The C++ Memory Model

Abstract

This paper describes current support of atomics and memory ordering provided by C++ which are the foundations of its memory model. It is based on Rainer Grimm[1], Valentin Ziegler and Fabio Fracassi[2] conferences

1 Multi-threading with C++

Since C++11 a new set of features has given answer to the requirements of multicore architectures thanks to:

- A standardized **thread interface**:
  - Threads and tasks.
  - Shared data protection and safe initialization.
  - Thread-local data (i.e. allocated in TLS).
  - Synchronization of threads.
- A well defined **memory model**:
  - Atomic operations
  - Partial ordering of operations
  - Visible effects of operations

The memory model describes the interactions between threads through memory, in particular relatively to their shared use of data. The compiler can be forced to respect some constraints which affect its code generator in order to avoid *instruction reordering* that would break the expected intuitive behaviour of the program.

The constraints fixed by the memory model also help to avoid *data races* on shared data. In general, a data race happens under the following conditions:

- Memory operations deal with a memory location which is an object of scalar type or a maximal sequence of adjacent non-zero width bit-fields.
- Two or more threads access to the memory location concurrently and at least one of them writes to it.

[1] https://www.youtube.com/watch?v=e0DsVqZLMzU

[2] https://www.think-cell.com/en/career/talks/pdf/think-cell_talk_memorymodel.pdf

- Two conflicting actions are performed by distinct threads and none of these actions *happens-before* the other.

For example:

```
int i;                // These are all shared data whose
char c;               // variable names represents memory
int a:5, b:7;         // locations which are scalar or
unsigned char *p;     // adjacent bit-fields.

void T1_func() {
  a = 10;
  unsigned char *tmp = p;
  int n = i;          // Data race!!!
}

void T2_func() {
  c = '@';
  unsigned char *tmp = p;
  i = 1024;           // Data race!!!
}

int main() {
  thread T1(T1_func);
  thread T2(T2_func);

  T1.join();
  T2.join();
}
```

2 The Contract

In order to obtain highly optimized programs tailored for the target architecture and, at the same time, preserve the program semantic, a *contract* is established between the programmer and the system, according to which the former follows some rules concerning:

- Atomic operations.
- Partial ordering of memory operations.
- Visible effects of memory operations.

The system will then perform code *optimization* at its best and the generated assembly code won't break the original source code as expected by the programmer.

The optimization level that the compiler can perform varies with the constraints that the contract imposes. The weaker are the constraints, the more are the possible optimizations at compile-time:

- A *single-threaded* program is the strongest contract as there's no risk of data races but the optimization possibilities are limited.

- A *multi-threaded* program guarantees better performances but it's also more risky as it is potentially exposed to data-races, specially if the expertise level of the programmer is not adequate.
- **Atomic operations** and **memory ordering** constraints are an even weaker contract but they allow the best optimizations for a concurrent program. However the risks are the highest, as the expertise level required for safe programs.

As seen later, operations on atomics bring a memory ordering with themselves which can be of the following main types:

Sequential-consistency This is a *strong* memory model as it imposes a global ordering of memory operations that can't be changed. It is like all threads see a *universal clock* whose ticks corresponds each one to a program instruction.

Acquire-release semantic It allows faster code but only *partial* ordering of memory operations is possible, allowing synchronization only between operations involving the same atomics. Needless to say that intuition is betrayed and the programmer might incur in unexpected results.

Relaxed semantic This is a *weak* memory that allows both the compiler and the processor to move around memory operations without limits, providing the best optimizations possible but incurring in the greatest risks.

3 Atomics

Operations on atomics are the foundation of the C++ memory model as they define **synchronization and ordering constraints** which affect all operations, even non-atomic.

Atomic data types allow to declare *data race free* variables which are accessed thanks to their atomic operations. In particular, an atomic `store` synchronizes with an atomic `load` on the same data.

Even the higher level thread interface uses the capabilities of atomic operations to implement mutexes, condition variables and locks.

3.1 The `atomic_flag` data type

This is the first and simplest atomic data type which is entirely *lock-free*[3] for every implementation. It works like a boolean type, representing a single bit of information, and it allows only two operations as function members:

- `test_and_set`, that atomically sets the variable (i.e. setting it to `set`) and returns its previous value.
- `clear`, which resets its content (i.e. setting it to `clear`).

[3]The other atomic structures like integers, pointers and user-defined types, provide operations that might use locks under the hood.

Notice that its `set` and `clear` possible values are not necessarily mapped to `1` and `0` respectively. They could be the reversals for some implementations.

The `atomic_flag` data type is the building block for **spinlocks**, a locking mechanism for multi-threaded programs which is based on *busy-waiting*. In order to protect a shared resource, threads have to lock it by setting it through a `test_and_set` call and they have to unlock it when done, through a `clear` call.

When the lock is set, meaning that a thread is owning the shared resource, the other concurrent threads will repeatedly try to lock the resource, but they will be stuck in the loop until the `test_and_set` call returns `set`, meaning the resource was previously busy and now it's been acquired by the caller thread:

```
class Spinlock {
  std::atomic_flag flag;

public:
  Spinlock() : flag(ATOMIC_FLAG_INIT) {}

  void lock() {
    while (flag.test_and_set());
  }

  void unlock() {
    flag.clear();
  }
};

Spinlock spin;

void WorkOnResource() {
  spin.lock();
  sleep_for(seconds(2));
  spin.unlock();
}

int main() {
  std::thread t1(workOnResource);
  std::thread t2(workOnResource);

  t1.join();
  t2.join();
}
```

Needless to say that the loop keeps the threads actually busy (100% utilization) until they acquire the lock on the shared data.

3.2 The `atomic<bool>` data type

This is actually a boolean as it can be set to `true` or `false` and it provides the most relevant operation for lock-free programming: the **CAS (Compare and Swap)**, which is provided by the following operator:

```
bool compare_exchange_strong(exp, des)
```

It is provided by the `atomic<T>` template and so it's not limited to atomic boolean types and its meaning is as follows:

```
atom<bool> SharedData;
SharedData.compare_exchange_strong(exp, des);

// Atomically performs:
//
// if (*SharedData == exp) {
//   *SharedData = des;
//   return true;
// } else {
//   *exp = SharedData;
//   return false;
// }
```

Condition variables, which allow to synchronize multiple threads, can be easily implemented using `atomic<bool>`. At first, the standard way to define and use such a variable thanks to the thread interface:

```
vector<int> SharedData;
bool DataReady;

mutex mtx;
condition_variable cvar;

void SetDataReady() {                       // Producer
  SharedData = { 1, 0, 3 };
  {
    lock_guard<mutex> lck(mtx);
    DataReady = true;
  }
  cvar.notify_one();
}

void WaitingForData() {                     // Consumer
  unique_lock<mutex> lck(mtx);
  cvar.wait(lck, []{ return DataReady; }
  SharedData[1] = 2;
}

int main( {
  thread T1(WaitingForData);
  thread T2(SetDataReady);

  T1.join();
  T2.join();

  for (auto v : SharedData)
    cout << v << "␣" <<< endl;              // 1 2 3
```

```
}
```

The consumer thread will simply wait but this `wait` won't be busy as it will keep the thread stuck until data will be available. This is a more efficient wait and synchronization mechanism then spinlock but it's usage is lock-based and we can get a better and simpler implementation with atomics:

```
vector<int> SharedData;
atomic<bool> DataReady;

void SetDataReady() {
  SharedData = { 1, 0, 3 };
  DataReady = true;                          // Atomic assignment
}

void WaitingForData() {
  while (!DataReady.load())                  // Atomic load
    sleep_for(milliseconds(5));

  SharedData[1] = 2;
}

int main() {
  thread T1(WaitingForData);
  thread T2(SetDataReady);

  T1.join();
  T2.join();

  for (auto v : SharedData)
    cout << v << "␣";                        // 1 2 3
}
```

The atomic operations define synchronization and ordering constraints which are **happens-before** relations:

- The atomic assignment *synchronizes with* the atomic load.
- The assignment to the shared data within `SetDataReady` is *sequenced before* the atomic assignment.
- The atomic load is *sequenced before* the assignment to the shared data.

A partial ordering of memory operations and producer/consumer synchronization are enough for the condition variable to be safe.

3.3 The `atomic<T>` template class

Thanks to this template class, other data types can be declared as atomics, such as integer, pointer and user-defined types. However, some requirements must be satisfied when `T` is user-defined:

- The copy-assignment operator of `T` and that of its parent class must be *trivial*.

- Virtual methods and virtual base classes are not admitted for `T`.
- The `T` class must be *bitwise comparable* in order for the CAS operations to work (for the comparison with the expected value).

The atomic operations supported by the `atomic<T>` template and their type (read, write and read-modify-write) are shown in Table 1.

Operation	R	W	RMW
test_and_set			•
clear		•	
is_lock_free	•		
load	•		
store		•	
exchange			•
compare_exchange_weak			•
compare_exchange_strong			•
fetch_add, +=			•
fetch_sub, -=			•
fetch_and, &=			•
fetch_or, |=			•
fetch_xor, ≙			•
++			•
--			•

Table 1: Atomic operations supported by the `atomic<T>` template class.

Functions like `fetch_add` perform an atomic operation and return the old value that was stored in the atomic data. Even if multiplication and division are not supported, they can be easily implemented with the other atomic operations:

```
template <typename T>
T fetch_mult(atomic<T> &shared, T m) {
  T old = shared.load();
  while (shared.compare_exchange_strong(old, old * m));

  return old;
}

int main() {
  atomic<int> SharedValue{5};
  fetch_mult(SharedValue, 5);
  cout << SharedValue << endl;                // 25
}
```

Even more complex data structures can be implemented in a lock-free way thanks to atomics:

```
template <typename T>
class LockFree_List {
  struct Node {
```

```
    T data;
    Node *next;
  };

  std::atomic<Node *> head;

public:
  void push(T const &data) {
    Node *const new_node = new Node(data);
    new_node->next = head.load();
    while (!head.compare_exchange_weak(new_node->next, new_node));
  }
};
```

4 Synchronization and Ordering

C++ supports six memory models that can be used by atomic operations, each one with its ordering constraints:

```
enum memory_order {
  memory_order_relaxed,
  memory_order_consume,
  memory_order_acquire,
  memory_order_release,
  memory_order_acq_rel,
  memory_order_seq_cst
};
```

The default model is sequential-consistency. This model is used when the operation doesn't explicitly specify a model:

```
atomic<int> Shared;

Shared.load(); // => Shared.load(memory_order_seq_cst);
```

Sequential-consistency is also the default memory model used by other languages like Java and C#.

If we want to use a weaker model we should choose it properly for atomic operations according to their type (read, write or read-modify-write) as reported by Table 2.

Memory Operation	Memory Models
R	`memory_order_acquire`, `memory_order_consume`
W	`memory_order_release`
RMW	`memory_order_acq_rel`, `memory_order_seq_cst`

Table 2: Memory models and operations.

As seen above, sequential-consistency forces a **global memory ordering**, assuring that all threads will see the same order of operations that corresponds to the *happens-before* relations found in each thread code.

The acquire-release model, on the other hand, provides a **partial memory ordering and synchronization** only between operations affecting the same atomic. This model is obtained thanks to the following enum values:

```
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
```

The relaxed model doesn't guarantee any synchronization and ordering at all, leaving to the compiler and the out-of-order execution engine of the CPU total freedom to change the program source order.

Sequential-consistency, which was defined by Leslie Lamport for the first time in 1979, is a strong memory model which forces a global ordering of memory operations such:

- The operations of *all* threads are executed in some sequential order.
- The operations of *each* thread appear in the previous sequence in the order specified by their source code.

Thus, each thread code will reflect the happens-before relations found between the instructions of its source code. This doesn't leave any opportunity of reordering thread instructions both to the compiler and the processor, even if they can be reordered interleaving the code of distinct threads. The global clock analogy also tells us that distinct thread operations cannot happend at the same clock tick. Thus, if we have two distinct threads like in Table 3, the executed code can only take one of the six orders shown in Table 4.

Thread 1	Thread 2
`x.store(1)`	`y.store(1)`
`res1 = y.load()`	`res2 = x.load()`

Table 3: Operations performed by two distinct threads.

n.1	n.2	n.3

`x.store(1)`
`res1 = y.load()`
`y.store(1)`
`res2 = x.load()`
	`x.store(1)`
	`y.store(1)`
	`res1 = y.load()`
	`res2 = x.load()`
		`x.store(1)`
		`y.store(1)`
		`res2 = x.load()`
		`res1 = y.load()`

n.4	**n.5**	**n.6**

`y.store(1)`
`x.store(1)`
`res2 = x.load()`
`res1 = y.load()`
	`y.store(1)`
	`x.store(1)`
	`res1 = y.load()`
	`res2 = x.load()`
		`y.store(1)`
		`res2 = x.load()`
		`x.store(1)`
		`res1 = y.load()`

Table 4: Possible sequential orders of operations of threads in

Despite the global order of memory operations there's still a contract:

- The programmer has to guarantee that the source code doesn't contain any potential data race.
- The system guarantees the sequentially-consistent execution of program code.

Data races can be avoided by using locking mechanisms in source code that define critical regions, where calls to `unlock` will synchronize with calls to `lock` on the same shared mutex object performed by different threads.

```
vector<int> SharedData;
mutex mtx;
bool DataReady = false;

extern void prepareData();
extern void consumeData();

void dataProducer() {
  mtx.lock();       // (1)
  prepareData();    // (2)
  dataReady = true; // (3)
  mtx.unlock();     // (4)
}

void dataConsumer() {
  mtx.lock();       // (5)
  if (DataReady)    // (6)
    consumeData();  // (7)
  mtx.unlock();     // (8)
}

int main() {
  thread T1(dataProducer);
  thread T2(dataConsumer);

  T1.join();
  T2.join();
}
```

The example features the relations shown in Table 5 which, because of their transitive nature, assure that `prepareData()` will be called before `consumeData()`

Happens-before	
Sequenced-before	**Synchronizes-with**
(1) → (2)	(1) → (2)
(2) → (3)	
(3) → (4)	
(5) → (6)	
(6) → (7)	
(7) → (8)	

Table 5: Relations found in the mutex lock/unlock example.

This, however, doesn't exclude that the synchronize-with relation might be (8) → (1), that is `dataConsumer()` is executed before `dataProducer`. In order for the consumer to call `consumeData`, a condition variable or a loop should be used to check `DataReady` until it's true. An alternative to standard mutex lock/unlock is represented by scoped locks like `lock_guard<T>`.

With the **acquire-release semantic** the ordering is no longer global as it only allows *synchronization* between a release and an acquire operation on the same atomic data, leading also to a *partial* ordering constraint which affect even non-atomic data.

Acquire-operation A read-operation like `load` or `test_and_set` (the test is actually a read even if the instruction as a whole is read-modify-write).

Release-operation A write-operation like `store` or `clear`.

The partial ordering resulting from the model comes as a result of the inner memory barriers of the two operation types:

Aquire-barrier Reads and writes cannot be moved before an acquire-operation.

Release-barrier Reads and writes cannot be moved after a release-operation.

Thus a store-release operation synchronizes with all load-acquire operations reading the stored value of the same atomic and all operations in a realising thread preceding the store-release *happen-before* all operations following the load-acquire in the acquiring thread.

An example of synchronization and partial memory ordering between three threads thanks to atomic operations and their implicit barriers:

```
vector<int> SharedData;

atom<bool> data_produced;
atom<bool> data_consumed;

void dataProducer() {
  Sharedata = { 1, 0, 3 };                              // (1)
  data_produced.store(true, memory_order_release);      // (2)
}

void dataDelivery() {
  while(!data_produced.load(memory_order_acquire));     // (3)
  data_consumed.store(true, memory_order_release);      // (4)
}

void dataConsumer() {
  while (!data_consumed.load(memory_order_acquire));    // (5)
  SharedData[1] = 2;                                    // (6)
}

int main() {
  thread T1(dataProducer);
```

```
  thread T2(dataConsumer);
  thread T3(dataDelivery);

  T1.join();
  T2.join();
  T3.join();

  for (auto v : SharedData)
    cout << v << "␣" << endl;
}
```

The atomic operations establish the relations shown in Table 6 (note that the synchronize-with relations mean that a release-operation cannot be moved before an acquire operation).

Happens-before	
Sequenced-before	**Synchronizes-with**
(1) → (2) (3) → (4) (5) → (6)	(2) → (3) (4) → (5)

Table 6: Relations found in the producer/delivery/consumer example.

As in the case of sequential consistency this could result in multiple execution orders but these will reflect the previous relations.

The acquire-release semantic allows to define critical regions. A mutex lock is indeed an acquire-operation, while a mutex unlock is a release operation. As such a lock can be implemented with a read-acquire operation and an unlock with a write-release:

```
mutex mtx;
int SharedData_1{0};
int SharedData_2{0};

void Worker() {
  mtx.lock();
  //————————————————————//
  SharedData_1 += 1;       // Critical Region
  //————————————————————//
  mtx.unlock();

  SharedData_2 += 1; // Potential race condition
}

int main() {
  thread T1(Worker);
  thread T2(Worker);

  T1.join();
  T2.join();
}
```

The unlock will thus synchronize-with the *next* lock operation using the same mutex in order to guarantee that everything before the unlock will be visible before the next lock. Code inside the critical region cannot be moved outside of it, but the code outside of it can be moved inside.

As shown in Table 7, many operations performed through the thread interface reflect the acquire-release semantic, leading to couples that can force synchronization and establishing a partial order.

	Acquire-operations	**Release-operations**
Thread	Starting	Joining
Mutex	Locking	Unlocking
Condition Variable	Waiting	Notifying

Table 7: Acquire-release semantic and thread interface operations.

Even a potentially faster spinlock can be implemented thanks to the acquire-release semantic instead of using the default sequential-consistency:

```
class Spinlock {
  atomic_flag flag;

public:
  Spinlock() : flag(ATOMIC_FLAG_INIT) {}

  void lock() {
    while(flag.test_and_set(memory_order_acquire));    // (1)
  }

  void unlock() {
    flag.clear(memory_order_release);                  // (2)
  }
};
```

The acquire-release semantic establishes a (2) $\rightarrow$ (1) synchronize-with relation and a partial ordering which assures that the effects of the operations before the release-operation will be visible to the next acquire-operation on the same atomic.

Consume-release semantic is a specialization of acquire-release but *without the ordering constraints*. It only provides the synchronization between the two operations. The key is data dependency as the following relations are established:

- *Carries-a-dependency-to* in a thread.
- *Dependency-ordered-before* between threads from the consume-operation to the release-operation.

All the operations in the releasing thread preceding the store-release will happen-before operations in the consuming thread that depend on the value loaded.

```
struct DummyStruct { int n; };
int m;
atomic<DummyStruct *> p;

void consumeFunc() {
  m = 11;
  auto DStr = new DummyStruct;
  DStr->n = 13;
  p.store(DStr, memory_order_release);
}

void releaseFunc() {
  DummyStruct *DStr;
  while (!DStr = p.load(memory_orer_consume))
    ;
  assert(p->n == 13);
  assert(m == 11);        // Data race!
}

int main() {
  thread T1(consumeFunc);
  thread T2(releaseFunc);

  T1.join();
  T2.join();
}
```

Finally, **relaxed semantic** allows only atomic operations and there's *no synchronization nor ordering constraints*. Memory operations performed by the same thread on the same memory location are not reordered with respect to the total *modification order* of the memory location, which can't be observed directly. This model should be used whenever possible but keeping in mind the following rule:

*Atomic operations with **stronger** memory orderings are used to order atomic operations with **relaxed** semantic.*

5 Singleton Pattern

A single object of a singleton class can be active at a time. Such a class can be implemented in the following way:

```
class Singleton {
public:
  static Singleton& getInstance() {
    lock_guard<<mutex> lock(mtx); // Extremely expensive!!!

    if (!instance)
```

```
      instance = new Singleton();

    return *instance
  }

private:
  Singleton() = default;
  ~Singleton() = default;

private:
  Singleton(const Singleton&) = delete;
  const Singleton operator=(const Singleton&) = delete;

private:
  static Singleton *instance = nullptr;
};
```

However, this code is extremely inefficient as each time an instance of the class is requested there will be a lock attempt of the mutex to protect the instantiation of the singleton.

We can do better by attempting the lock only under a specific condition thanks to a cheap comparison in place of the mutex lock:

```
class Singleton {
public:
  static Singleton& getInstance() {
    Singleton* sin = instance.load(); // seq-cst <---
                                                   //  |
    if (!sin) {                                    //  |
      lock_guard<mutex> lock(mtx);                 //  |
      sin = instance.load(memory_order_relaxed); //  |
      if (!sin) {                                  //  |
        sin = new Singleton();                     //  |
        instance.store(sin); // ---------------------
      }
    }

    return sin;
  }

  ...
private:
  static atomic<Singleton*> instance;
  static mutex mtx;
  ...
};
```

The atomic load is executed twice as the atomic value might change in the meanwhile and for this reason this is known as *double-check locking pattern*.

Instead of sequential-consistency, acquire-release semantic can be used in order to gain performance benefits without altering the desired synchronize-with relation:

```
class Singleton {
public:
  static Singleton& getInstance() {
    Singleton* sin =
      instance.load(memory_order_acquire); //  <------------
                                                          //      |
    if (!sin) {                                           //      |
      lock_guard<mutex> lock(mtx);                        //      |
      sin = instance.load(memory_order_relaxed);          //      |
      if (!sin) {                                         //      |
        sin = new Singleton();                            //      |
        instance.store(sin, memory_order_release); // ------
      }
    }

    return sin;
  }

  ...
private:
  static atomic<Singleton*> instance;
  static mutex mtx;
  ...
};
```

However, the *Meyer's pattern* is the fastest as it takes advantage of a C++11 guarantee according to which static variables are created in a thread-safe way:

```
class Singleton {
public:
  static Singleton& getInstance() {
    static Singleton instance;
    return instance; // Returns always the same instance
  }

private:
  Singleton() = default;
  ~Singleton() = default;
  Singleton(const Singleton&) = delete;
  const Singleton& operator=(const Singleton&) = delete;
};
```

6 Memory Models and the x86 Architecture

The memory model originally used by the ubiquitous x86 architecture was a strong ordering model known as **program ordering**. According to it, memory operations were issued by the CPU (i.e. appear on the system bus) in the same order they occurred in the instruction stream of the program assembly code. This reminds us of the sequentially-consistent model seen above.

Nowadays, however, x86 CPUs have adopted more relaxed models in order to allow certain degrees of performance optimization at run-time. Currently,

the model of choice is the so-called **processor ordering**, whose purpose is to increase instruction execution speed while guaranteeing memory coherency, even in multicore and SMP systems.

The variant of processor ordering used is defined as *write-ordered with store-buffer forwarding*. In a single-core system and for write-back cacheable memory regions, the model respects the following principles:

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes are not reordered with other writes with the exception of streaming writes executed with non-temporal move instructions and string operations.
- Reads may be reordered with older writes only to different locations.
- Reads and writes cannot be reordered with I/O instructions, locked instructions or serializing instructions.
- Writes cannot be reordered with cache line flushing instructions apart from those involving lines not containing the written location.
- Reads cannot be moved before `lfence` and `mfence` instructions.
- Writes and cache line flushing instructions cannot be moved before `lfence`, `sfence` and `mfence` instructions.
- `lfence` cannot be moved before earlier reads.
- `sfence` cannot be moved before earlier writes or cache line flushing instructions.
- `mfence` cannot be moved before reads, writes and cache line flushing instructions.

In a multi-core or an SMP system the following ordering constraints are respected:

- Each logical core follows the previous constraints seen for a single-core CPU.
- Writes from a single logical core are observed in the same order by all logical cores.
- Writes from an individual logical core are not affected by the ordering of the writes from other logical cores.
- Any two store are seen in a consistent order by all logical cores other than those performing the stores.
- Locked instructions have a total order.

Therefore, the following pseudo-code describing the assembly code of two thread functions would be correct for x86, without the need for any memory fence as those commented (which would force acquire-release semantic):

```
//
// Thread 1 code
//
A = 1;     // (W)
B = 1;     // (W)
// ----- SFENCE -----
//

//
// Thread 2 code
//
while (!B) ;  // (R)
// ----- LFENCE -----
print A;      // (R)
//
```

By default, the x86 architecture marks memory as write-back and thus the constraints seen above are reflected at run-time, meaning that each `mov` instruction automatically guarantees ordering for acquire-release consistency, without any additional instruction (like locks or memory fences). The same stands when memory is marked as write-through or uncacheable. Thus, `load` and `store` atomic operations will always be translated to plain `mov` instructions when the model is acquire-release or relaxed.

For sequential-consistency on x86 one of the following memory fences is mandatory:

- *Implicit* using a `lock`.
- *Explicit* using `lfence`, `sfence` or `mfence`.

This can be done in one of the following ways:

- A load without fence and a store with `mfence`.
- A load without fence and a `lock xchg`.
- A load with `mfence` and a store without fence.
- A `lock xadd` and a store without fence.

However there are cases in which the `mov` instructions do not automatically guarantee acquire-release memory ordering, such as:

- When accessing memory which is marked as *write-combined* in the page table (for POSIX systems this is done through the `ioremap_wc` system call), which guarantees only acquire consistency.
- When performing stores which cannot be reordered between them (i.e. string operations and cacheability control instructions as not-temporal moves and `clflush`).

In these circumstances an `sfence` instruction must be inserted between two writes to the same location in order to guarantee acquire-release consistency.

Remember that when writing C++ code not only the CPU but even the compiler can reorder instructions and the only way to affect it is by specifying one of the barriers supported by the atomic operations. Otherwise volatile inline assembly code must be written.