Memory order

Pieter P

References

- cppreference: Memory model
- cppreference: std::memory_order
- Herb Sutter: atomic<> Weapons (Channel 9 links are broken, but videos can still be found on YouTube)
- Jeff Preshing: An Introduction to Lock-Free Programming
- Jeff Preshing: Memory Barriers Are Like Source Control Operations

Load-acquire ▼

Store-release ▲

SC Load-acquire ▼

SC Store-release ▲

Memory orders for C++ atomics

The C++ standard defines six constants to specify memory ordering: The main two orderings are `memory_order_acquire` and `memory_order_release`. The default ordering is `memory_order_seq_cst`, which implies `memory_order_acquire` for load operations and `memory_order_release` for store operations, and additionally creates a global, sequentially consistent order of operations on atomic variables. The combined `memory_order_acq_rel` ordering applies only to read-modify-write operations, and implies `memory_order_acquire` for the read and `memory_order_release` for the write. On some architectures, it can in theory be beneficial to use `memory_order_consume`, which is a weaker version of `memory_order_acquire`, but at the time of writing, no compiler implements it and the specifics are still being discussed, so we won't cover it here. Finally, `memory_order_relaxed` does not impose any ordering constraints, so it won't be discussed in this section about memory order.

`memory_order_acquire` ▼

No reads or writes in the current thread can be reordered before this load.

Applies to load operations.

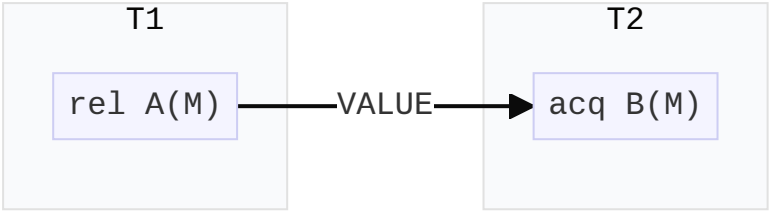
`memory_order_release` ▲

No reads or writes in the current thread can be reordered after this store.

Applies to store operations.

Acquire-release

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.

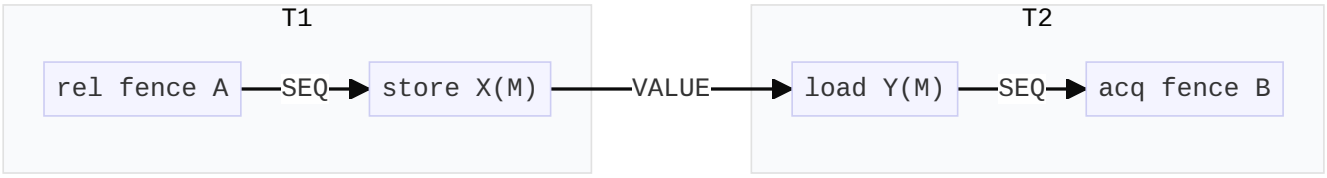


C++ standard: [atomics.order]

Fences

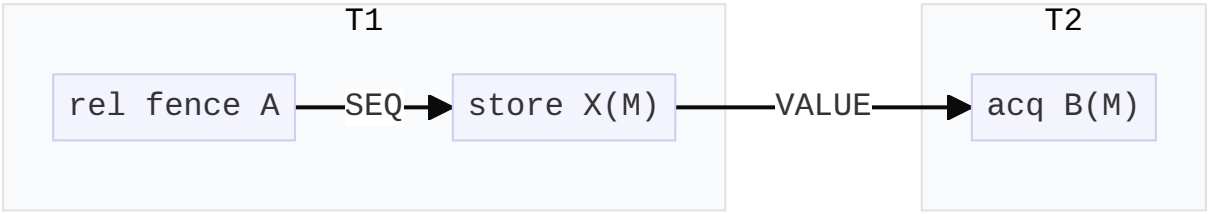
Fence–fence

A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.



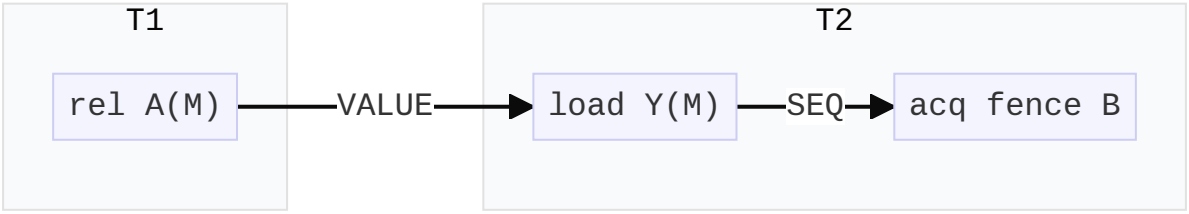
Fence–atomic

A release fence A synchronizes with an atomic operation B that performs an acquire operation on an atomic object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.



Atomic–fence

An atomic operation A that is a release operation on an atomic object M synchronizes with an acquire fence B if there exists some atomic operation X on M such that X is sequenced before B and reads the value written by A or a value written by any side effect in the release sequence headed by A.

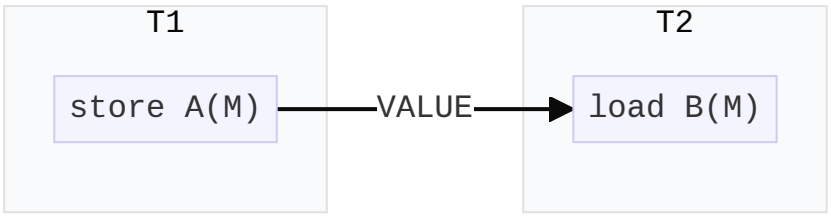


C++ standard: [atomics.fences]

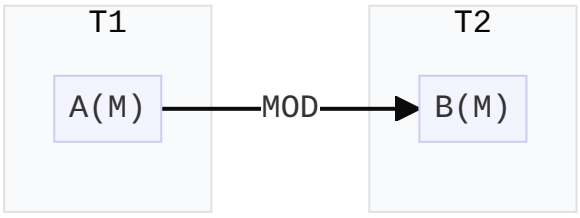
Coherence-ordered before

An atomic operation A on some atomic object M is coherence-ordered before another atomic operation B on M if

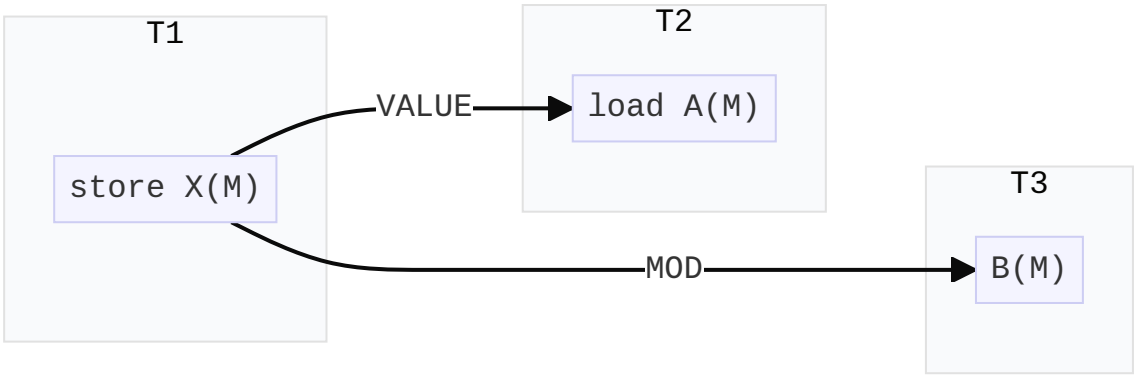
- A is a modification, and B reads the value stored by A, or



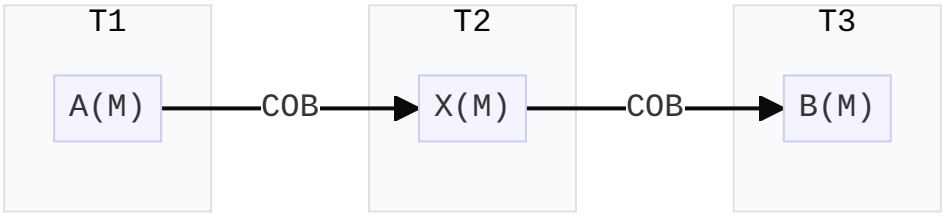
- A precedes B in the modification order of M, or



- A and B are not the same atomic read-modify-write operation, and there exists an atomic modification X of M such that A reads the value stored by X and X precedes B in the modification order of M, or



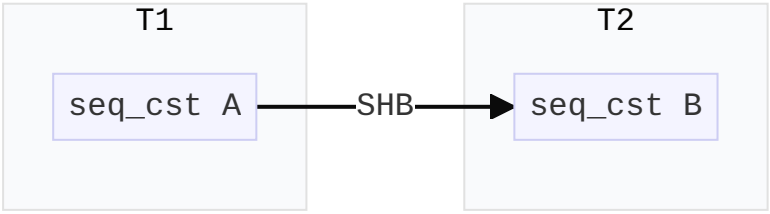
- there exists an atomic modification X of M such that A is coherence-ordered before X and X is coherence-ordered before B.



Total order of sequentially consistent operations

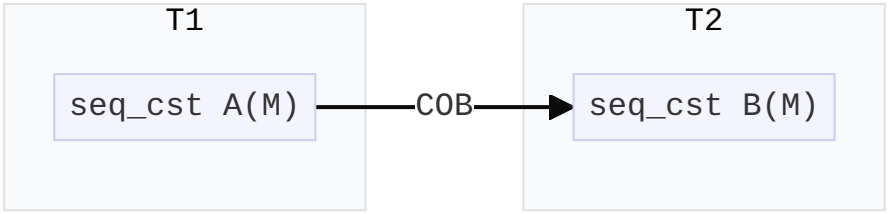
There is a single total order S on all `memory_order::seq_cst` operations, including fences, that satisfies the following constraints.

First, if A and B are `memory_order::seq_cst` operations and A strongly happens before B, then A precedes B in S.

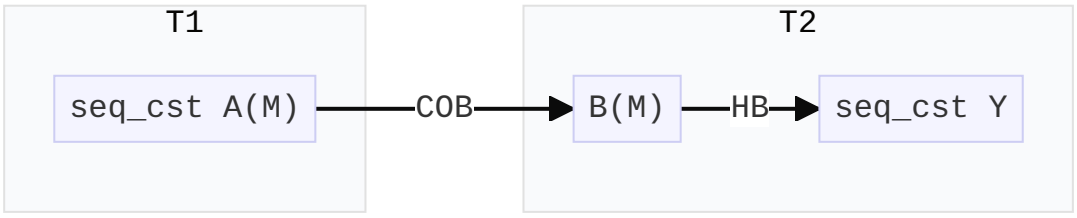


Second, for every pair of atomic operations A and B on an object M, where A is coherence-ordered before B, the following four conditions are required to be satisfied by S:

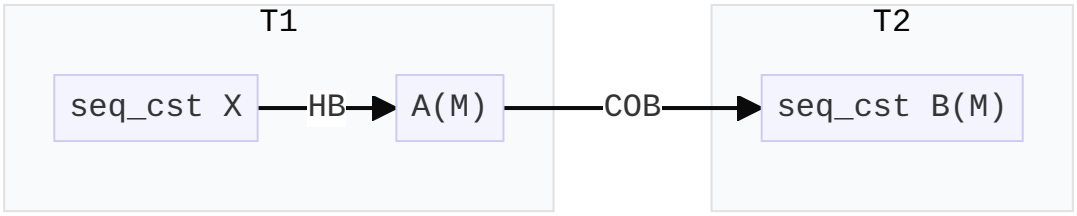
- if A and B are both `memory_order::seq_cst` operations, then A precedes B in S; and



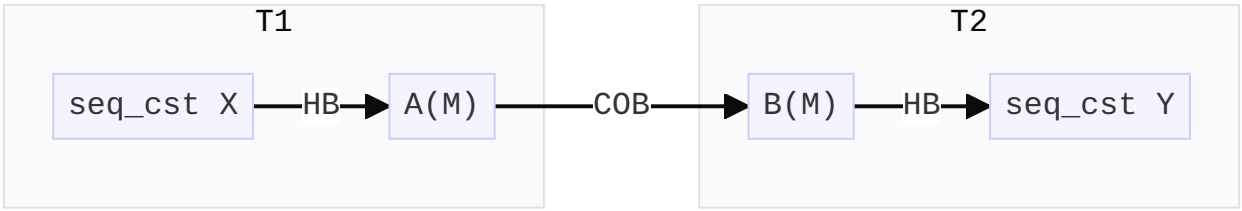
- if A is a `memory_order::seq_cst` operation and B happens before a `memory_order::seq_cst` fence Y, then A precedes Y in S; and



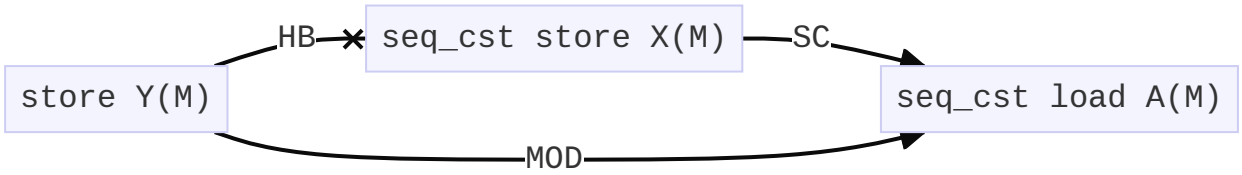
- if a `memory_order::seq_cst` fence X happens before A and B is a `memory_order::seq_cst` operation, then X precedes B in S; and



- if a `memory_order::seq_cst` fence X happens before A and B happens before a `memory_order::seq_cst` fence Y, then X precedes Y in S.



This definition ensures that S is consistent with the modification order of any atomic object M. It also ensures that a `memory_order::seq_cst` load A of M gets its value either from the last modification of M that precedes A in S or from some non-`memory_order::seq_cst` modification of M that does not happen before any modification of M that precedes A in S.



`memory_order::seq_cst` ensures sequential consistency only for a program that is free of data races and uses exclusively `memory_order::seq_cst` atomic operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In many cases, `memory_order::seq_cst` atomic operations are reorderable with respect to other atomic operations performed by the same thread.