` and `r3 = 0`? (Hint: Recall the difference between simply happens before and strongly happens before.)

Summary

- Why atomics over traditional synchronization? Why non-sequentially-consistent atomics?
  - Speed! Maximizing concurrency!
- The guarantees that programs have, based on the different memory orderings.
  - Modification Order, Coherence Order
  - Strongly happens before: Operations must see the effects of those other operations.
  - Memory Orders:



Attendance

Session ID: 1939488

CS3211 Tutorial 2

5 February 2026 14:00-16:00

BIZ2-02-02 - SEMINAR ROOM 2-2

That's it!

Anonymous Feedback (throughout the semester):

https://forms.gle/6a9T4t88wNwYxG4G8

The link to the tutorial slides will be posted in telegram.