Memory Models for C/C++ Programmers

Manuel Pöter

Jesper Larsson Träff

Research Group Parallel Computing

Faculty of Informatics, Institute of Computer Engineering

TU Wien

Favoritenstrasse 16, 1040 Wien, Austria

March 29, 2018

1   Introduction

The memory model is the crux of the concurrency semantics of shared-memory systems. It defines the possible values that a read operation is allowed to return for any given set of write operations performed by a concurrent program, thereby defining the basic semantics of shared variables. In other words, the memory model specifies the set of allowed outputs of a program's read and write operations, and constrains an implementation to produce only (but at least one) such allowed executions. The memory model may and often does allow executions where the outcome cannot be inferred from the order in which read and write operations occur the the program. It is impossible to meaningfully reason about a program or any part of the programming language implementation without an unambiguous memory model. The memory model defines the possible outcomes of a concurrent programs read and write operations. Conversely, the memory model also defines which instruction reorderings may be permitted, either by the processor, the memory system, or the compiler.

In this note, our programming languages will be C and C++ and programs execute threads concurrently. Our machine consists of CPU's or cores connected to a common, shared memory from which the cores (at the programming language level: The threads) read and write data in some order. We also distinguish between program (as written) and the code (as compiled and executed by the cores).

Recent books by Michael L. Scott [Sco13] and Sorin et al. [SHW11] provide good introductions to memory models in both hardware and software. Overviews of the complex issues can be found in numerous papers, for instance those by Adve and Gharachorloo [AG96], Adve and Boehm [AB10, AB11], McKenney [McK05], Batty et al. [BOS+11, BMN+15], to mention a few.

A memory model can be refined to differentiate between the *programming language memory model* and the *hardware memory model*.

- Language memory model: Defines the optimizations, memory access re-writes and reorderings a compiler is allowed to perform when transforming program into code.

- Hardware memory model: Defines the optimizations and memory access reorderings a specific hardware architecture is allowed to perform.

These optimizations can cause memory accesses to be executed or perceived in orders that differ from what is defined in the source code, leading to distinguishing between the following four orderings:

Source code order: The order in which the memory operations are specified by the source code by the programmer.

Program order: The order in which the memory operations are specified in the machine code that is executed by the CPU. Note that this can differ from the source code order, because depending on the definition of the language memory model, compilers are allowed to reorder instructions as part of the optimization process.

Execution order: The order in which the individual memory-reference instructions are executed on a given CPU. The execution order can differ from the compiled order due to optimizations based on the hardware memory model of the specific CPU-implementation.

Perceived order: The order in which a CPU perceives its and other CPUs' memory operations. The perceived order can differ from the execution order due to caching, interconnect and memory-system optimizations. Different CPUs can perceive the same set of memory operations as occurring in different orders. This is also defined by the hardware memory model.

The reason why these orders can be different stems from the fact that increases in memory performance have not kept up with the rate at which CPU instruction performance has increased. To try to hide the fact that memory operations are increasingly expensive compared to simple register-to-register instructions, modern CPUs are equipped with increasingly large caches in order to reduce the overhead of these memory accesses.

However, CPUs have become so fast that even these caches cannot keep up with them. Therefore, caches are often partitioned into *banks* that can operate nearly independently from each other. This allows each of the banks to run in parallel, therefore keeping up better with the CPU. Memory usually is divided evenly among the banks by address, e.g., even-numbered cache lines are processed by bank 0 while odd-numbered cache lines are processed by bank 1. However, this type of hardware parallelism now allows memory operations to complete out of order.

Consider two memory write operations where the first one is processed by bank 0 and the second one is processed by bank 1. Now if bank 0 is already busy processing an earlier request and bank 1 is idle, the second write would be visible to another CPU *before* the first write – the writes would be perceived *out of order* by other CPUs. However, this kind of reordering is not limited to write operations – read operations can be reordered in a similar manner.

2 Sequential consistency

Sequential consistency as introduced by Lamport [Lam79] is an (the most) intuitive memory model for reasoning about the outcome and correctness of concurrent program,

algorithm or data structure. Many results on concurrent algorithms or data structures either explicitly oder implicitly assume a sequentially consistent memory model.

A natural view of the execution of a multi-threaded program is as follows. For each step one of the threads is randomly chosen and the next step in that thread's execution (in say, the program or the compiled order) gets executed. This process is repeated until the program as a whole terminates. This is effectively equivalent to taking all the steps of all threads in (program or compiled) order, and interleaving them in some way, resulting in a single total order of all steps. No reordering of the thread's steps is permitted. Therefore, whenever an object is accessed, the last value stored to the object in this order is retrieved. An execution that can be understood as such an interleaving is referred to as *sequentially consistent.*

Listing 1 shows parts of an implementation of Dekker's mutual exclusion algorithm [Dij65]. The steps of the two threads can be interleaved in many ways, but since the program order is preserved it is ensured that at least one of the load operations sees the value of the prior store operation, i.e., the program order, execution order and perceived order are all identical. It is therefore not possible that both, `r1` and `r2`, are zero.

Listing 1: Dekker's mutual exclusion algorithm

```
1  Initially: X = 0, Y = 0
2
3  Thread 1:
4    X = 1
5    r1 = Y
6
7  Thread 2:
8    Y = 1
9    r2 = X
```

Unfortunately ensuring sequential consistency is quite expensive and none of todays processor architectures provide a fully sequentially consistent memory model. While they allow to enforce sequential consistency at certain points normal execution is not sequentially consistent, but depends highly on the implementation of the specific architecture.

3 Weaker memory models

3.1 x86-TSO

Even though the Intel x86 memory model is somewhat weaker than the sequentially consistent model, it is still one of the strongest models amongst todays modern CPU implementations. However, as Sewell et al. [SSO+10] point out, for a long time the information provided by Intel and AMD on their respective x86 architecture implementations were mostly informal, missing concrete examples and sometimes even inconsistent with the actual implementation.

Based on the available material they formally described a new memory model called *x86-TSO* (Total Store Order) which is consistent with the concrete examples in Intel's and AMD's latest documentation available at that time. This model is illustrated in Figure 1.

As can be seen, the hardware threads interact with a storage subsystem represented by the dotted box. The storage subsystem comprises a shared memory that maps addresses
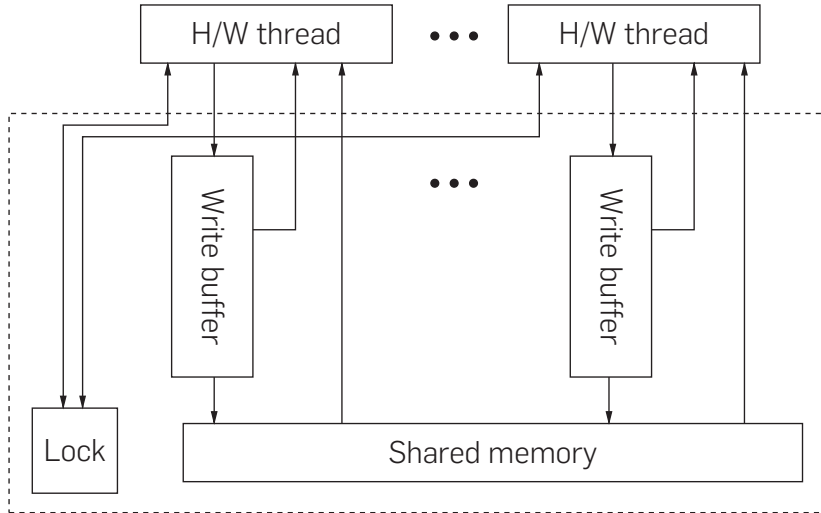
Figure 1: x86-TSO block diagram from [SSO$^+$10].

to values, a global lock to indicate when a particular hardware thread has exclusive access to memory, and one store buffer per hardware thread. A formal definition of the behavior of the storage subsystem can be found in [SSO$^+$10], but the main points are:

- The store buffers are FIFO and a reading thread must read its own most recent buffered write, if there is one, to that address. Otherwise reads are satisfied from shared memory.

- An `mfence` instruction flushes the store buffer of that thread.

- To execute a `lock`'d instruction[1], a thread must first acquire the global lock. At the end of the instruction, it flushes its store buffer and releases the lock. While the lock is held by one thread, no other thread can read. This essentially means that `lock`'d instructions enforce sequential consistency.

- A buffered write from a thread can propagate to the shared memory at any time except when some other thread holds the lock.

The model defines the perceived execution order. x86-TSO does not permit local reordering except of reads after writes to different addresses.

Since writes are buffered, the new value is not visible to other threads until it has propagated to the shared memory. Therefore Dekker's algorithm from Listing 1 no longer guarantees mutual exclusion under the x86-TSO model, as it is perfectly possible that `r1` as well as `r2` are both zero. This could be resolved by either introducing an `mfence` instruction after the first store operation, or by performing the store operation using a `lock xchg` instruction.

Another memory model that is very similar to x86-TSO is the SPARC v8 TSO model [SPA92].

---

[1]These are read-modify-write instructions with a `lock` prefix for atomicity like, e.g., `lock xadd` (atomic fetch-and-add) or `lock cmpxchg` (atomic compare-and-swap). A complete list of instructions that support the `lock` prefix can be found in [Int16, 8.1.2.2].

3.2   ARM and POWER

The ARM and POWER architectures have considerably more relaxed memory models, allowing a wider range of hardware optimizations. Maranget et al. [MSS12] provide a very detailed and extensive description of both architectures and their observable behaviors.

While memory order relaxations can improve performance, power efficiency and hardware complexity, it makes the life of a programmer, who is implementing concurrent data structures, significantly harder. In contrast to TSO models the following behaviors are possible on these architectures:

1. Hardware threads can perform reads and writes out-of-order, or even speculatively, i.e., before preceding conditional branches have been resolved. Any local reordering is allowed unless otherwise specified.

2. The memory system does not guarantee that a write becomes visible to all other hardware threads at the same time (this behavior is called *write non-atomicity*).

Since a certain ordering of instructions is crucial already for the simplest non-blocking data structures, these architectures provide various memory barriers and dependency guarantees that the programmer has to use correctly in order to enforce a desired appropriate ordering of memory operations.

To understand the behavior of such a machine it can be helpful to think of each hardware thread as effectively having its own copy of memory as illustrated in Figure 2. The collection of all the memories and their interconnect (i.e., everything except the threads) is usually referred to as the *storage subsystem*. A write by one thread may propagate to other threads in any order, and the propagations of writes to different addresses can be interleaved arbitrarily, unless they are constrained by barriers or cache coherence. One can also think of barriers as propagating from the hardware thread that executed them to each of the other threads.

The ARM `dbm` and POWER `sync` barrier (fence) instructions can be used to enforce the following orderings between two instructions:

Read/Read the barrier ensures that they are satisfied and committed in program order.

Read/Write the barrier ensures that the read is satisfied and committed before the write can be committed (and thus propagated and become visible to others).

Write/Write the barrier ensures that the first write is committed and has propagated to all other threads before the second write is committed.

Write/Read the barrier ensures that the write is committed and has propagated to all other threads before the read is satisfied.

The POWER architecture provides an additional "lightweight sync" instruction called `lwsync`, which is weaker and potentially faster than `sync`. It mainly differs in how a write before the barrier is handled relative to the second instruction:

Write/Write the barrier ensures that for any particular thread, the first write propagates to that thread before the second.
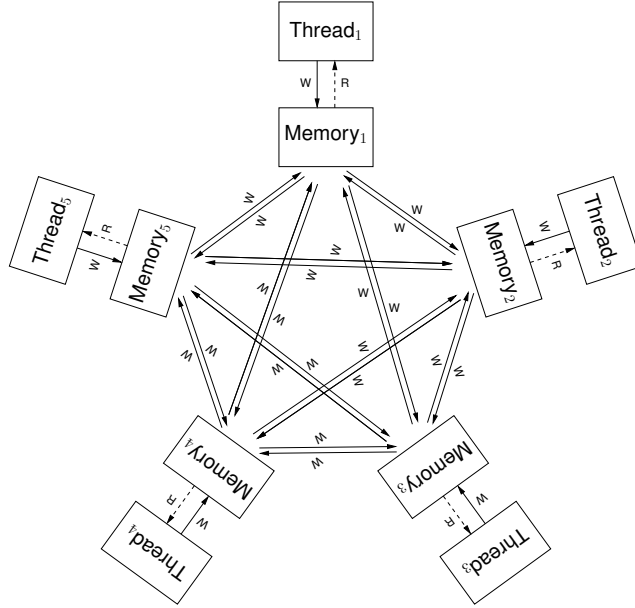
Figure 2: Storage subsystem from [MSS12].

Write/Read the barrier ensures that the write is committed before the read is satisfied, but the read can be satisfied before the write is propagated to any other thread.

In addition to barriers, these architectures provide the following dependencies to enforce orderings:

Address Dependency: There is an address dependency from a read to a program-order-later read or write when the value read by the first instruction is used to compute the address of the second instruction.

Control Dependency: There is a control dependency from a read to a program-order-later read/write where the value read by the first instruction is used to compute the condition of a conditional branch that is program-order-before the second instruction.

Data Dependency: There is a data dependency from a read to a program-order-later write where the value read by the first instruction is used to compute the value that is written by the second instruction.

A read-to-read control dependency has little force since ARM and POWER processors can speculatively execute past the conditional branch, thus satisfying the second read before the first. To give a read-to-read control dependency some force, one can add an ISB (ARM) or isync (POWER) instruction between the conditional branch and the second read.

In contrast, a read-to-write control dependency does have some force: the write cannot be seen by any other thread until the branch is committed, and hence until the value of the first read is fixed.

To summarize, from one read to another, an address dependency or control dependency with ISB/isync will prevent the second read from being satisfied before the first one, while

a plain control dependency will not. From a read to a write, an address, control or data dependency will prevent the write from being visible to any other thread before the value of the read is fixed.

4 The C++11 memory model

On August 12$^{\text{th}}$ 2011 the new C++ standard, now commonly referred to as C++11, was approved and ratified by ISO, replacing the previous version C++03. Since this official ISO C++ standard is not freely available we instead refer to the "Working Draft, Standard for Programming Language C++" from January 2012 [CSC12] which contains the C++11 standard plus minor editorial changes.

This new C++ standard is the first version to define the notion of *multi-threaded executions*. The C++ Standard prior to C++11 specified program execution in terms of observable behavior, which in turn described sequential execution on an implicitly single-threaded abstract machine. Therefore multi-threaded C++ programs relied on libraries for threading support, like POSIX threads, Win32, or Boost. Unfortunately, a pure library approach in which the compiler is designed independently of threading issues entails all sorts of problems as discussed in the well-known paper by Boehm [Boe05]. Without a clearly defined memory model as a common ground between the compiler, the hardware, the threading library, and the programmer, multi-threaded C++ code is fundamentally at odds with compiler and processor-level optimizations [MA04]. That is why with the introduction of multi-threaded executions also a new memory model had to be defined.

The memory model defines when multiple threads may access the same memory location, and specifies when updates by one thread become visible to other threads. It is largely based on the work by Boehm, Alexandrescu et al. [BA08, ABH$^+$04].

The new C++11 library defines a number of synchronization operations, consisting of atomic operations [CSC12, 29, pp. 1100] and operations on mutexes [CSC12, 30, pp. 1118]. These operations play an important role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume, an acquire, a release, or both an acquire and release operation.

A synchronization operations without an associated memory location is a fence and can be either an acquire, a release, or both an acquire and release fence. Fences are discussed in more detail in Section 4.3.

In addition, there are relaxed atomic operations, which are not synchronization operations, as they do not affect the visibility of any assignment to other threads, and atomic read-modify-write operations, which have special characteristics as explained later.

The execution model is defined on the basis of *evaluations*. There are two kinds of evaluations performed by the compiler for each expression or subexpression (both of which are optional):

Value computation: Calculation of the value that is returned by the expression. This may involve determining an object's identity (e.g., when the expression returns a reference to some object) or reading the value previously assigned to an object (e.g., when the expression returns a number or some other value)

Side effect: Access (read or write) to a `volatile` object, modify (write) to a (non-volatile) object, calling a library I/O function, or calling a function that does any of those operations. All such side effects are changes in the state of the execution environment.

However, conforming implementations are required to emulate (only) the observable behavior of the abstract machine. This is often called the "as-if" rule, because an implementation is free to disregard any requirement of the C++ standard as long as the result is *as if* the requirement had been obeyed, as far as can be determined from the observable behavior of the program, i.e., it effectively *allows any and all code transformations that do not change the observable behavior of the program.*

One of the most important aspects is the definition of a data race [CSC12, 1.10.21, p. 14]:

> The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither *happens-before* the other. Any such data race results in undefined behavior.

Conflicting actions are defined as follows [CSC12, 1.10.4, p. 11]:

> Two expression evaluations conflict if one of them modifies a memory location and the other one accesses or modifies the same memory location.

This definition implies that any program written according to the old standard that uses some other threading libraries and shares any data between those threads exhibits undefined behavior. The memory operations are ordered by means of the *happens-before* relationship that can be roughly described as follows:

> Let A and B represent operations performed by a multi-threaded process. If A happens-before B, then the memory effects of A effectively become visible to the thread performing B before B is performed.

The *happens-before* relation (denote: $\rightarrow$) is a strict partial order and as such transitive, irreflexive and antisymmetric.

transitivity: $\forall a, b, c,$ if $a \rightarrow b$ and $b \rightarrow c,$ then $a \rightarrow c$

irreflexivity: $\forall a, a \nrightarrow a$

antisymmetry: $\forall a, b,$ if $a \rightarrow b$ then $b \nrightarrow a$

The complete formal definition specifically for C++ can be found in [CSC12, 1.10, p. 11-14].

Sequenced before is an anti-symmetric, transitive, pair-wise relation between instructions executed by a single thread, which induces a partial order among those evaluations [CSC12, 1.9.13, p. 10]. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B.

One might assume that the definition of sequenced-before relation effectively prevents the compiler from performing any instruction reordering. But since the sequenced-before relation is only defined between instructions executed by *a single thread*, the compiler may

freely reorder instructions as long as these changes are not observable to the executing thread (recall the "as-if" rule).

A *happens-before* order between two operations from the same thread (source code order) is implicitly given by the *sequenced-before* order [CSC12, 1.10.12, p. 13]. A *happens-before* order between two operations from different threads (in the standard this is referred to as *inter-thread-happens-before*) must be established using atomic operations.

Since the happens-before relation can be established between operations in *different threads*, the compiler can only reorder instructions as long as these changes are not observable by *either* thread. We will get back to this again in Section 4.2.

4.1 Atomic operations

The C++11 standard library introduces a new generic class `std::atomic<T>` that provides the following atomic operations to work with instances of type `T`:

- `load(std::memory_order order)`

- `store(T desired, std::memory_order order)`

- `exchange(T desired, std::memory_order order)`

- `compare_exchange_weak(T& expected, T desired, std::memory_order order)`

- `compare_exchange_strong(T& expected, T desired, std::memory_order order)`

For integral and pointer types it also provides the following operations:

- `fetch_add(T arg, std::memory_order order)`

- `fetch_sub(T arg, std::memory_order order)`

And for integral types only it provides the following additional operations.

- `fetch_and(T arg, std::memory_order order)`

- `fetch_or(T arg, std::memory_order order)`

- `fetch_xor(T arg, std::memory_order order)`

The `order` parameter for all operations defaults to `memory_order_seq_cst`, but the operations can be relaxed by explicitly defining a weaker memory order. The available memory orders and their effects are discussed in Section 4.2.

For many of these operations the class also provides operators like the assignment operator for `store` or post-fix increment for `fetch_add`. These operators are a nice syntactic sugar, but they rely on the standard memory order for all operations and do not allow customization.

The `atomic` class can work with any type `T`, regardless of its size. For types with a size less or equal to the size of a pointer all the operations are usually lock-free. For other types the implementation may fall back to a lock-based version to achieve atomicity. The class provides the `is_lock_free` method for checking whether the operations on the given type can be performed in a lock-free manner.

4.2 Memory orders

Each atomic operation takes a parameter of the type `memory_order` which is an enumeration type with the following memory order values (from strong to relaxed memory order):

- `memory_order_seq_cst`

- `memory_order_acq_rel`

- `memory_order_release`

- `memory_order_acquire`

- `memory_order_consume`

- `memory_order_relaxed`

As explained in Section 2, there is a single total order S of all sequentially consistent operations. An operation B that performs a load on an object M will observe the result of the last modification A of M that precedes B in S [CSC12, 29.3.3, p. 1104]. From this it follows that there is always a *happens-before* relation between two `memory_order_seq_cst` operations operating on the same object.

The orders `memory_order_consume` and `memory_order_acquire` can only be used for operations that perform a *read*, `memory_order_release` can only be used for operations that perform a *write* and `memory_order_acq_rel` can only be used for operations that perform a *read-modify-write* operation. Although the language does not enforce these constraints some implementations do check them at runtime[2].

A *happens-before* relationship can be established by using the following combinations of memory orders [3]:

- `memory_order_seq_cst` and `memory_order_seq_cst`

- `memory_order_release` and `memory_order_acquire`

- `memory_order_release` and `memory_order_consume`

An atomic operation A that performs a store-release operation on an atomic object M *synchronizes with* an atomic operation B that performs a load-acquire operation on M and takes its value from any side effect in the release sequence (defined below) headed by A. This *synchronize-with* order is compatible with the *inter-thread-happens-before* order, i.e., if A *synchronizes with* B, then A *inter-thread-happens-before* B.

An example can be seen in Listing 2: Thread A writes two values to the two variables x and y. In order to guarantee that when thread B sees the new value of y it also sees the new value of x, a *happens-before* relation must be established. In line 5 thread A uses *release* semantics to store the new value of y while in line 8 thread B uses *acquire* semantics to load the value of y. If this *acquire* load returns the value stored by the

---

[2]For example when the `DEBUG` macro is defined the Microsoft STL implementation inserts code to verify these constraints at runtime.

[3]`memory_order_acq_rel` is the combination of `memory_order_release` and `memory_order_acquire`. Wherever either of these is used it is also possible to use `memory_order_acq_rel`.

release store the two operations *synchronize with* each other, therefore establishing a *happens-before* relation. Since the store to x is *sequenced before* the store to y and the load of y is *sequenced before* the load of x it follows by transitivity that the store to x *happens-before* the load of x.

Listing 2: Example of *synchronize-with* relation with release/acquire operations.

```
1  std:.atomic<int> x, y;
2
3  // thread A
4  x.store(1, std::memory_order_relaxed);
5  y.store(2, std::memory_order_release);
6
7  // thread B
8  y.load(std::memory_order_acquire);
9  x.load(std::memory_order_relaxed);
```

As previously described, in accordance with the "as-if" rule the compiler may freely reorder any instructions as long as the observable behavior for the executing thread is consistent with the sequenced-before relation. With regards to atomic operations, and specifically the inter-thread-happens-before relation established via release/acquire operations, the following additional restrictions apply:

- atomic operations on the same object may never be reordered [CSC12, 1.10.19, p. 14],

- (non-)atomic write operations that are sequenced before a release operation A may not be reordered after A,

- (non-)atomic load operations that are sequenced after an acquire operation A may not be reordered before A.

These restrictions effectively retain the source code order of the according instructions and thus ensure that the behavior observed by the thread that performs the acquire operation is consistent with the happens-before relation.

The `memory_order_consume` order is based on the address dependency concept described in Section 3.2. As such it is not only more complicated but also weaker than `memory_order_acquire`. According to Hans Boehm, the current definition of `memory_order_consume` in the standard is not useful[4]. He has therefore proposed to temporarily deprecate `memory_order_consume` in C++17 and the proposal was accepted in the Oulu meeting in July 2016. The `memory_order_relaxed` order can never be used to establish a *happens-before* order.

All modifications to a particular atomic object occur in some particular total order, called the *modification order*. If A and B are modifications of an atomic object M and A *happens-before* B, then A precedes B in the modification order of M. There are separate modification orders for each atomic object and there is no requirement that these can be combined into a single total order for all atomic objects.

Atomic read-modify-write operations shall always read the last value in the modification order written before the write associated with the read-modify-write operation [CSC12, 29.3.12, p. 1105].

---

[4]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0371r0.html

A *release sequence* is a subsequence of the modification order of an atomic object. It is headed by a release operation A and followed by an arbitrary number of

- atomic operations performed by the same thread that performed A or

- atomic read-modify-write operations.

For operations performed by the same thread that performed A it is not relevant which memory order is used – it can even use `memory_order_relaxed`. If a thread is reading a value that is part of a *release sequence* using acquire semantics, this read synchronizes with the release operation that is heading the sequence. Note that there can exist several release sequences on the same object at the same time. Suppose there are two release-CAS operation on some atomic object A. Since both use release semantics, they both act as head of their own release sequence. And since a CAS is an atomic read-modify-write operation, the second CAS is also part of the release sequence headed by the first CAS. So an acquire-load on A that returns the value stored by the second CAS will actually synchronize-with *both* release-CAS operations.

The C++11 standard describes two different compare-and-swap operations for atomic objects: `compare_exchange_strong` and `compare_exchange_weak`. The difference is that `compare_exchange_weak` is allowed to fail spuriously, that is, act as if `*obj != *expected` even if they are equal, but can result in better performance on some platforms. Both operations take two `memory_order` parameters: The first one describes the semantics of the read and write operations in case of success, and the second one describes the semantics of the reload operation in the failure case. In addition, the standard defines overloads for both operations taking only a single `memory_order` parameter. They forward to the two parameter version, passing the given `memory_order` as the first argument. The second argument is also derived from the given `memory_order` by removing any semantics that are only relevant for write operations, i.e., `memory_order_release` is replaced with `memory_order_relaxed` and `memory_order_acq_rel` is replaced with `memory_order_acquire` [CSC12, 29.6.5.21, p. 1113].

The C++11 standard states that "the failure argument shall be no stronger than the success argument" [CSC12, 29.6.5.20, p. 1113]. But Bastien and Boehm noted that the standard does not define the term "stronger" in this context, and also questioned whether there is even a point in restricting success/failure orderings [BB16]. Based on their proposal this requirement was therefore removed in C++17.

4.3 Fences

Another synchronization operation that can be used to establish a *happens-before* relation is a *fence* [CSC12, 29.8, p. 1116]. Just like the operations on atomic objects, the `atomic_thread_fence` operation also takes a `memory_order` parameter and, depending on the order, it has the following effects:

- has no effects, if `order == memory_order_relaxed`

- is an *acquire-fence*, if `order == memory_order_acquire` or `order == memory_order_consume`

- is a *release-fence*, if `order == memory_order_release`

- is both an *acquire-fence* and a *release-fence*, if `order == memory_order_acq_rel`

- is a sequentially consistent acquire- and release-fence, if `order == memory_order_seq_cst`

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. Alternatively a release fence can synchronize with a load-acquire on object M and an acquire fence can synchronize with a store-release on object M, given that the same sequenced before relations between the fences and the corresponding operation are in place.

An adapted version of the previous example can be seen in Listing 3. The release-fence in line 5 is sequenced before the store operation in line 6 and the load operation in line 9 is sequenced before the acquire-fence in line 10. Therefore, when the load operation in line 9 returns the value written by the store operation in line 6, the acquire-fence *synchronizes with* the release-fence. From here the happens-before relation for the operations on x follows as already described in the previous example for the release/acquire operations in Listing 2.

Listing 3: Example of *synchronize-with* relation with release/acquire fences.

```
1  std:.atomic<int> x, y;
2
3  // thread A
4  x.store(1, std::memory_order_relaxed);
5  std::atomic_thread_fence(std::memory_order_release);
6  y.store(2, std::memory_order_relaxed);
7
8  // thread B
9  y.load(std::memory_order_relaxed);
10 std::atomic_thread_fence(std::memory_order_acquire);
11 x.load(std::memory_order_relaxed);
```

A `memory_order_seq_cst`-fence is not only both a *release* and an *acquire*-fence, but also provides some additional properties [CSC12, 29.3.4-29.3.8]. They are also part of the single total order of all sequentially consistent operations, enforcing the following observations:

- For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced before B, then B observes either the last `memory_order_seq_cst` modification of M preceding X in the total order S or a later modification of M in its modification order.

- For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.

- For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B observes either the effects of A or a later modification of M in its modification order.

- For atomic operations A and B on an atomic object M, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B occurs later than A in the modification order of M.

References

[AB10]   Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, August 2010.

[AB11]   Sarita V. Adve and Hans-Juergen Boehm. Memory models. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1107–1110. Springer, 2011.

[ABH+04] Andrei Alexandrescu, Hans-J. Boehm, Kevlin Henney, Doug Lea, and Bill Pugh. Memory model for multithreaded C++. Document WG21/N1680=J16/04-0120; available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1680.pdf`, September 2004.

[AG96]   Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[BA08]   Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78, 2008.

[BB16]   J.F. Bastien and Hans-J. Boehm. Fail or succeed: there is no atomic lattice. C++ standards committee paper, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0418r1.html`, August 2016.

[BMN+15] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *Programming Languages and Systems - 24th European Symposium on Programming (ESOP)*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307, 2015.

[Boe05]   Hans-Juergen Boehm. Threads cannot be implemented as a library. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 261–268, 2005.

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 55–66, 2011.

[CSC12]   Stefanus Du Toit C++ Standards Committee. Working draft, standard for programming language c++. C++ standards committee paper, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`, January 2012.

[Dij65]   Edsger Wybe Dijkstra. Cooperating sequential processes. Technical Report EWD-123, X, 1965.

[Int16]   Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, September 2016.

[Lam79]   Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess procegrams. *IEEE Computer*, 28(9):690–691, 1979.

[MA04]   Scott Meyers and Andrei Alexandrescu. C++ and the perils of double-checked locking. *Doctor Dobbs Journal*, Jul 2004.

[McK05]   Paul E. McKenney. Memory ordering in modern microprocessors. *Linux Journal*, 30:52–57, 2005.

[MSS12]   Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. Technical report, `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`, October 2012.

[Sco13]   Michael L. Scott. *Shared-Memory Synnchronization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publisher, 2013.

[SHW11]   Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publisher, 2011.

[SPA92]   SPARC International Inc. *The SPARC Architecture Manual Version 8 Revision SAV080SI9308*, 1992.

[SSO+10]   Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.