Happens before

Pieter P

Definitions

This section quotes some definitions from the standard, adding diagrams for easier interpretation.

Sequenced before

Partial evaluation order within a single thread. This is not necessarily the order observed by other threads.

Most important rule: *"Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated."*

cppreference: Order of evaluation: Ordering
C++ standard: [intro.execution]

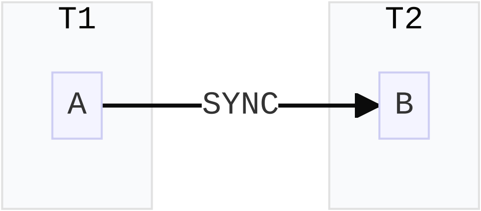
Synchronizes with

Certain library calls synchronize with other library calls performed by another thread. For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store.

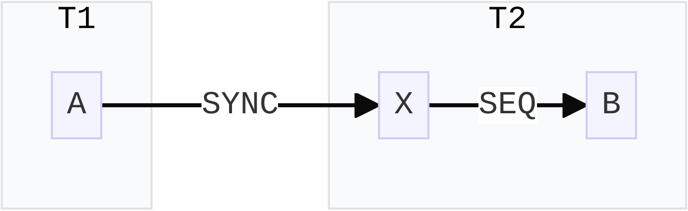
Inter-thread happens before

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

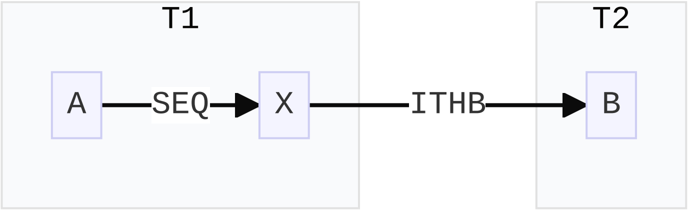
- *A* synchronizes with *B*, or



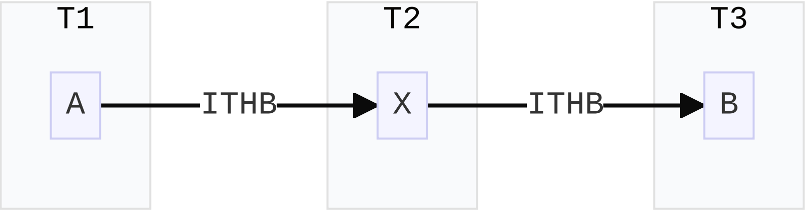
- *A* is dependency-ordered before *B*, or
- for some evaluation *X*
  - *A* synchronizes with *X* and *X* is sequenced before *B*, or



  - *A* is sequenced before *X* and *X* inter-thread happens before *B*, or



  - *A* inter-thread happens before *X* and *X* inter-thread happens before *B*.

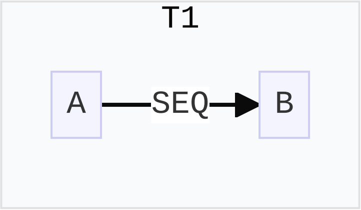


C++ standard: [intro.races]

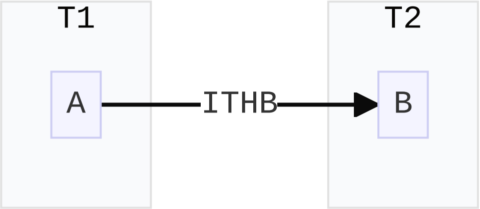
Happens before

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

- *A* is sequenced before *B*, or



- *A* inter-thread happens before *B*.

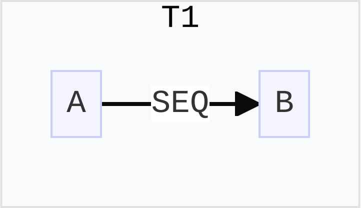


C++ standard: [intro.races]

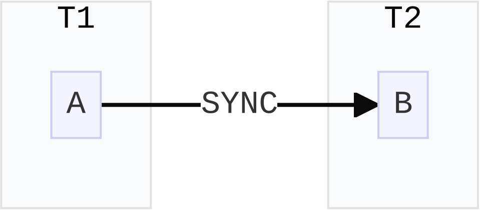
Simply happens before

An evaluation *A* simply happens before an evaluation *B* if either

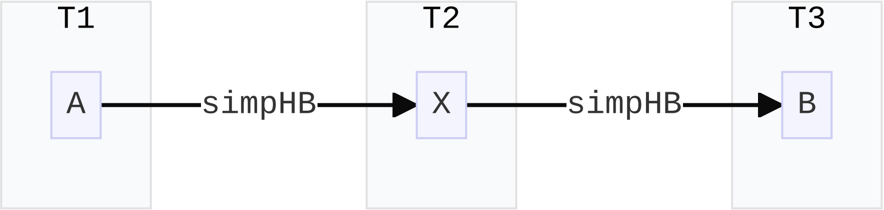
- *A* is sequenced before *B*, or



- *A* synchronizes with *B*, or



- *A* simply happens before *X* and *X* simply happens before *B*.



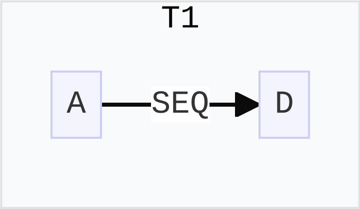
In the absence of consume operations, the happens before and simply happens before relations are identical.

C++ standard: [intro.races]

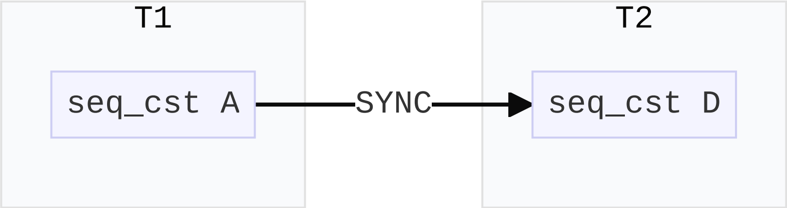
Strongly happens before

An evaluation *A* strongly happens before an evaluation *D* if, either

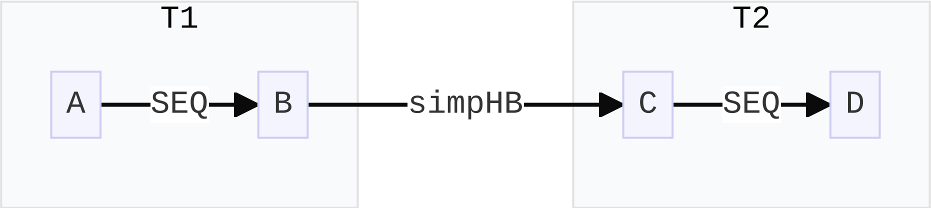
- *A* is sequenced before *D*, or



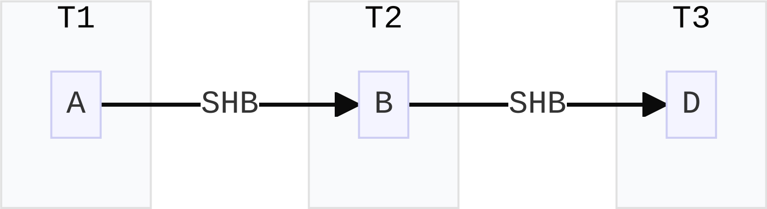
- *A* synchronizes with *D*, and both *A* and *D* are sequentially consistent atomic operations, or



- there are evaluations *B* and *C* such that *A* is sequenced before *B*, *B* simply happens before *C*, and *C* is sequenced before *D*, or



- there is an evaluation *B* such that *A* strongly happens before *B*, and *B* strongly happens before *D*.



Informally, if *A* strongly happens before *B*, then *A* appears to be evaluated before *B* in all contexts. Strongly happens before excludes consume operations.

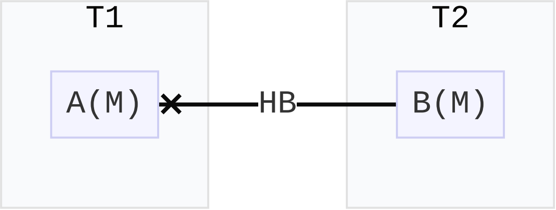
C++ standard: [intro.races]

Modification order

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

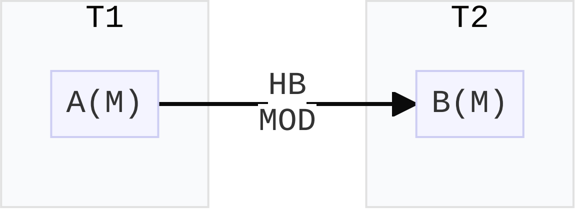
This order is usually guaranteed by the cache coherence of the hardware.
Note that there is no total order of modifications of *distinct* atomic objects.

The value of an atomic object *M*, as determined by evaluation B, shall be the value stored by some side effect *A* that modifies *M*, where *B* does not happen before *A*.



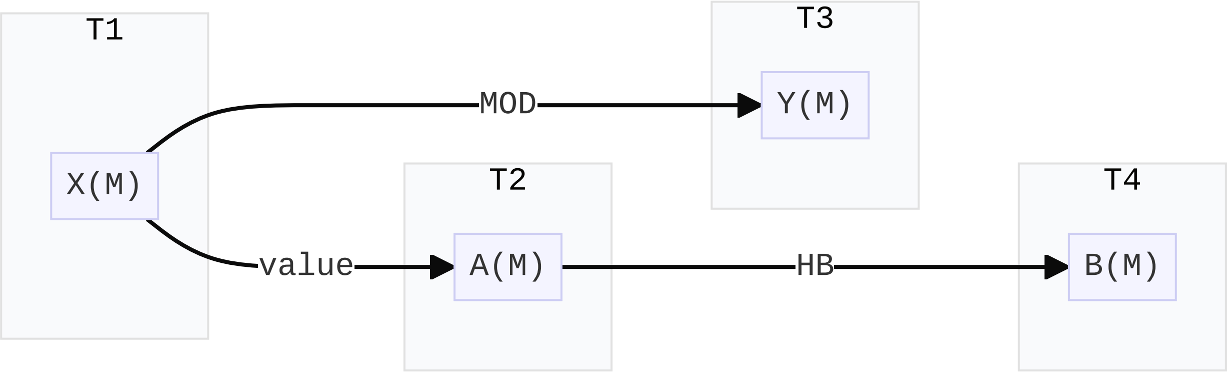
Write-write coherence

If an operation *A* that modifies an atomic object *M* happens before an operation *B* that modifies *M*, then *A* shall be earlier than *B* in the modification order of *M*.



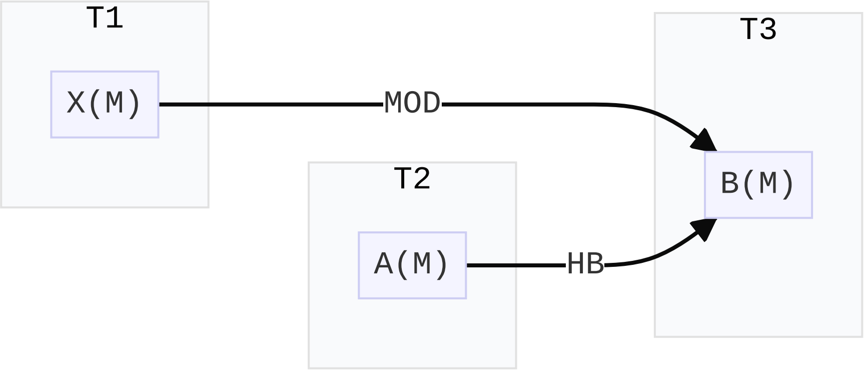
Read-read coherence

If a value computation *A* of an atomic object *M* happens before a value computation *B* of *M*, and *A* takes its value from a side effect *X* on *M*, then the value computed by *B* shall either be the value stored by *X* or the value stored by a side effect *Y* on *M*, where *Y* follows *X* in the modification order of *M*.



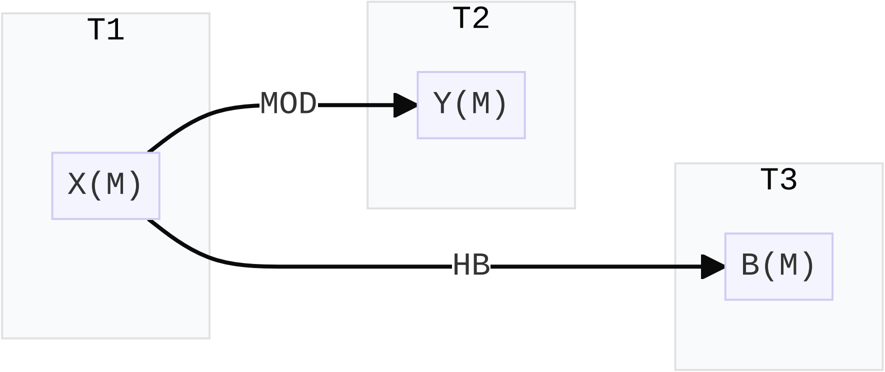
Read-write coherence

If a value computation A of an atomic object M happens before an operation B that modifies M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M.



Write-read coherence

If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M.



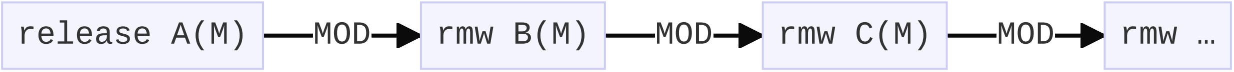
The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations.

The value observed by a load of an atomic depends on the “happens before” relation, which depends on the values observed by loads of atomics. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the “happens before” relation derived as described above, satisfy the resulting constraints as imposed here.

C++ standard: [intro.races]

Release sequence

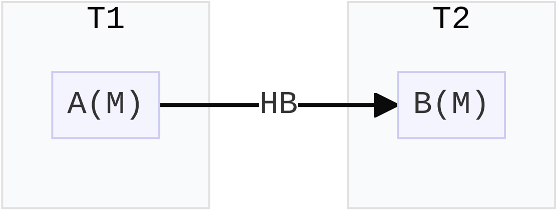
A release sequence headed by a release operation A on an atomic object M is a maximal contiguous sub- sequence of side effects in the modification order of M, where the first operation is A, and every subsequent operation is an atomic read-modify-write operation.



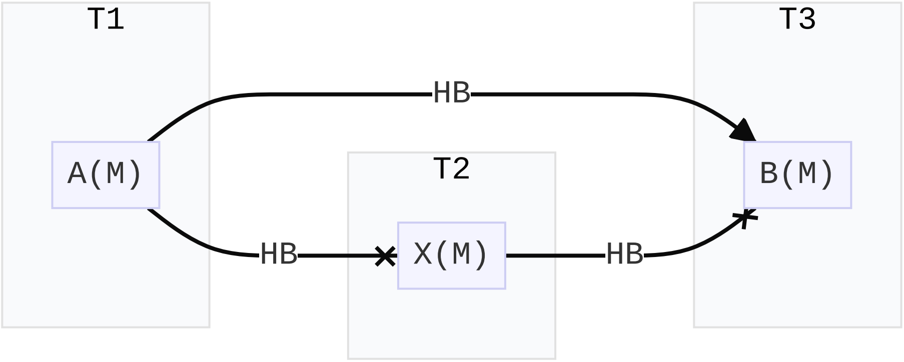
Visible side effect

A visible side effect *A* on a scalar object or bit-field *M* with respect to a value computation *B* of *M* satisfies the conditions:

- A happens before B and



- there is no other side effect *X* to *M* such that *A* happens before *X* and *X* happens before *B*.



The value of a non-atomic scalar object or bit-field *M* , as determined by evaluation *B*, shall be the value stored by the visible side effect *A*.

Potentially concurrent

Two actions are potentially concurrent if

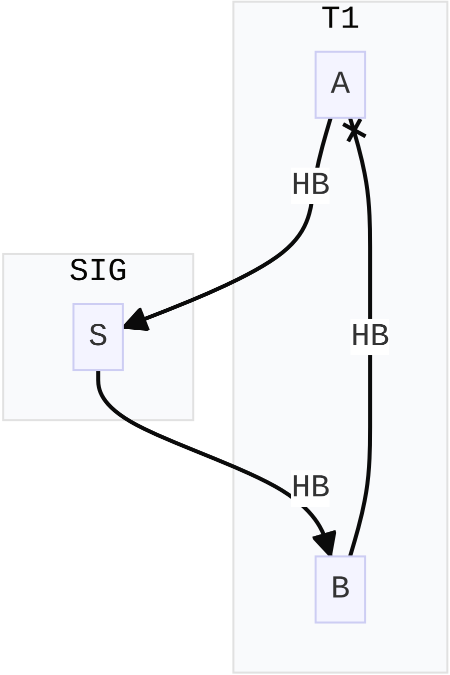
- they are performed by different threads, or
- they are unsequenced, at least one is performed by a signal handler, and they are not both performed by the same signal handler invocation.

Data race

The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race results in undefined behavior.

It can be shown that programs that correctly use mutexes and `memory_order::seq_cst` operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as “sequential consistency”. However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result has undefined behavior.

Two accesses to the same object of type `volatile std::sig_atomic_t` do not result in a data race if both occur in the same thread, even if one or more occurs in a signal handler. For each signal handler invocation, evaluations performed by the thread invoking a signal handler can be divided into two groups A and B, such that no evaluations in B happen before evaluations in A, and the evaluations of such `volatile std::sig_atomic_t objects` take values as though all evaluations in A happened before the execution of the signal handler and the execution of the signal handler happened before all evaluations in B.


T1
A
HB
SIG
S
HB
HB
B