Lecture 15: Language Memory Model

Vivek Kumar

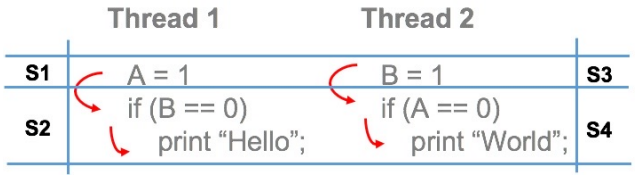
Computer Science and Engineering

IIIT Delhi

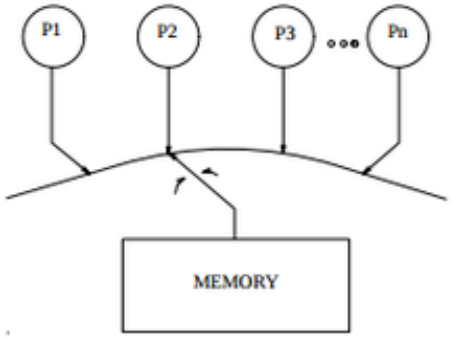
vivekk@iiitd.ac.in

Last Lecture (Recap)

- Memory latency continues to limit the performance of multicore processors
  - Several optimizations inside processors for hiding the load/store latency
    - As a side effect of these optimizations, load/store inside a program could be reordered, and hence may not happen in the source code order as expected by programmer

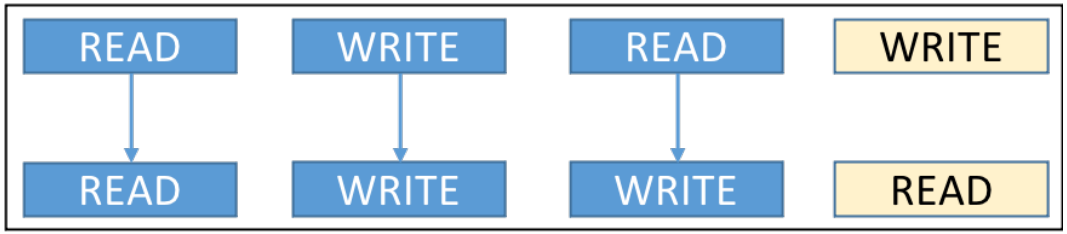


- Memory consistency model defines a set of rules for valid set of reordering of **two different memory accesses**
  - Both compiler and processor can perform reordering

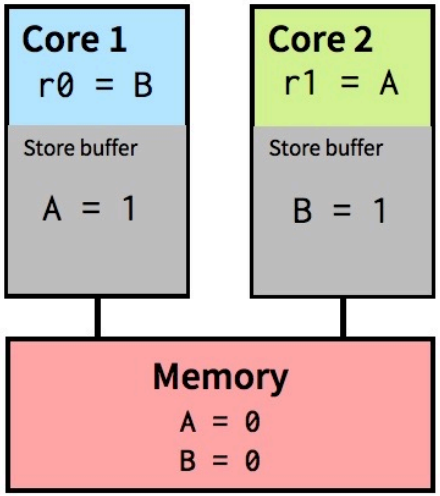


- Sequential consistency is the most primitive form of memory consistency that basically says memory access to any location always happens atomically, and the effect is visible to each and every core
  - Modern programming languages supports sequential consistency only for code block within a mutex lock/unlock operation (Data Race Free)

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)
  - Write-Read reordering is only allowed
  - Due to presence of store buffers
- Store buffer
  - Temporarily holds write before they are committed to the cache
  - Size enough for ~50 stores on Intel processor (Skylake)

Today's Class

- Mutex lock v/s atomic variable
- C++ memory model
- Lock free work-stealing deque

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++

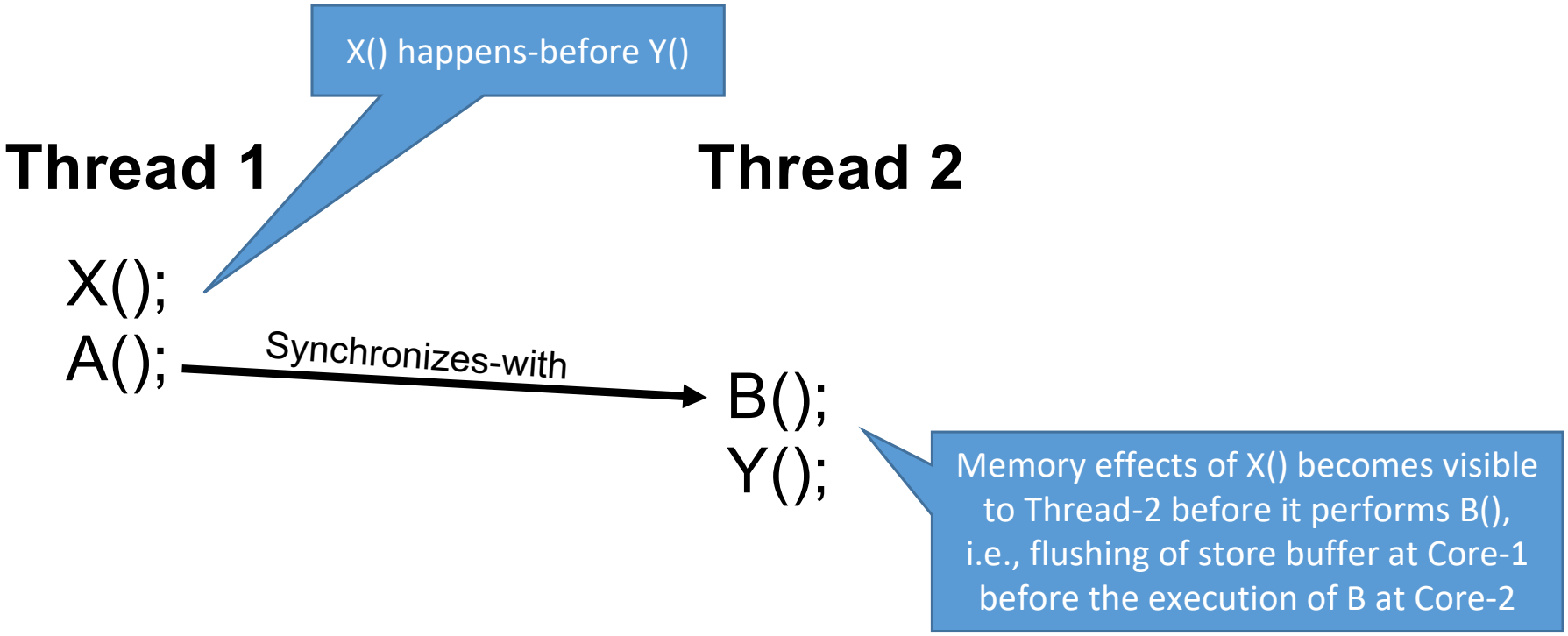
- Synchronization operations
  - Lock based operations
    - Mutex lock/unlock
  - Lock-free (A.K.A. atomic) operations
    - Atomic load (read)
    - Atomic store (write)
    - Atomic Read-Modify-Write (RMW), i.e., compare and swap on x86 platforms
      - E.g., std::atomic<int>::fetch_add
- Non-synchronization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

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

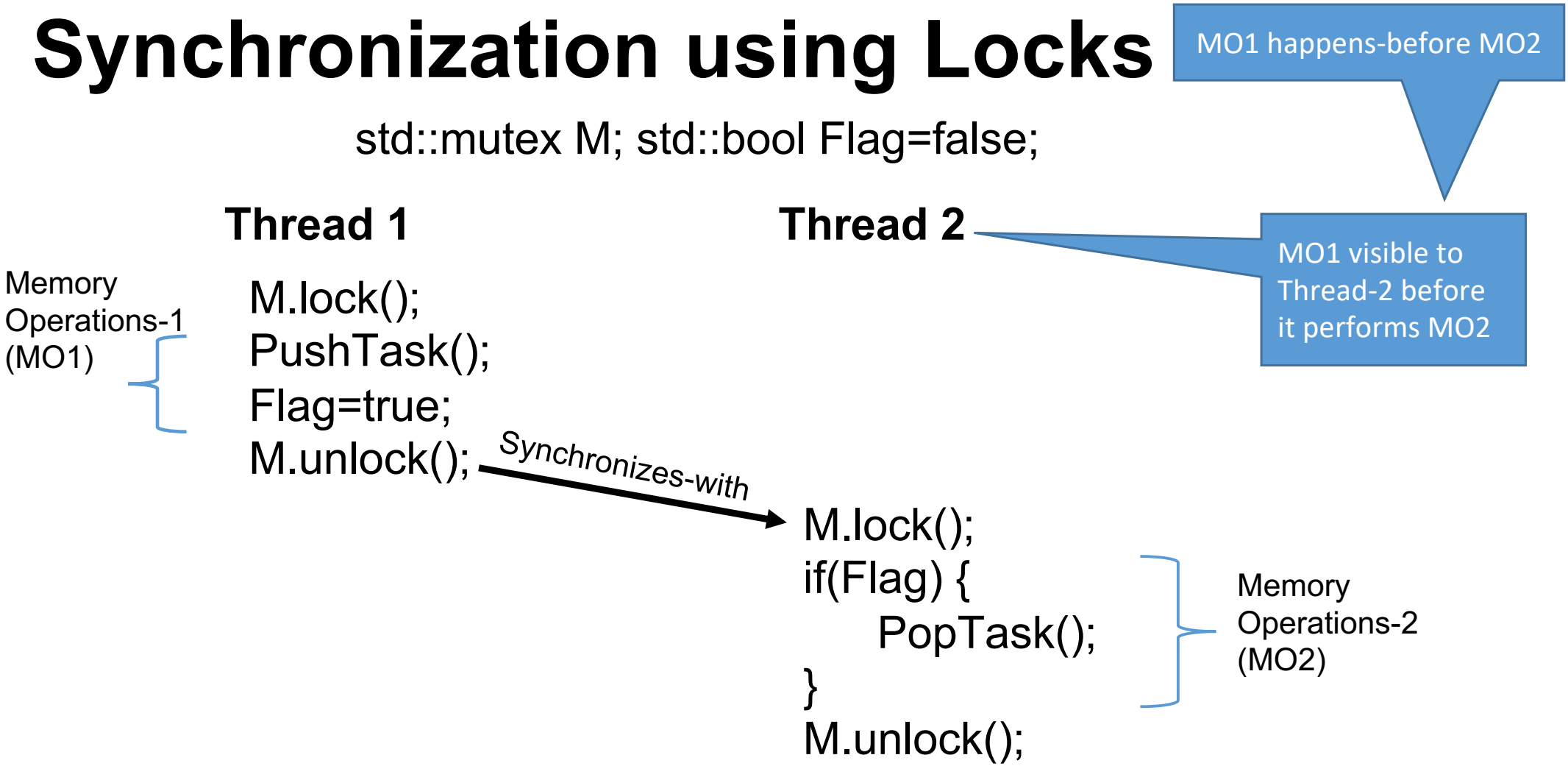
Achieving synchronization with some extra effort (slightly lower productivity but, high performance)

Synchronizes-with Relationship

- Synchronizes-with → Happens-before relationship



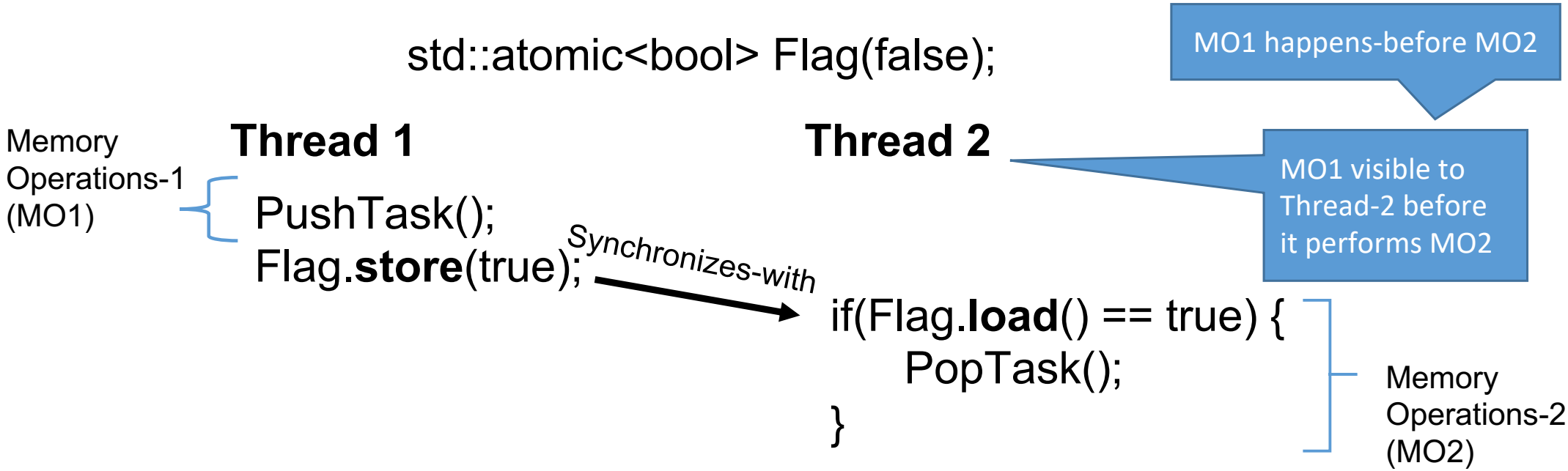
Synchronization using Locks



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<>



Sequential consistency by default !!

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!

Mutex v/s Atomic

```
std::mutex M; std::bool Flag=false;

Thread 1                          Thread 2

M.lock();
PushTask();
Flag=true;
M.unlock();  --Synchronizes-with-->  M.lock();
                                     if(Flag) {
                                         PopTask();
                                     }
                                     M.unlock();
```

```
std::atomic<bool> Flag(false);

Thread 1                          Thread 2

PushTask();
Flag.store(true);  --Synchronizes-with-->  if(Flag.load() == true) {
                                               PopTask();
                                           }
```

- Mutex
  - If lock is already acquired by T1, OS will context switch T2 if it attempts to acquire the same lock
    - System call used for wait/sleep (transition from user-space to kernel-space)
    - Extremely heavy weight operation if the critical section is just about updating a shared variable
  - Overheads significant when the contention increases (more threads competing)
  - Preferable to use when updating multiple shared variables in one go to ensure atomicity
- Atomic
  - Avoids the costly context switch
  - Waste CPU cycles, but its much cheaper than the context switch, as the spinning doesn't happen for long
    - The C.S. is just updating a Boolean variable

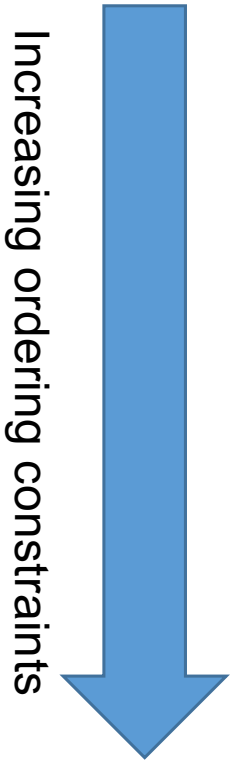
Benefits of Sequential Consistency?

- 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)

Today's Class

- Mutex lock v/s atomic variable
- C++ memory model
- Lock free work-stealing deque

Memory Orderings with std::atomic<>

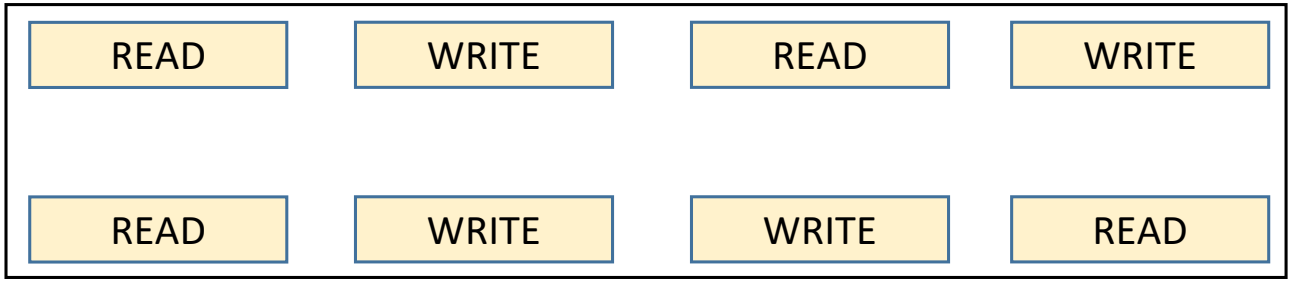


- Relaxed ordering
  - memory_order_relaxed
- Acquire-Release ordering
  - memory_order_acquire
  - memory_order_release
- Sequential consistency ordering
  - memory_order_seq_cst
    - Default behavior

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

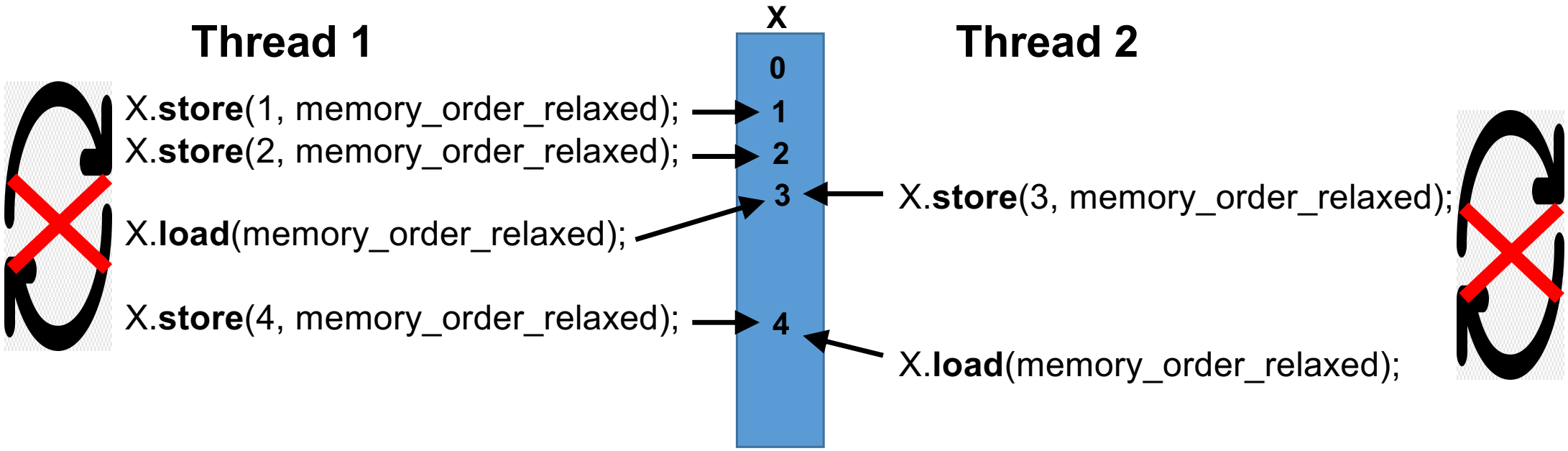
C++ Memory Order Relaxed

- **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

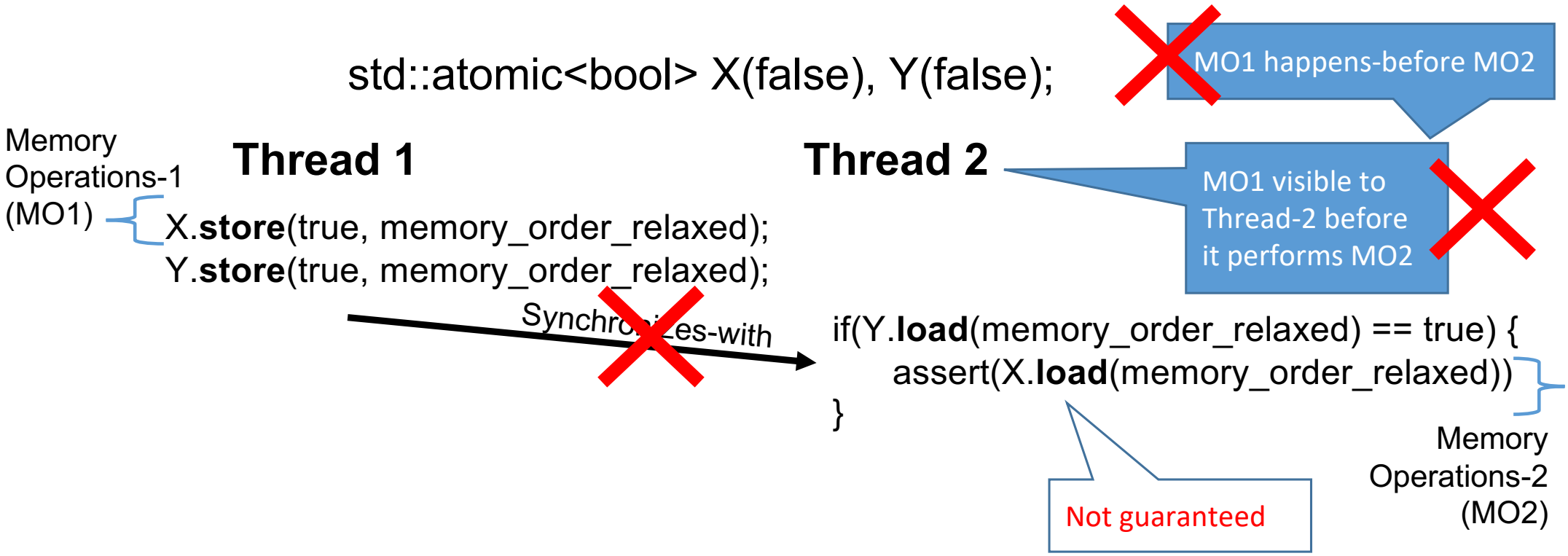
std::atomic<int> X(0);



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

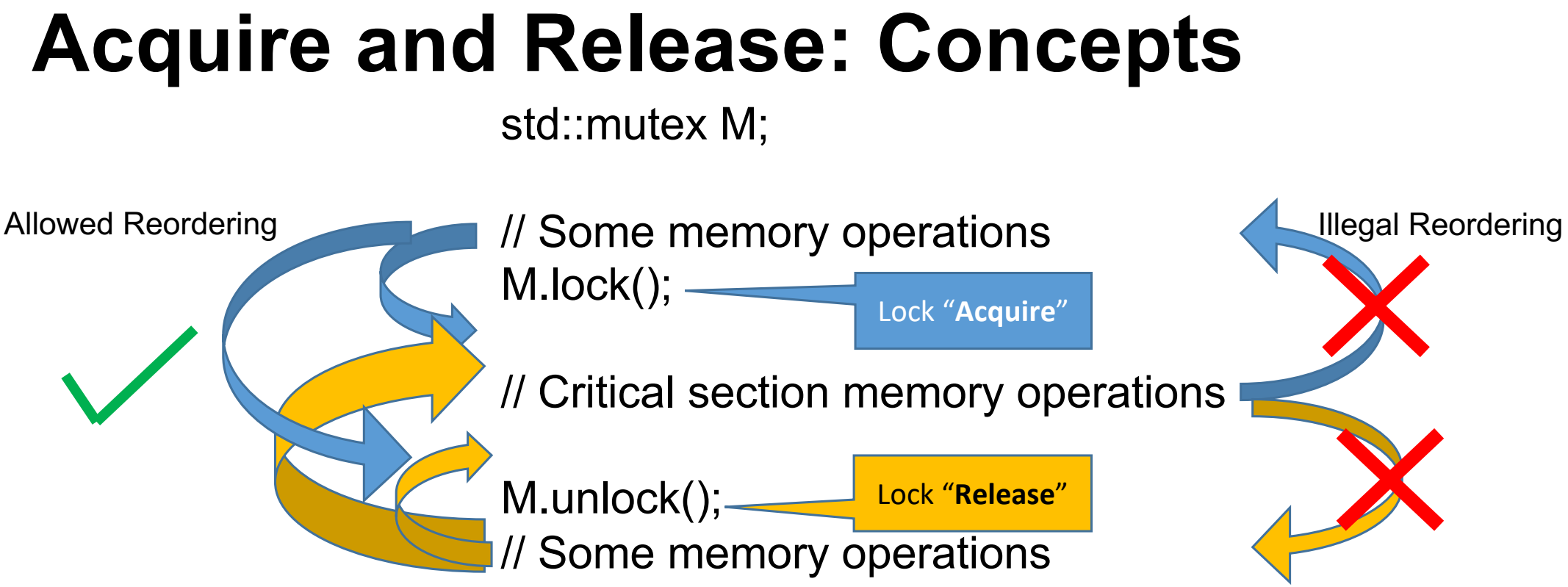
All possible reordering of operations over two **different** memory locations inside a thread, out of which one is an atomic operation

```
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))
	}

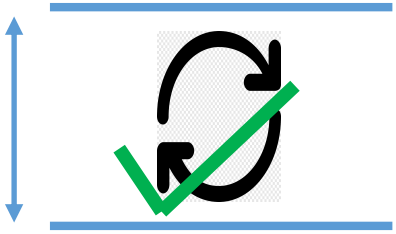
Not guaranteed

Acquire and Release: Concepts



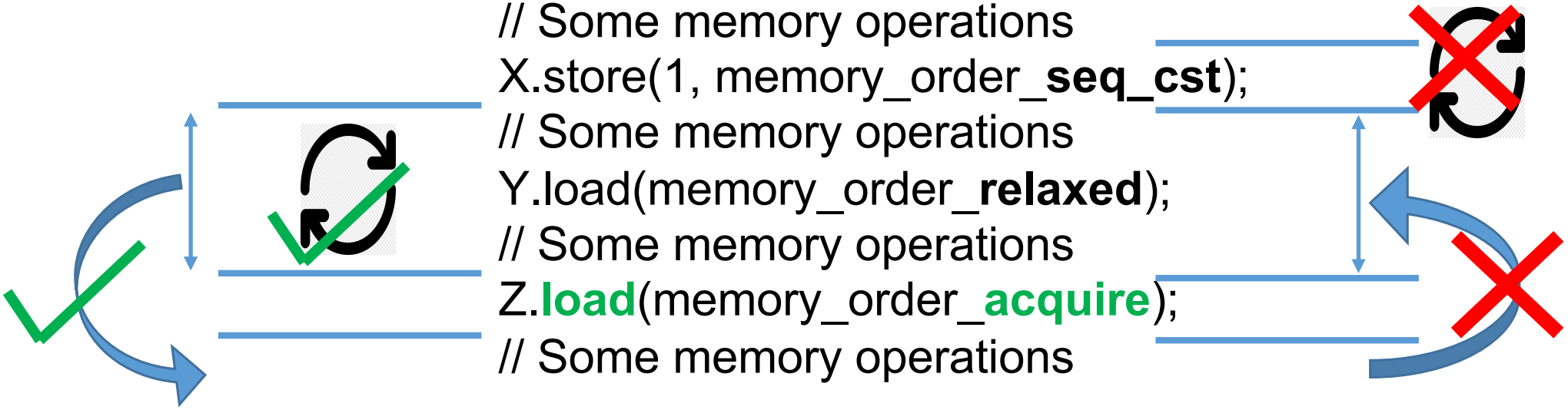
Memory Order Acquire/Release: Concepts

```
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
```



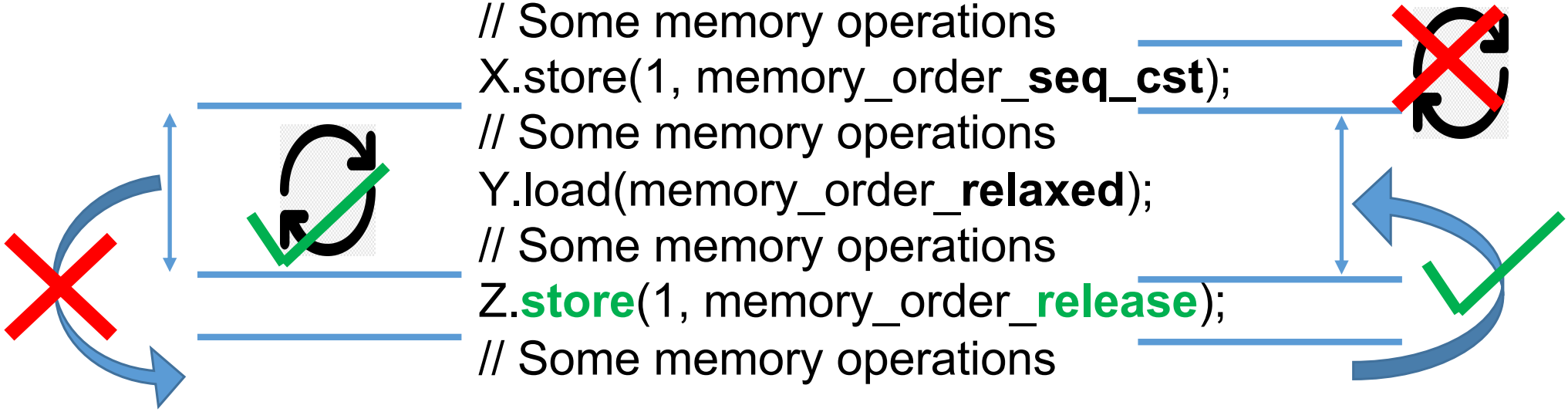
Memory Order Acquire/Release: Concepts

```
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)
}
```

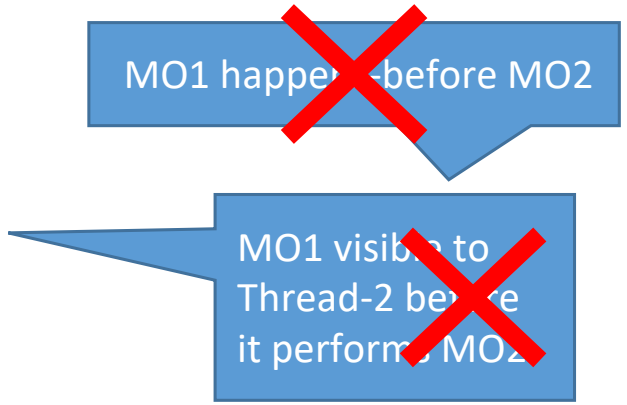
MO1 happens-before MO2

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)
}
```

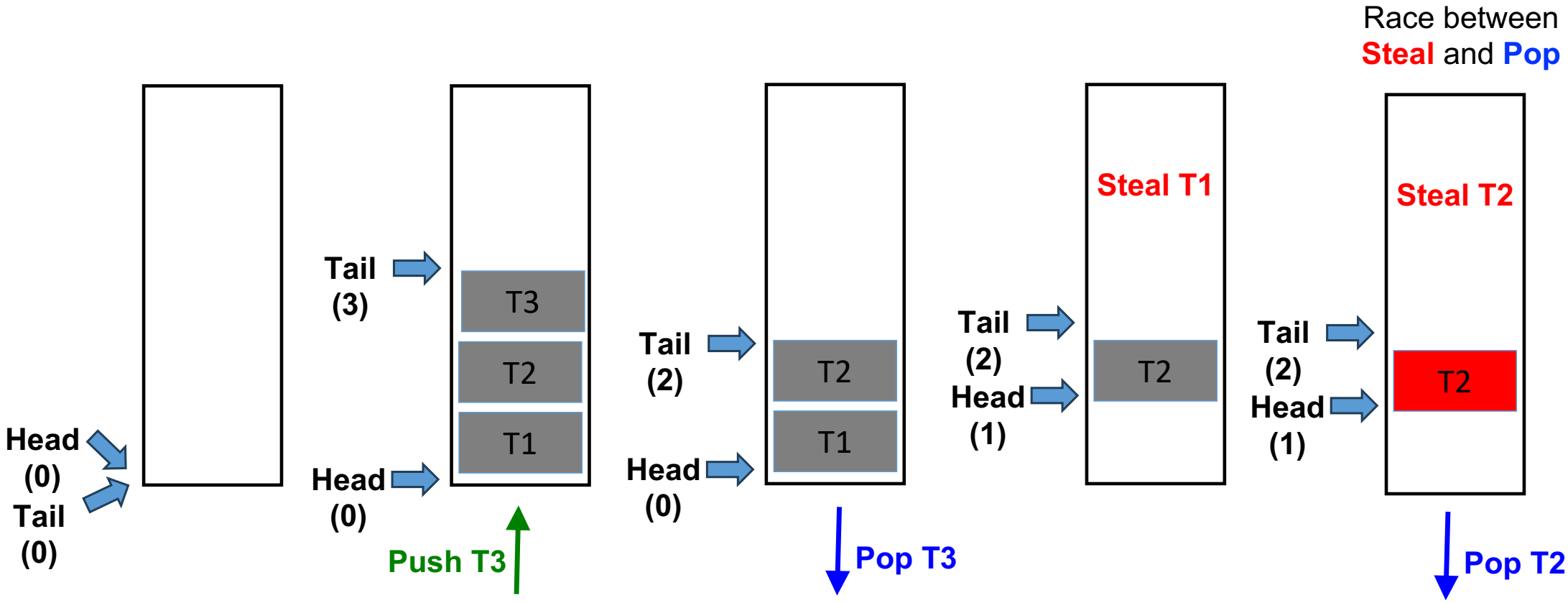
NOT same as in sequential consistency order

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

Today's Class

- Mutex lock v/s atomic variable
- C++ memory model
- Lock free work-stealing deque

Lock Free Work-Stealing Deque (Chase-Lev)

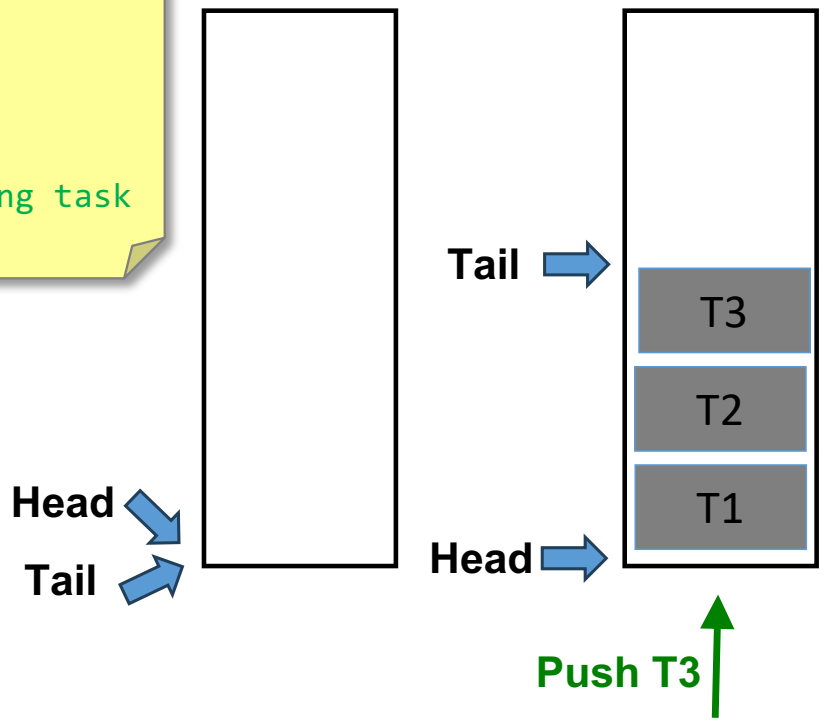


The push Operation

```
void push(deque_t* deque, task_t* task) {
  int head = deque->head.load();
  int tail = deque->tail.load();
  if((tail – head) == SIZE) throw_error(“Deque Full”);
  deque->buffer[tail % SIZE] = task; // First store the task
  deque->tail.store(tail + 1);       // Increment after storing task
}
```

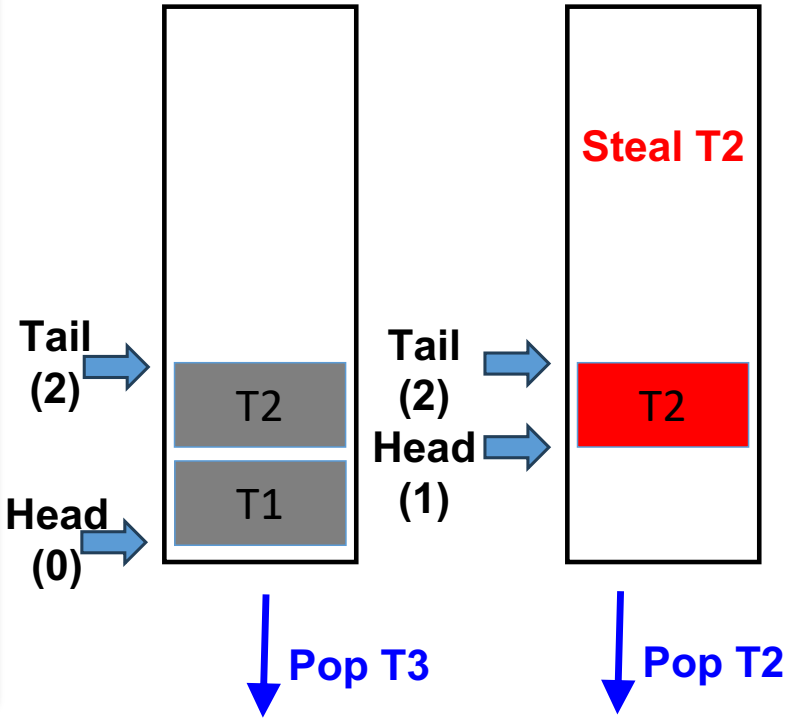
std::atomic<int> tail, head;

std::memory_order_seq_cst
by default everywhere



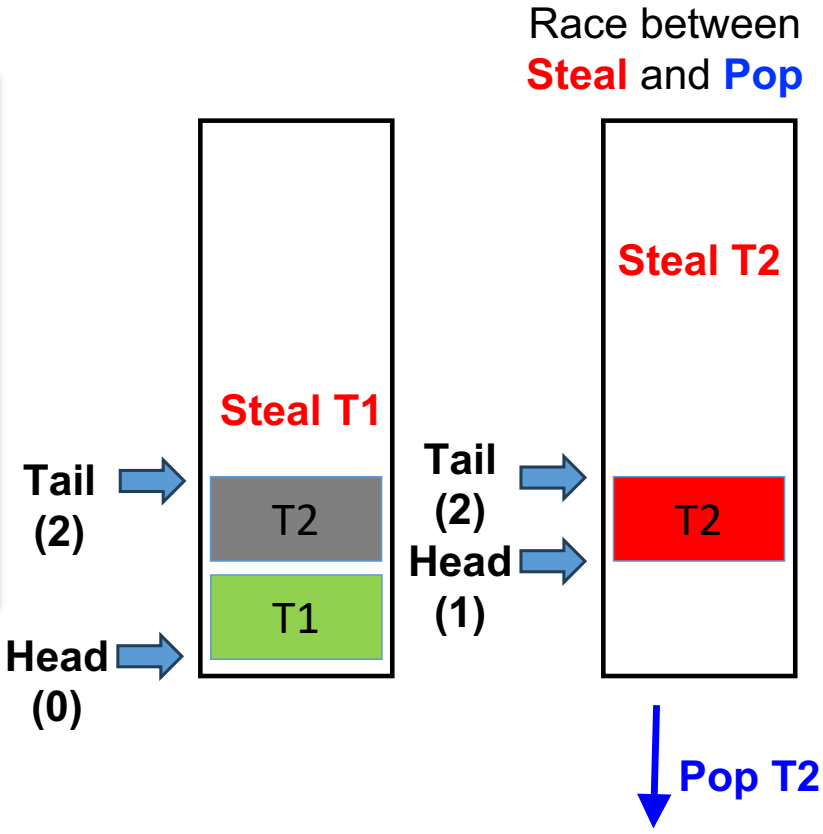
The pop Operation

```
task_t* pop(deque_t* deque) {
  int head = deque->head.load();
  int tail = deque->tail.load();
  deque->tail.store(tail - 1);
  task_t* task = deque->buffer[tail];
  if(tail < head) { // Deque just got empty
    deque->tail.store(head) // Both head and tail point to same slot
    task = NULL;
  }
  else if(tail > head) { // Sufficient tasks available on deque
    // Do nothing. We are all set
  } else { // Only one task remaining - race with thief
    if(!deque->head.compare_and_exchange_strong(head, head + 1)) {
      task = NULL; // Thief won the race
    }
    deque->tail.store(tail + 1); // increment tail to match head
  }
  return task;
}
```



The steal Operation

```
task_t* steal(deque_t* deque) {
  int head = deque->head.load();
  int tail = deque->tail.load();
  if((tail <= head) {  // No task available
    return NULL;
  } else {           // Tasks are available (count doesn't matter)
    task_t* task = deque->buffer(head % SIZE];
    if(!deque->head.compare_and_exchange_strong(head, head + 1)) {
      task = NULL; // Race lost with either victim or other thief
    }
    return task;
  }
}
```



Using release–acquire Ordering

- Loads from “tail” follows **relaxed** ordering at victim
  - As only updated locally by victim
- Loads from “head” follows **relaxed** ordering at both victim and thief
- Stores to “tail” from victim (**release**) should synchronize with load from “tail” at thief (**acquire**)
  - Thief should be aware of updated value of “tail” at steal
- Stores on “head” at both victim and thief must use CAS operation (**acquire_release**)

Reference Materials

- https://preshing.com/20120913/acquire-and-release-semantics/
- Atomic weapons – Herb Sutter
  - https://www.youtube.com/watch?v=A8eCGOqgvH4
- https://www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-dequeue.pdf

Next Lecture

- Parallel programming using SIMD vector units