Tutorial 2

C++ atomics

admin stuff

Email: travis.toh@u.nus.edu

Please don’t PM me for debugging questions

Please peruse the tutorial sheet prior especially since this is a Friday class:)

I will show QR code for survey/attendance taking near 1 hour mark

I will post tut slides in Tele group by end of the day

Learning objectives

Atomics vs Mutexes

3 C++ memory orders

- SeqCst
- AcqRel
- Relaxed

Today we build mental models to reason about memory orders

recap: data races in C++

What is a data race?

1. two or more accesses to the memory location
2. at least one of them is a write
3. accesses are on different threads, at least one is not *atomic*, and without any *happens-before* relationship

last week: using std::mutex to fix races

```
std::thread([]() {
    auto lk = std::unique_lock(mtx);
    counter++;
});
```

this week: using std::atomic to fix races

```
auto start = sc::high_resolution_clock::now();
for(int i = 0; i < LOOPS; i++)
{
    // TODO: add atomic implementation here
}
```

seq_cst vs acquire/release vs relaxed

try it yourself
Section 2.2

seq_cst vs acquire/release vs relaxed

sequentially consistent:
– very predictable behaviour
– potentially very slow

relaxed atomics:
– not-so-predictable behaviour
– (can be) significantly faster

To reason about acq/rel and relaxed models, we use a mental model based on Synchronises-with and Sequenced-before relationships.

modification order

[4] All modifications to a particular atomic object *M* occur in some particular total order, called the *modification order* of *M*.

[*Note 3*: There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads can observe modifications to different objects in inconsistent orders. — *end note*]

— barney soursop

optimisations and reordering

try it yourself
Section 3.2

modification order & coherence

- R–R: if a read **R1** *happens-before* a read **R2**, and **R1** read some value **X**, then **R2** must read either **X**, or a value **Y** written by someone that is after **X** in the **MO**.

- R–W: if a read **R** *happens-before* a write **W**, then **R** must read a value written by someone before **W** in the **MO**.

modification order & coherence

- W–R: if a write **W** *happens-before* a read **R**, then **R** must read either the value written by **W**, or a value written by someone after **W** in the **MO**.

- W–W: if a write **W1** *happens-before* a write **W2**, then **W1** is before **W2** in the **MO**.

optimisations and reordering

19 [*Note 19*: The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations. — *end note*]

memory orders (acquire/release)

key point

acquire-release helps you establish *synchronises-with*

requirements for synchronises-with

2 An atomic operation A that performs a release operation on an atomic object M synchronizes with an atomic operation B that performs an acquire operation on M and takes its value from any side effect in the release sequence headed by A.

requirements for synchronises-with

“and takes its value from any side effect in the release sequence headed by A”

don’t worry about release sequences:
just think of it as “***the release store of A***”

recap: sequenced-before

in all examples below, **AAA** is *sequenced-before* **BBB**

```
AAA; BBB;        // (1)
AAA, BBB;        // (2)
AAA && BBB;      // (3)
AAA || BBB;      // (4)
// ... many more
```

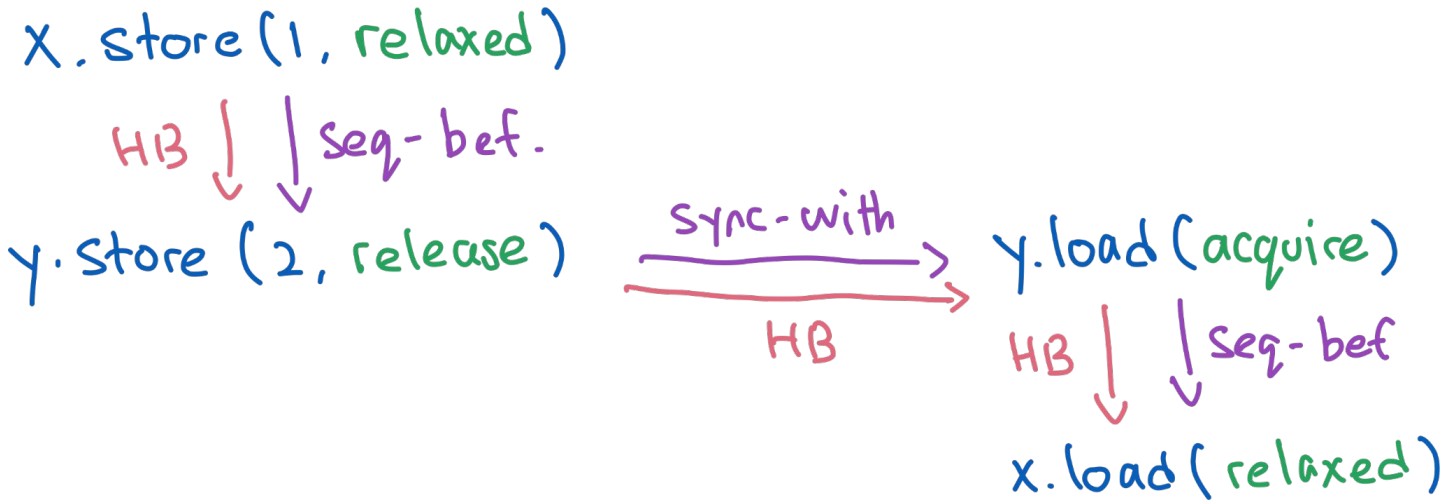
Happens-before

1) A is sequenced-before B
2) A inter-thread happens before B (includes synchronises-with)

code snippet A

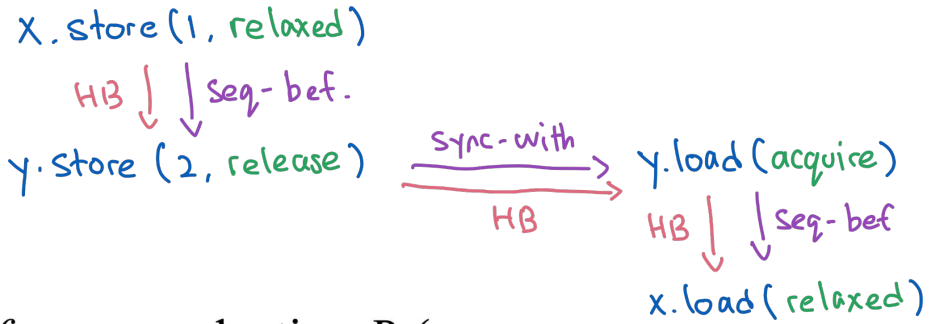
What are the SW, SB, and HB relationships?

code snippet A



code snippet A

why does **(a)** *happen-before* **(b)**, and **(c)** *happen-before* **(d)**?



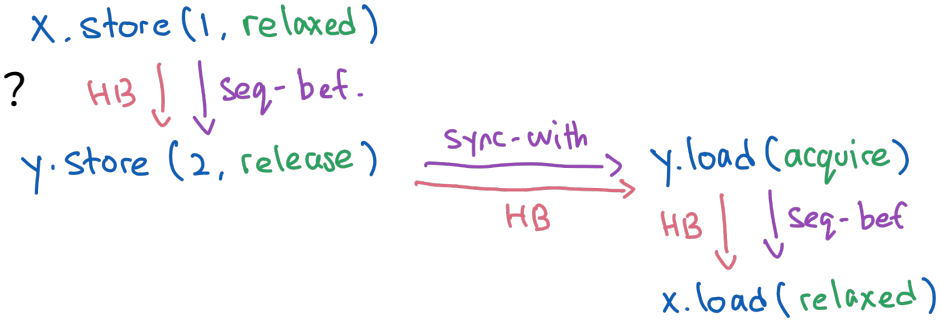
10 An evaluation *A* *happens before* an evaluation *B* (or, equivalently, *B* *happens after* *A*) if:

(10.1) — *A* is sequenced before *B*, or

(10.2) — *A* inter-thread happens before *B*.

code snippet A

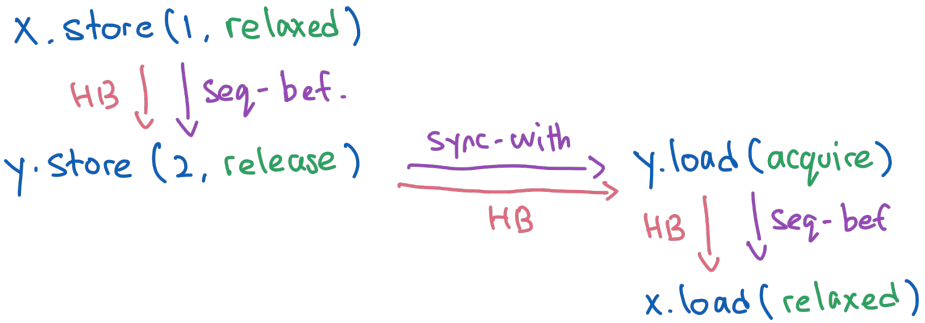
why does **(b)** *synchronise-with* **(c)**?



2 An atomic operation A that performs a release operation on an atomic object M synchronizes with an atomic operation B that performs an acquire operation on M and takes its value from any side effect in the release sequence headed by A.

code snippet A

why does **(b)** *happen-before* **(c)**?



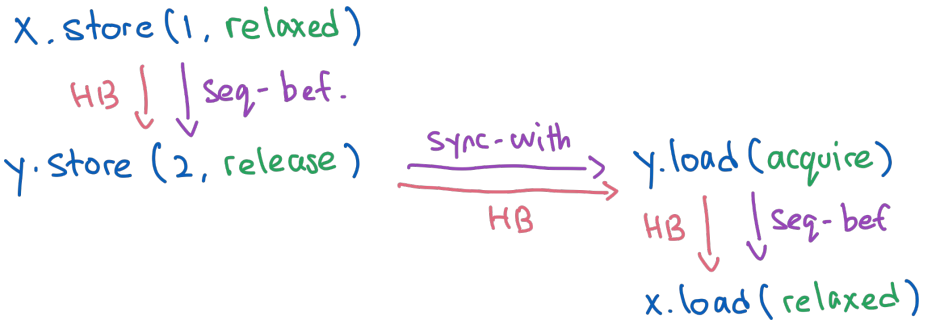
9 An evaluation *A inter-thread happens before* an evaluation *B* if

(9.1) — *A* synchronizes with *B*, or

code snippet A

why does **(a)** *happen-before* **(d)**?

happens-before is a
transitive relation



9 An evaluation *A inter-thread happens before* an evaluation *B* if

(9.3) — for some evaluation *X*

(9.3.1) — *A* synchronizes with *X* and *X* is sequenced before *B*, or

(9.3.2) — *A* is sequenced before *X* and *X* inter-thread happens before *B*, or

(9.3.3) — *A* inter-thread happens before *X* and *X* inter-thread happens before *B*.

code snippet A

Can we ever print x = 0?

No! Due to coherence rules, since **(d)** *happens-after* **(a)**, **(d)** must read either the value stored by **(a)**, or a value later in the MO (none here)

(choose R–W or W–R coherence to explain, up to you)

code snippet A

what if x is not atomic?

recall what a data race is:

3. accesses are on different threads, at least one is not *atomic*, and **without any *happens-before*** relationship

we established *happens-before*! so we are **race-free**.

requirements for synchronises-with

```
x.store(1, relaxed);
y.store(2, release);
```

```
z.load(acquire);
print(x.load(relaxed));
```

does not work! releasing **y** but acquiring **z**!

requirements for synchronises-with

```
x.store(1, relaxed);
y.store(2, release);
```

```
y.load(acquire);
print(x.load(relaxed));
```

does not work! what if y.load() returns 0?

this is why we always use loops

code snippet B

.....

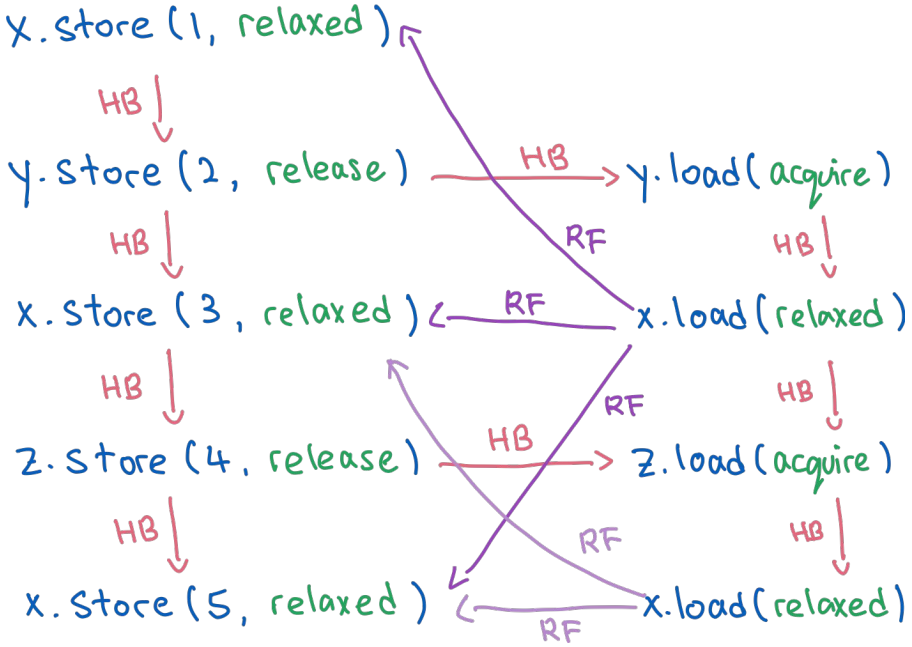
code snippet B

same idea as code snippet A

what can the first print output? x = 1, 3, or 5 — **never** 0!

second print? x = 3 or 5 — **never** 0 or 1!

code snippet B



modification order of X:
1, 3, 5

code snippet B

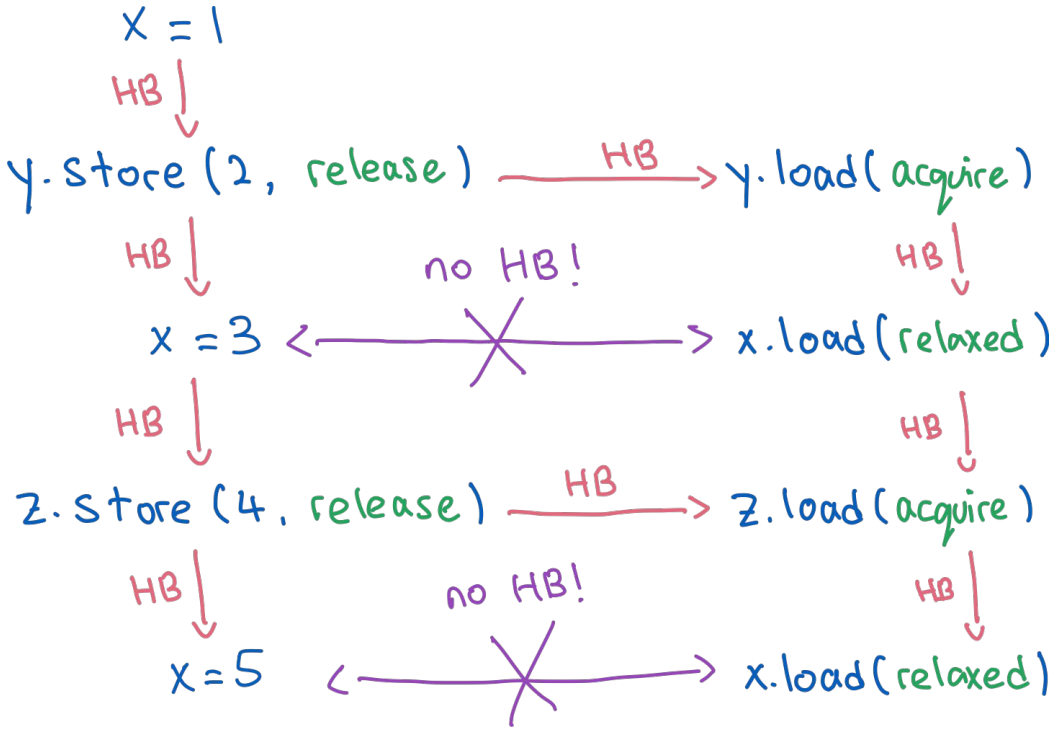
1. **the first load can return any value in the MO**
   ie. 1, 3 or 5

2. **the second load *cannot* return an earlier value in the MO**
   eg. if the first load returns 3, the second load **cannot** return 1

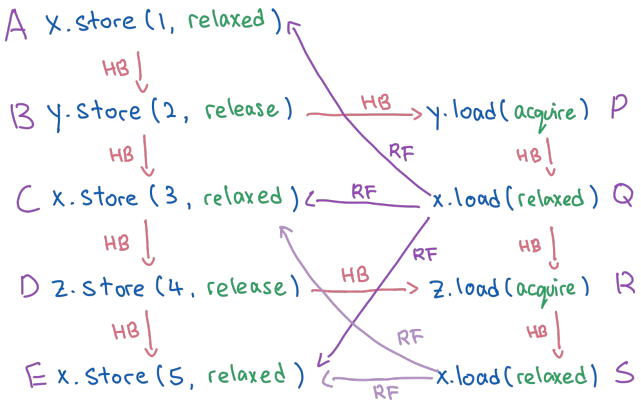
code snippet B

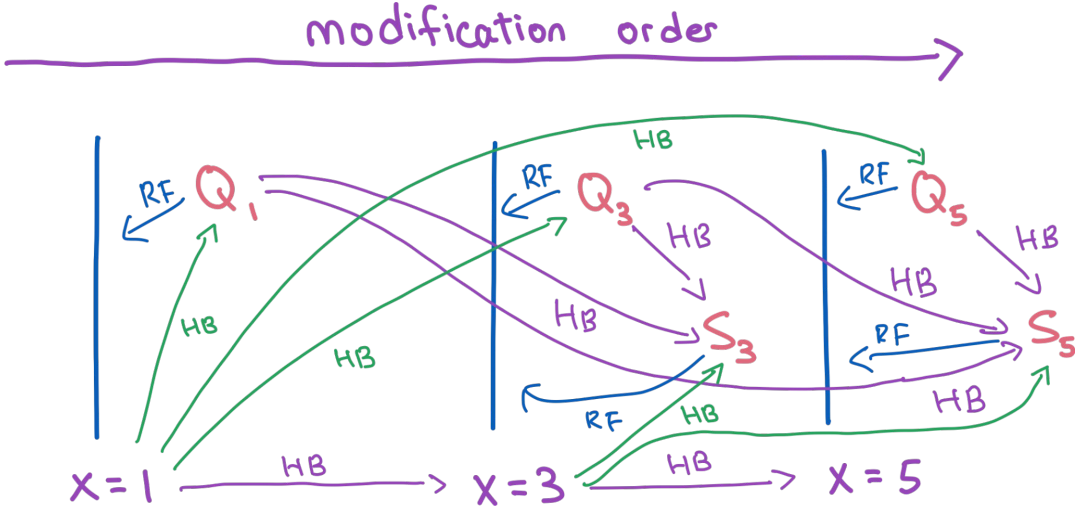
what if X is not atomic?

data race!



recap: modification order

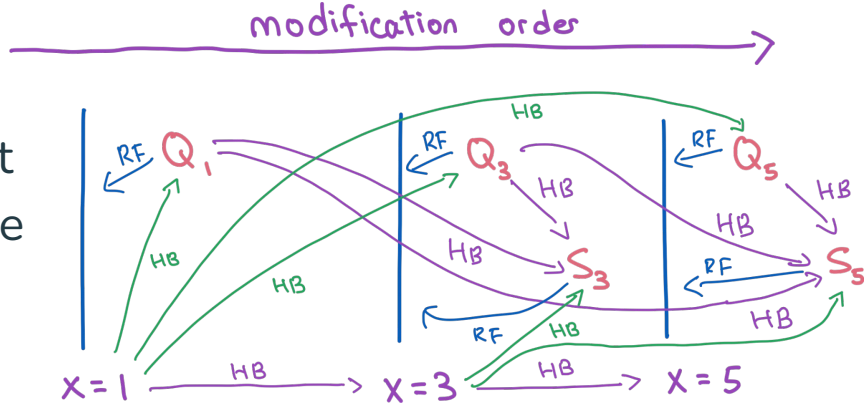




recap: modification order

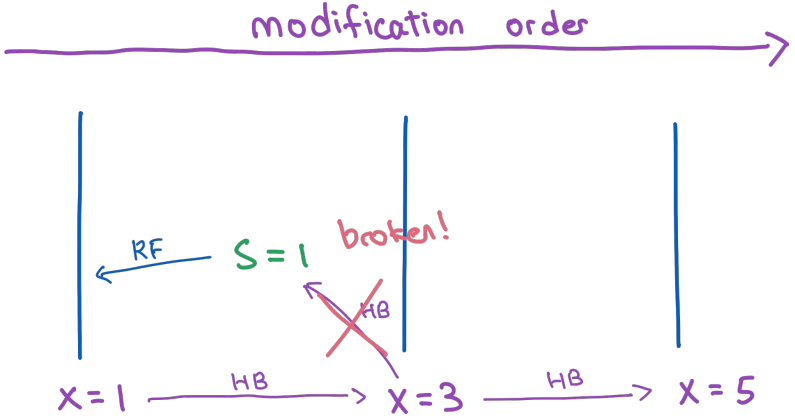
note: refer to code snippet B

1. *happens-before* cannot point left
2. reads *read-from* the closest write to their left (by definition)



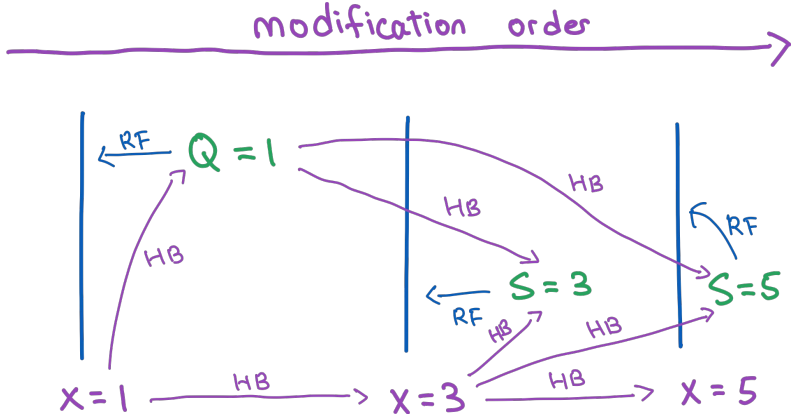
recap: modification order

S can never read 1



recap: modification order

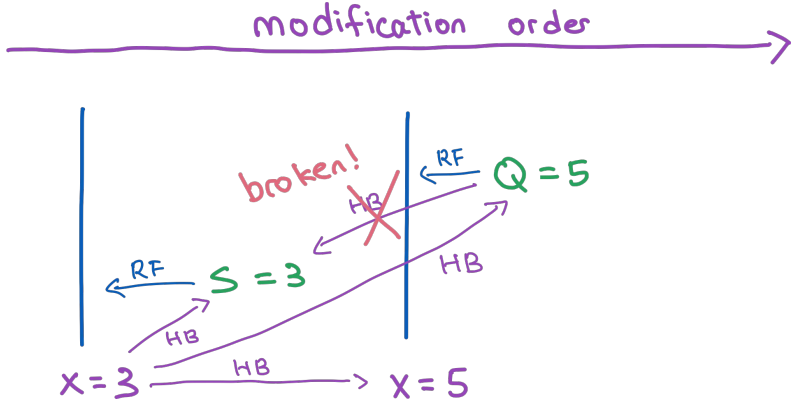
If Q reads **1** (Q_1), S can read either **3** (S_3) or **5** (S_5)



recap: modification order

If Q reads **5** (Q_5), S cannot read **3** (S_3):

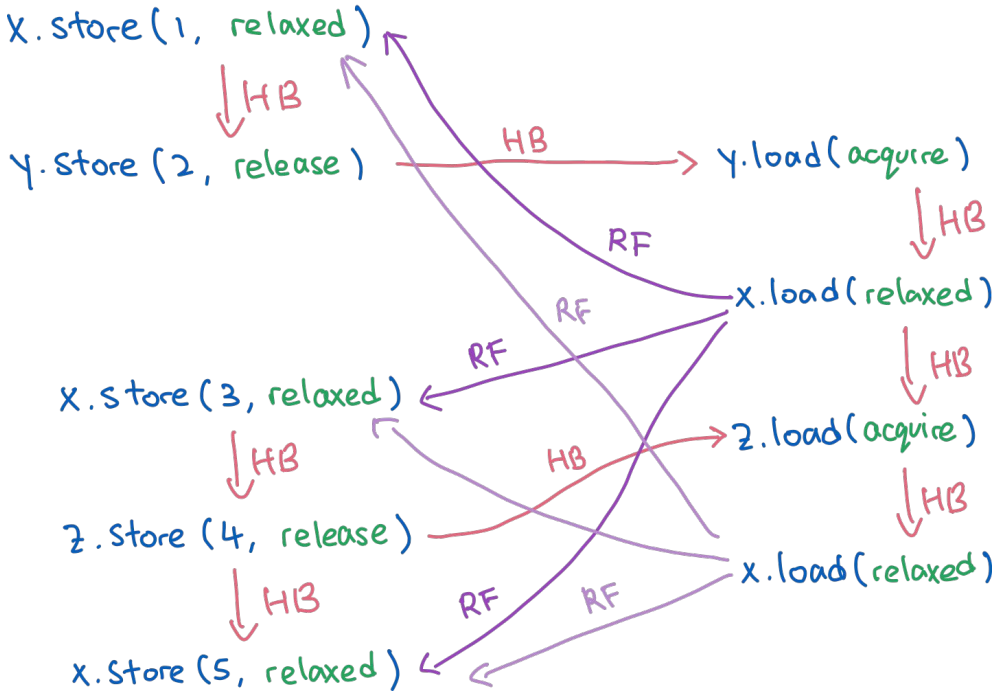
a. we know Q *happens-before* S
b. we would need to make Q_5 *HB* S_3!



code snippet C

.....

code snippet C



modification order of X:

- 1, 3, 5
- 3, 1, 5
- 3, 5, 1

why? no *HB* between T1 and T2

code snippet C

the first load and second loads can return **1, 3, or 5!**

actual possibilities:
(1, 1), (1, 3), (1, 5), (3, 3), (3, 5), (5, 5)

code snippet C

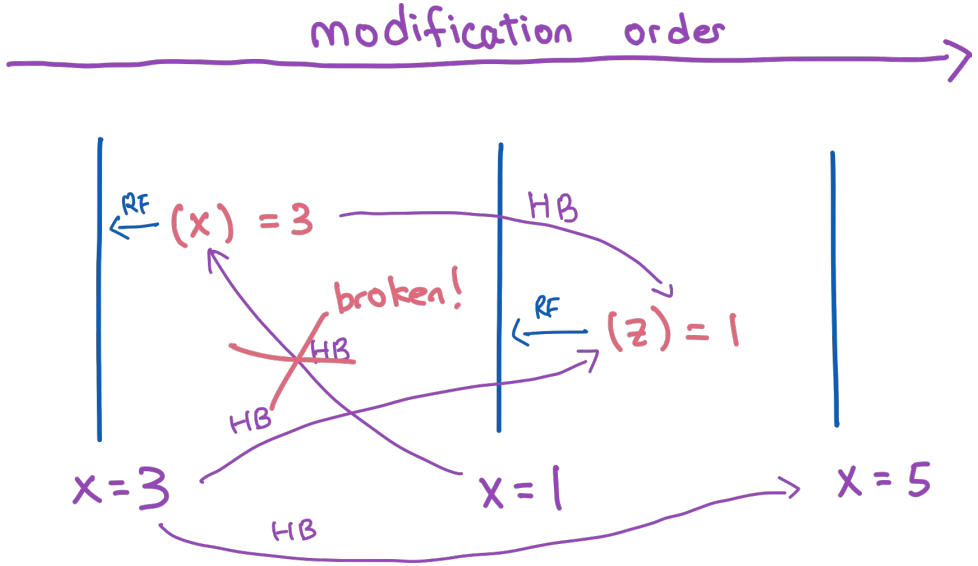
why can't the second load return a value smaller than the first load?

suppose the first load returns 3 (same argument for 5)

- W–R coherence: `store(1)` is earlier in the **MO** than `store(3)`
- therefore, the MO could **not** have been 3, 5, 1 or 3, 1, 5!
- leaving only 1, 3, 5 → second load must read 3 or 5.
  (R–R coherence)

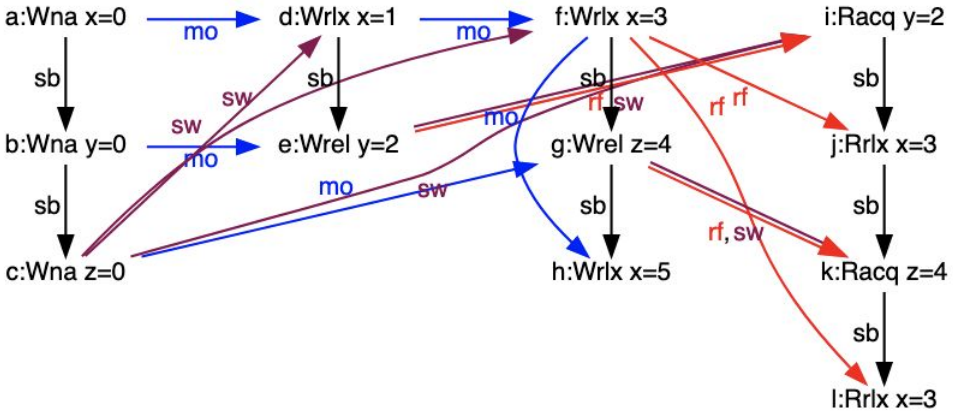
code snippet C

using the MO diagram, we can show that (3, 1) is not possible



cool website

http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/



Old mental model based on reordering does not work anymore

```
x.store(1, stdmo::relaxed);
y.store(x.load(stdmo::relaxed), stdmo::release);
x.store(3, stdmo::relaxed);
```

while another thread may observe x = 3 before y = 1, we cannot say it is reordered like so

```
x.store(1, stdmo::relaxed);
x.store(3, stdmo::relaxed);
y.store(x.load(stdmo::relaxed), stdmo::release);
```

because that might imply that it is possible that y = 3 which is not possible.

```
x.store(1, stdmo::relaxed);
tmp = x.load(stdmo::relaxed);
x.store(3, stdmo::relaxed);
y.store(tmp, stdmo::release);
```

Feedback & attendance

Have a great weekend ahead!

Slide adapted and modified from Zhiayang