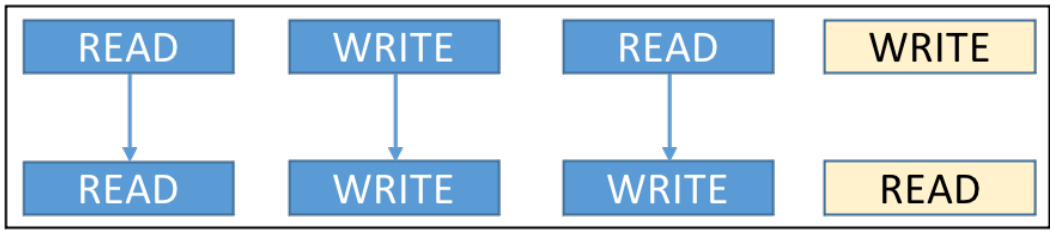
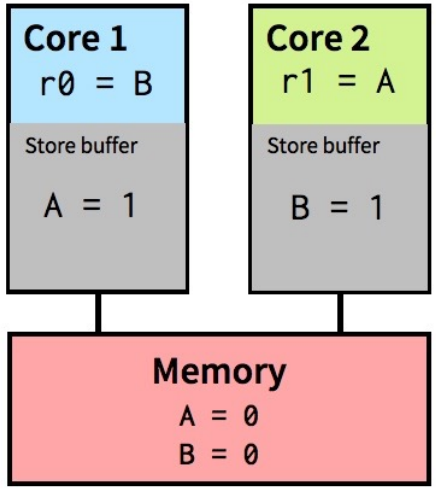
Lecture 10: Language Memory Model

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture (Recap)



- **Rule-5**: Concurrent Writes by two cores can be seen in different order
  - Each core may perceive its own Write occurring before that of other



Thread 1	Thread 2
(1)	(3)
(2)	(4)

Can `r0 = 0` and `r1 = 0`?

Executed

```
r0 = B (= 0)
r1 = A (= 0)
```

- x86-TSO memory model (Intel/AMD)
- Store buffer

Today's Class

- Mutex lock v/s atomic variable
- C++ memory model

Sequential Consistency for Data Race Free (DRF)

- C++ memory model guarantees sequential consistency for Data-Race-Free (DRF) code blocks in our program
  - No guarantees whatsoever for rest of the program
    - Expect reordering everywhere else!
- Sequential consistency for DRF
  - No guarantee for racy programs
  - Followed by almost all language memory model
  - Allows all possible optimizations by compiler and hardware in rest of the code

Memory Operations in C++

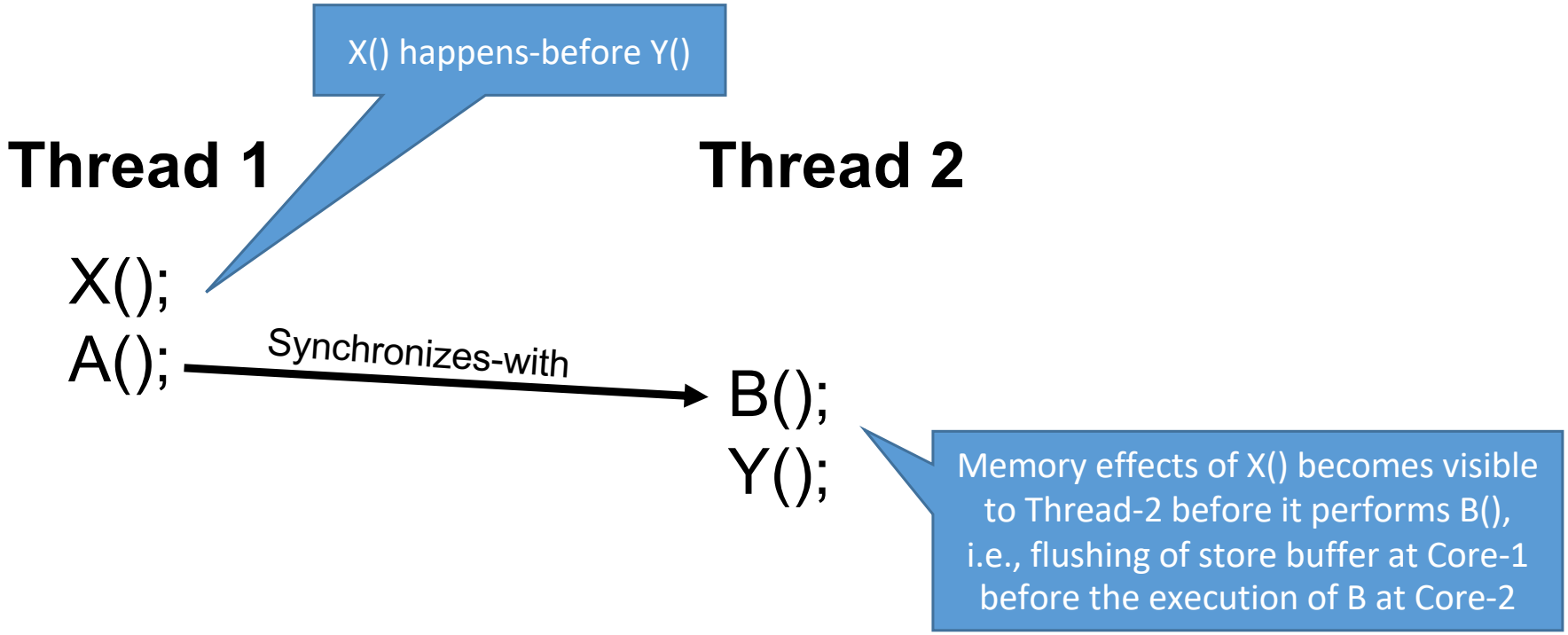
- Synchronization operations
  - Lock based operations
    - Mutex lock/unlock
  - Lock-free (A.K.A. atomic) operations
    - Atomic load (read)
    - Atomic store (write)
    - Atomic Read-Modify-Write (RMW), i.e., compare and swap on x86 platforms
      - E.g., std::atomic<int>::fetch_add
- Non-synchronization operations
  - Read/write

Achieving synchronization in an easy-way (high productivity, but low performance)

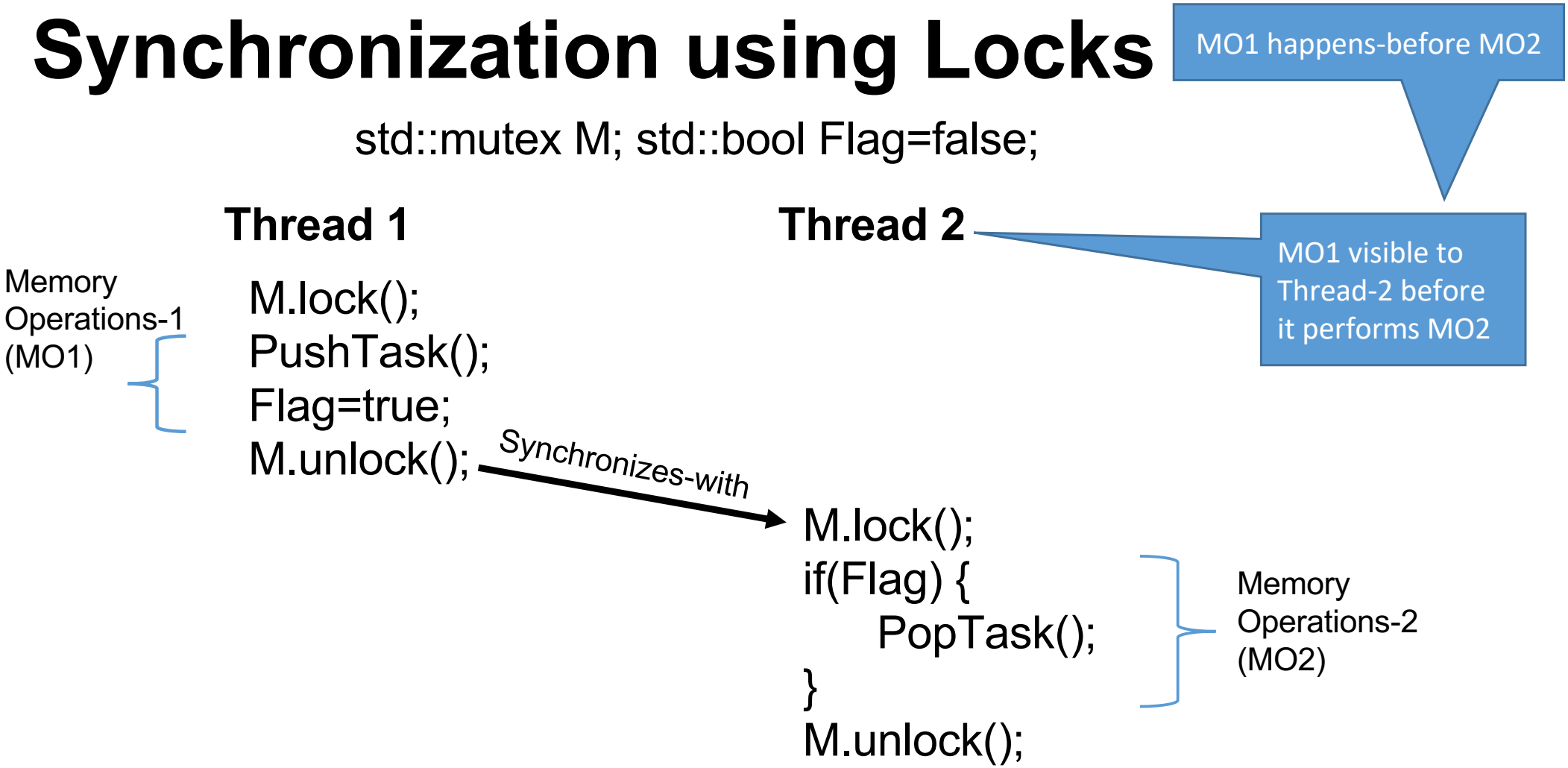
Achieving synchronization with some extra effort (slightly lower productivity but, high performance and portability)

Synchronizes-with Relationship

- Synchronizes-with → Happens-before relationship



Synchronization using Locks



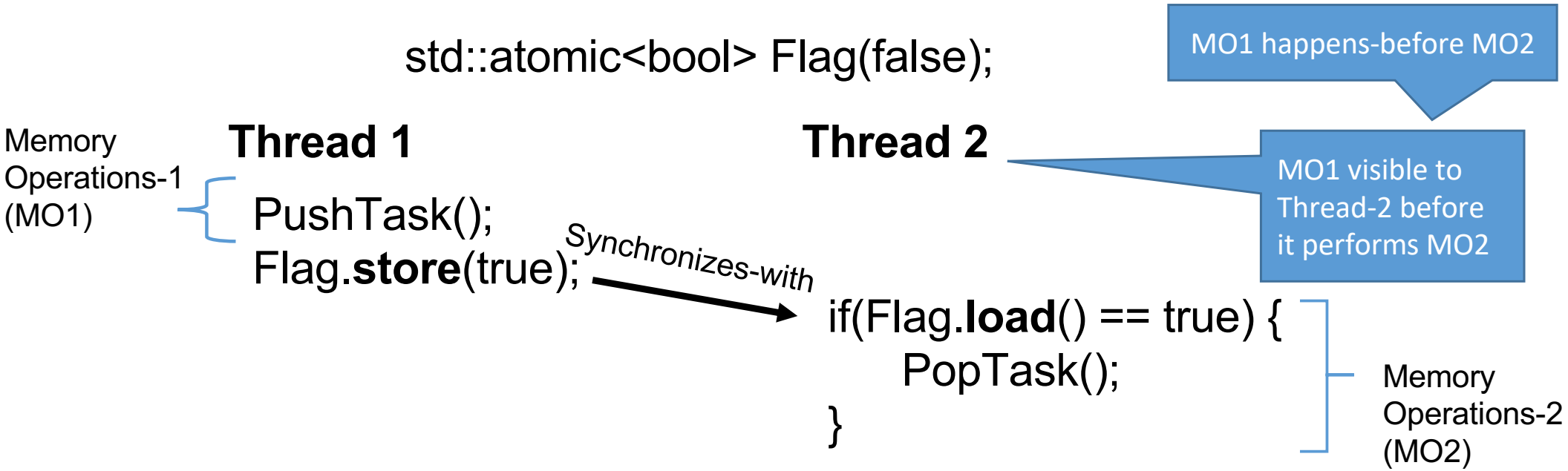
Synchronization using Locks: Summary

- Locks ensure that a read-modify-write operation on memory is carried out atomically
  - Matches with the switch based memory access analogy in sequential consistency
- Lock operations wait for all previous memory accesses to complete and for all buffered writes to drain to memory

Synchronization using std::atomic<>

- Data race free variable
  - We would use it when we want to achieve DRF on a single variable instead of a block of code
  - Provides inter thread synchronization
  - Sequential consistency by default
    - Similar to **volatile** in Java
    - Several other memory orderings allowed!
      - Unlike Java!

Synchronization using std::atomic<>

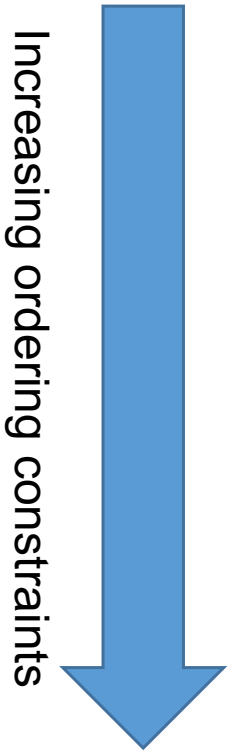


Sequential consistency by default !!

Benefits of Sequential Consistency?

- Guarantees **switch** based atomic access to each and every memory locations across two threads
  - Happens-before edge
- Can we relax this semantics?
  - Can we support multiple **switches**, where there is one **switch** for each atomic variable?
  - Can we allow reordering of ”other” variables (atomic or not) accessed before or after a single atomic variable?
    - Stores to other variables can propagate between cores with unpredictable delays (but not for a single atomic variable)

Memory Orderings with std::atomic<>

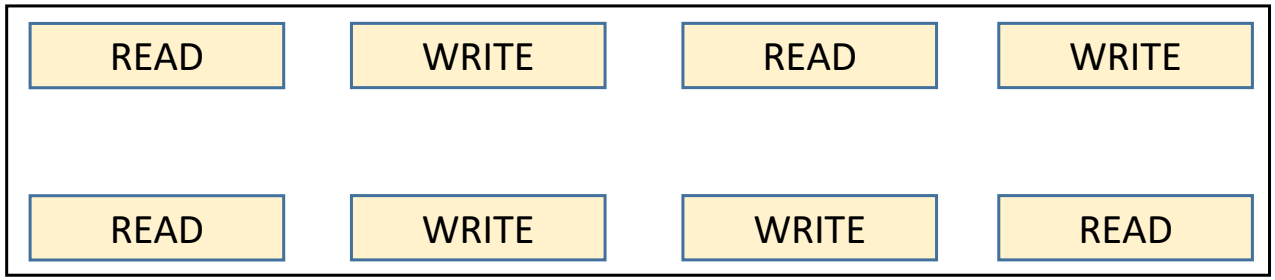


- Relaxed ordering
  - memory_order_relaxed
- Acquire-Release ordering
  - memory_order_acquire
  - memory_order_release
  - memory_order_acq_rel
  - *memory_order_consume*
- Sequential consistency ordering
  - memory_order_seq_cst
    - Default behavior

1. Helps in writing platform independent programs
2. Don't use memory fences yourself, but let the compiler do the job for you
3. Helps in understanding the exact intention of the programmer (improves readability)

We already saw its effect in the previous slide

Recall: Relaxed Memory Model



- All possible reordering of operations over two **different** memory locations inside a thread, out of which one is an atomic operation
- Memory operations performed by the same thread on the **same** memory location are not reordered with respect to the modification order

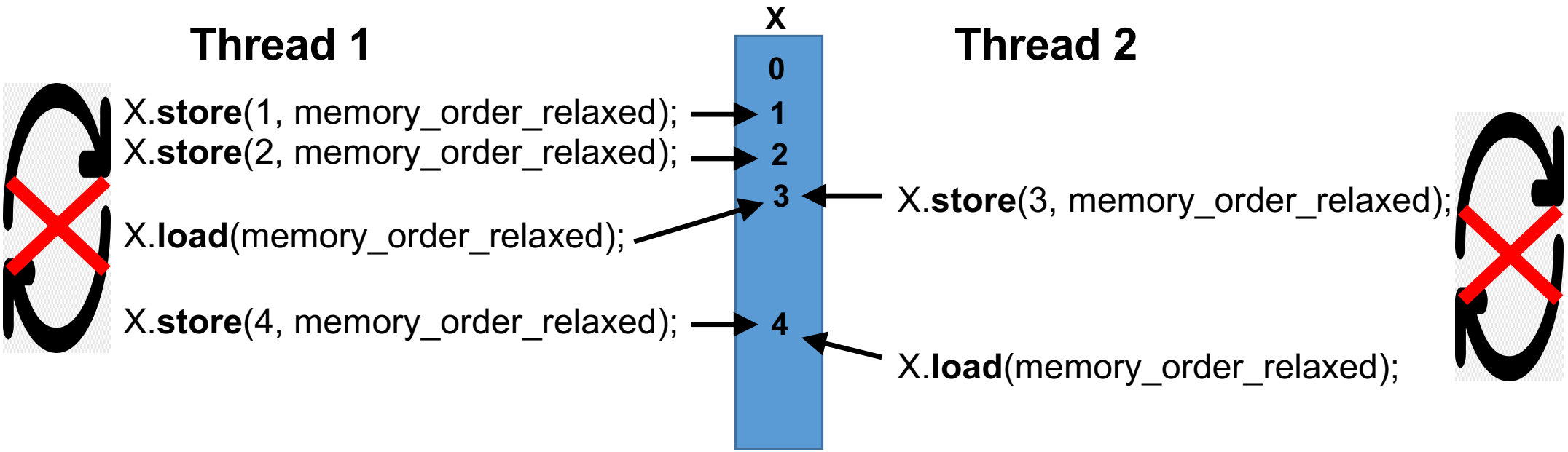
C++ Memory Order Relaxed

- **Rule-1**: There can never be any data race while performing read/write to a **single** atomic<>**A** var across multiple cores
  - Multiple accesses to same variable **A** can never be reordered

Memory Order Relaxed: Rule-1

Memory operations performed by the same thread on the **same** memory location are not reordered with respect to the modification order

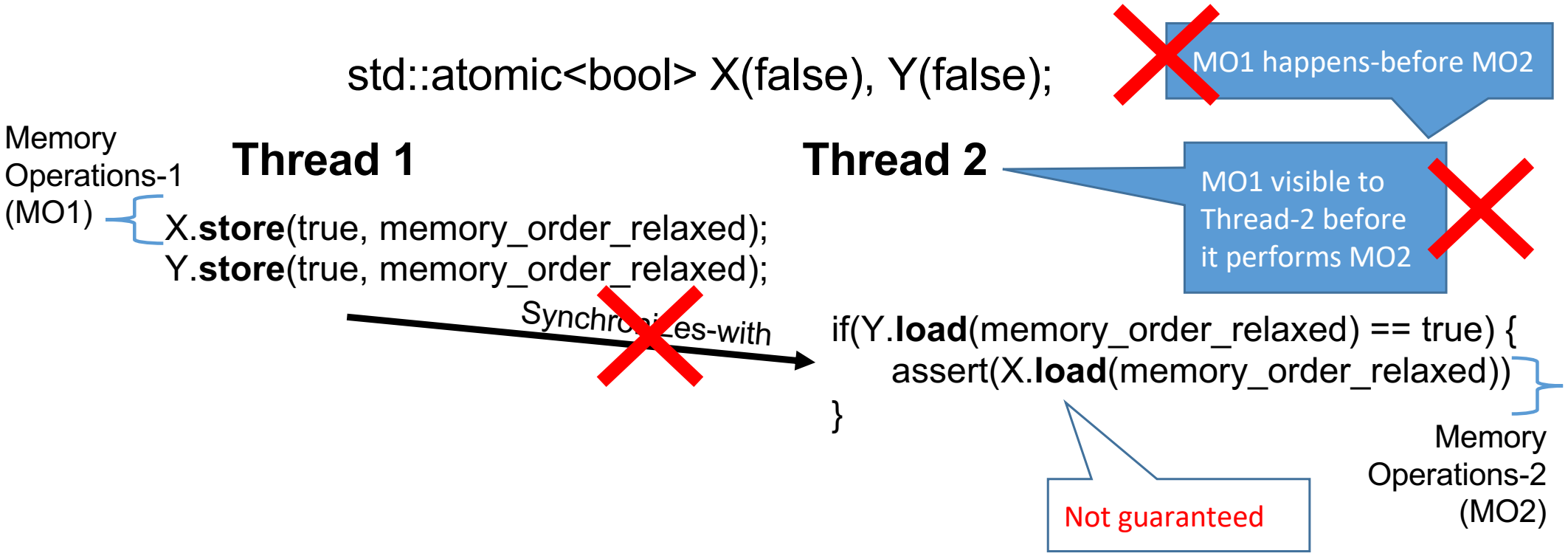
std::atomic<int> X(0);



C++ Memory Order Relaxed

- **Rule-1**: There can never be any data race while performing read/write to a **single** atomic<>**A** var across multiple cores
  - Multiple accesses to same variable **A** can never be reordered
- **Rule-2**: However, no guarantees of happens-before edge across accesses to **A** over two different threads
  - Operation is atomic only on atomic variable **A**
  - **A** can be reordered with read/write to any other variables (atomic or not) above or below it over a core
    - After accessing **A** on Core-1, Core-2 cannot judge if its safe to access other variables (atomic or not) that appeared before **A**'s access on Core-1

Memory Order Relaxed: Rule-2



Memory Order Relaxed: Rule-2

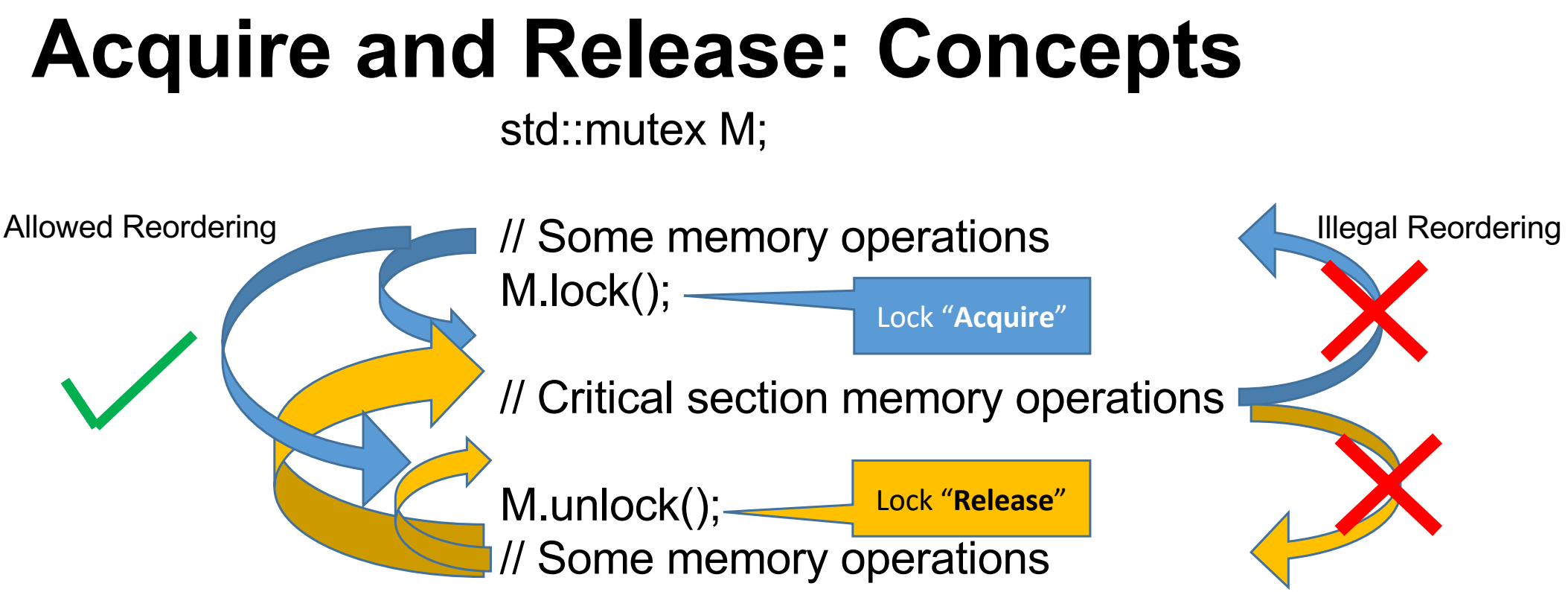
```
std::atomic<bool> X(false), Y(false);
```

Thread 1	Thread 2
X.**store**(true, memory_order_relaxed);	
Y.**store**(true, memory_order_relaxed);	
~~Synchronizes-with~~	if(Y.**load**(memory_order_relaxed) == true) {
	assert(X.**load**(memory_order_relaxed))
	}

All possible reordering of operations over two **different** memory locations inside a thread, out of which one is an atomic operation

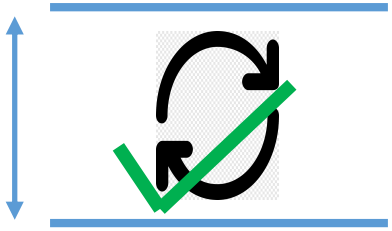
Not guaranteed

Acquire and Release: Concepts



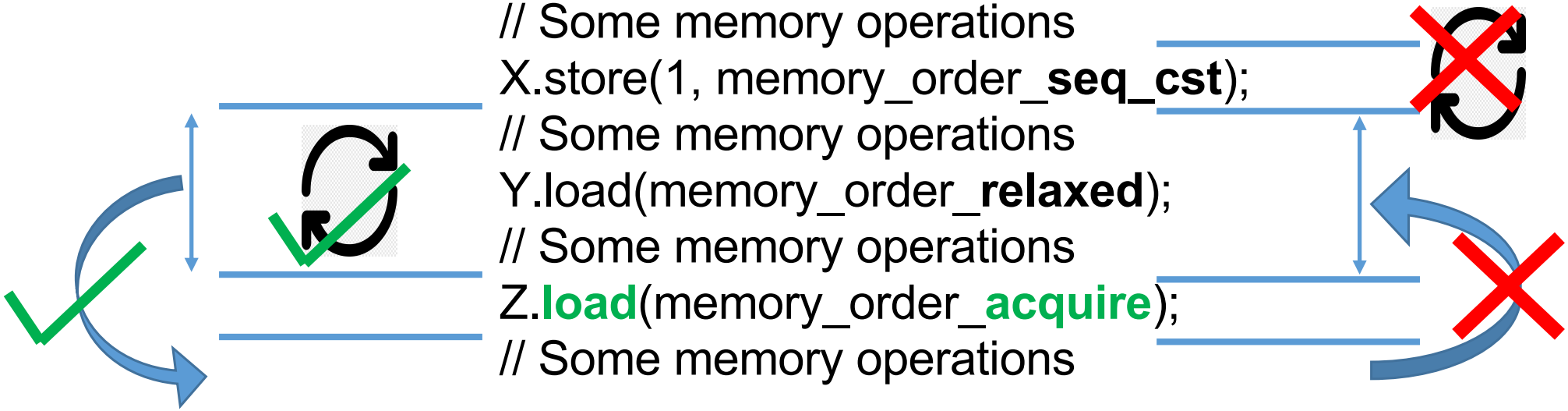
Memory Order Acquire/Release: Concepts

```
std::atomic<int> X, Y, Z;

// Some memory operations
X.store(1, memory_order_seq_cst);
// Some memory operations
Y.load(memory_order_relaxed);
// Some memory operations
```



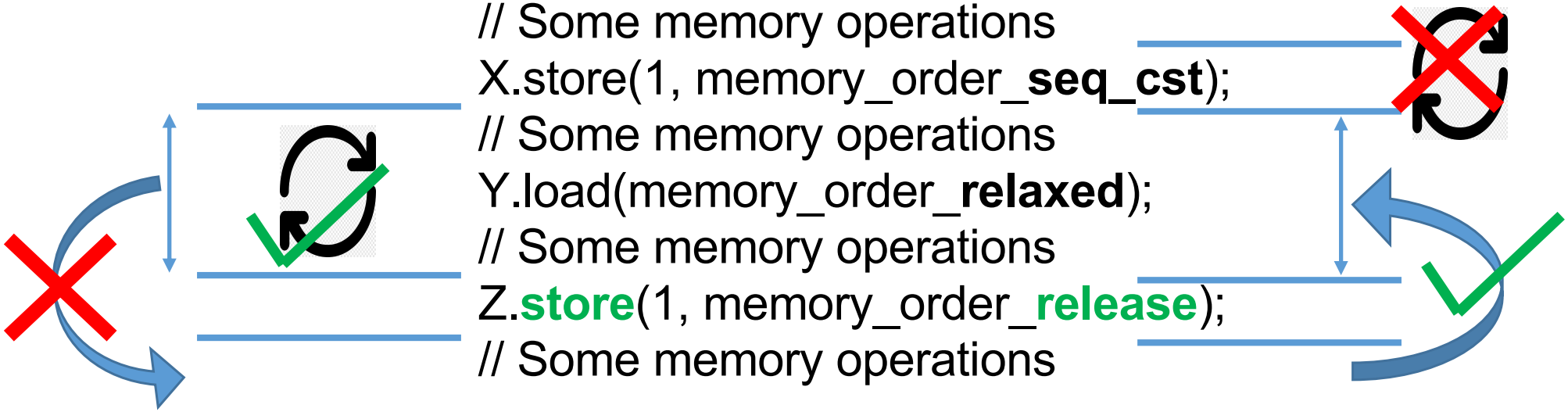
Memory Order Acquire/Release: Concepts

```
std::atomic<int> X, Y, Z;

// Some memory operations
X.store(1, memory_order_seq_cst);
// Some memory operations
Y.load(memory_order_relaxed);
// Some memory operations
Z.load(memory_order_acquire);
// Some memory operations
```



Memory Order Acquire/Release: Concepts

```
std::atomic<int> X, Y, Z;

// Some memory operations
X.store(1, memory_order_seq_cst);
// Some memory operations
Y.load(memory_order_relaxed);
// Some memory operations
Z.store(1, memory_order_release);
// Some memory operations
```



Memory Order Acquire/Release: Example

```
std::atomic<bool> A(false);
int non_atomic=0;
```

Thread 1

```
non_atomic =10 // Memory Operations MO1
A.store(true, memory_order_release);
```

Synchronizes-with

Thread 2

```
if(A.load(memory_order_acquire) == true) {
    // Memory Operations MO2
    assert(non_atomic == 10)
}
```

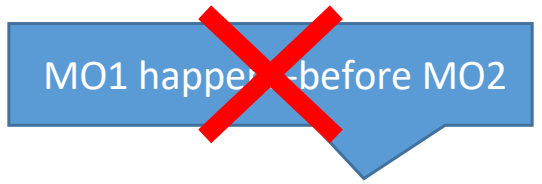
MO1 happens-before MO2

MO1 visible to Thread-2 before it performs MO2

Yes, this example looks same as in sequential consistency order

Memory Order Acquire/Release: Example

```
std::atomic<bool> A(false), B(false);
int non_atomic=0;
```



Thread 1

```
non_atomic =10 // Memory Operations MO1
A.store(true, memory_order_release);
```

Synchronizes-with

Thread 2

```
if(B.load(memory_order_acquire) == true) {
  // Memory Operations MO2
   assert(non_atomic == 10)
}
```

NOT same as in sequential consistency order

Acquire/Release ensures synchronization between threads that are storing and loading the same atomic object (also called as half-synchronization)

Memory Order Acquire/Release: Example

```
std::atomic<bool> A(false), B(false);
int non_atomic = 0;
```

Thread 1

```
non_atomic = 10;
A.store(true, memory_order_release);
```

Synchronizes-with

Thread 2

```
if(A.load(memory_order_acquire) == true) {
    B.store(true, memory_order_release);
}
```

Synchronizes-with

Thread 3

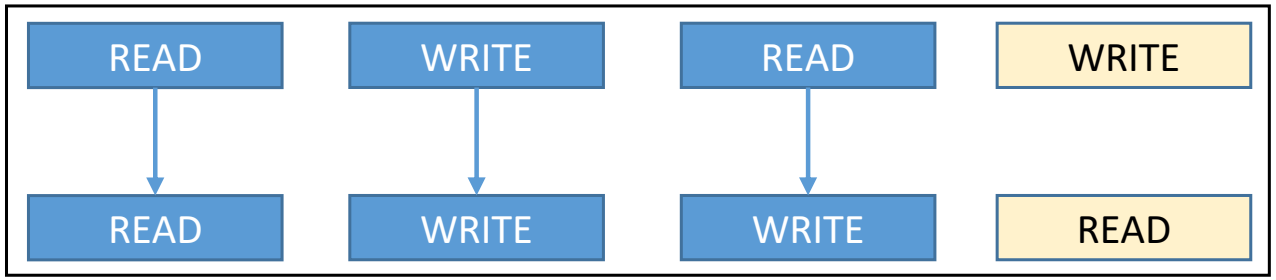
```
if(B.load(memory_order_acquire) == true) {
    assert(non_atomic == 10);
}
```

Acquire/Release ensures synchronization between threads that are storing and loading the same atomic object (also called as half-synchronization)

Achieving C++ Memory Orders on x86

Recall:

- x86-TSO memory consistency model



- **Rule-4b**: Reads CAN be reordered with earlier Writes to **DIFFERENT** memory locations (over same core)

- x86 CPUs have FIFO store buffers

Hence, if a write in store buffer of Core-1 is now visible to Core-2, then all previous writes from Core-1 are also now visible to Core-2

Achieving C++ Memory Orders on x86

Atomic Operation	Compiler Reordering	x86 implementation
A.store(1, memory_order_relaxed)	**Yes completely**	MOV (into memory)
A.load(memory_order_relaxed)	**Yes completely**	MOV (from memory)
A.store(1, memory_order_release)	Only as much permissible (Slide # 18)	MOV (into memory)
A.load(memory_order_acquire)	Only as much permissible (Slide # 18)	MOV (from memory)
A.store(1, memory_order_seq_cst)	**Not at all**	MOV (into memory), **MFENCE**
A.load(memory_order_seq_cst)	**Not at all**	MOV (from memory)

Source: https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

Reference Materials

- https://preshing.com/20120913/acquire-and-release-semantics/
- Atomic weapons – Herb Sutter
  - https://www.youtube.com/watch?v=A8eCGOqgvH4

Next Lecture (L #11)

- Non uniform memory access architecture