【Sequenced-before】

Within the same thread, evaluation A may be sequenced-before evaluation B, as described in evaluation order.

一个语句在另一个语句之前

【Carries dependency（传递依赖）】

Within the same thread, evaluation A that is sequenced-before evaluation B may also carry a dependency into B (that is, B depends on A), if any of the following is true

1) The value of A is used as an operand of B, except （A是B的操作数除了）

a) if B is a call to std::kill\_dependency

b) if A is the left operand of the built-in &&, ||, ?:, or , operators.

2) A writes to a scalar object M, B reads from M

3) A carries dependency into another evaluation X, and X carries dependency into B

【Modification order】

All modifications to any particular atomic variable occur in a total order that

is specific to this one atomic variable.The following four requirements are guaranteed for all atomic operations:

1) Write-write coherence:

If evaluation A that modifies some atomic M (a write) happens-before evaluation B that modifies M,

then A appears earlier than B in the modification order of M

A在M上的写入 比B在M上的写入早 A就在B的 Modification order之前

2) Read-read coherence:

if a value computation A of some atomic M (a read) happens-before a value computation B on M,

and if the value of A comes from a write X on M, then the value of B is either the value stored by X,

or the value stored by a side effect Y on M that appears later than X in the modification order of M.

A的读取发生在B对M的读取之前 如果A来自于一个X对M的写 则B的值要么来自于X的写 要么来自于一个M的副作用Y在modification order晚于X

3) Read-write coherence:

if a value computation A of some atomic M (a read) happens-before an operation B on M (a write),

then the value of A comes from a side-effect (a write) X that appears earlier than B in the modification order of M

A对M的读早于B对M的写， A的值来自于一个副作用的写X 那么这个X在 modification order 上早于B

4) Write-read coherence:

if a side effect (a write) X on an atomic object M happens-before a value computation (a read) 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

副作用X对M的写 早于B对M读 ，B的值要么来自于X，要么来自于副作用Y 在modification order 跟随X

**SEQUENTIALLY CONSISTENT ORDERING（顺序一致性）**

**【Sequentially-consistent ordering】**

Atomic operations tagged memory\_order\_seq\_cst not only order memory the same way as release/acquire ordering

(everything that happened-before a store in one thread becomes a visible side effect in the thread that did a load), but also establish a single total modification order of all atomic operations that are so tagged.

all threads must see the same order of operations，It also means that operations can’t be reordered; if your code has one operation before another in one thread, that ordering must be seen by all other threads. From the point of view of synchronization, a sequentially consistent store synchronizes with a sequentially consistent load of the same variable that reads the value stored. This provides one ordering constraint on the operation of two (or more) threads, but sequential consistency is more powerful than that.Any sequentially consistent atomic operations done after that load must also appear after the store to other threads in the system using sequentially consistent atomic operations所有的读操作必须出现在写操作之后。所有线程应用同一个顺序

Sequential consistency is the most straightforward and intuitive ordering, but it’s also the most expensive memory ordering because it requires global synchronization between all threads。

**NON - SEQUENTIALLY CONSISTENT MEMORY ORDERINGS（非顺序一致性）**

This means that different threads can see different views of the same operations, and any mental model you have of operations from different threads neatly interleaved one after the other must be thrown away It’s not just that the compiler can reorder the instructions. Even if the threads are running the same bit of code, they can disagree on the order of events because of operations in other threads in the absence of explicit ordering constraints, because the different CPU caches and internal buffers can hold different values for the same memory. It’s so important. **I’ll say it again: threads don’t have to agree on the order of events**

**Not only do you have to throw out mental models based on interleaving operations, you also have to throw out mental models based on the idea of the compiler or processor reordering the instructions. *In the absence of other ordering constraints, the only requirement is that all threads agree on the modification order of each individual variable.***

**memory\_order\_relaxed**

Relaxed operation: there are no synchronization or ordering constraints imposed on other reads or writes, only this operation's atomicity is guaranteed (see Relaxed ordering below)

Operations on the same variable within a single thread still obey happens-before relationships, but there’s almost no requirement on ordering relative to other threads

Without any additional synchronization, the **modification order** of each variable is the only thing shared between threads that are using memory\_order\_relaxed

**遵循 modification order 并不存在一个全局的顺序，相同atomic 变量遵循 happenes-before 关系 相同线程不同的atomic变量可以被重新排序。不同线程间没有固定顺序**

**memory\_order\_consume**

A load operation with this memory order performs a consume operation on the affected memory location: no reads or writes in the current thread dependent on the value currently loaded can be reordered before this load. Writes to data-dependent variables in other threads that release the same atomic variable are visible in the current thread.On most platforms, this affects compiler optimizations only (see Release-Consume ordering below)

**ACQUIRE - RELEASE ORDERING**

【memory\_order\_acquire。

A load operation with this memory order performs the acquire operation on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread (see Release-Acquire ordering below)

memory\_order\_release

A store operation with this memory order performs the release operation: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable (see Release-Acquire ordering below) and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic (see Release-Consume ordering below).

memory\_order\_acq\_rel

A read-modify-write operation with this memory order is both an acquire operation and a release operation. No memory reads or writes in the current thread can be reordered before or after this store. All writes in other threads that release the same atomic variable are visible before the modification and the modification is visible in other threads that acquire the same atomic variable. memory\_order\_seq\_cst A load operation with this memory order performs an acquire operation, a store performs a release operation, and read-modify-write performs both an acquire operation and a release operation, plus a single total order exists in which all threads observe all modifications in the same order (see Sequentially-consistent ordering below)】

**遵循 modification order 并不存在一个全局的顺序，相同atomic 变量遵循 happenes-before 关系 相同线程不同的atomic变量可以被重新排序。不同线程间 遵循 store acquire 的顺序**

【Release-Acquire ordering】

If an atomic store in thread A is tagged memory\_order\_release and an atomic load in thread B from the same variable is tagged memory\_order\_acquire,

all memory writes (non-atomic and relaxed atomic) that happened-before the atomic store from the point of view of thread A, become visible side-effects in thread B.

That is, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory.

The synchronization is established only between the threads releasing and acquiring the same atomic variable.

Other threads can see different order of memory accesses than either or both of the synchronized threads.

On strongly-ordered systems — x86, SPARC TSO, IBM mainframe, etc.— release-acquire ordering is automatic for the majority of operations.

No additional CPU instructions are issued for this synchronization mode; only certain compiler optimizations are affected

(e.g., the compiler is prohibited from moving non-atomic stores past the atomic store-release or

performing non-atomic loads earlier than the atomic load-acquire).

On weakly-ordered systems (ARM, Itanium, PowerPC), special CPU load or memory fence instructions are used.

Mutual exclusion locks, such as std::mutex or atomic spinlock, are an example of release-acquire synchronization:

when the lock is released by thread A and acquired by thread B, everything that

took place in the critical section (before the release) in the context of thread A has to be visible to thread B (after the acquire) which

is executing the same critical section.

Acquire-release ordering is a step up from relaxed ordering; **there’s still no total order**

of operations, **but it does introduce some synchronization**. Under this ordering model,

atomic loads are acquire operations ( memory\_order\_acquire ), atomic stores are release

operations ( memory\_order\_release ), and atomic read-modify-write operations (such

Figure 5.5 The notebook for

the man in the cubicle as fetch\_add() or exchange() ) are either acquire, release, or both ( memory\_order\_acq\_rel )

***A release operation synchronizes-with an acquire operation that reads the value written*. This means that different threads can still see different orderings, but these orderings are restricted.**

【TIPs】

If you use read-modify-write operations, it’s important to pick which semantics you

desire. In this case, you want both acquire and release semantics, so memory\_order\_acq\_rel is appropriate, but you can use other orderings too. **A fetch\_sub operation with memory\_order\_acquire semantics doesn’t synchronize with anything, even though it stores a value, because it isn’t a release operation**. Likewise, **a store can’t synchronize with a fetch\_or with memory\_order\_release semantics, because the read part of the fetch\_or isn’t an acquire operation**. Read-modify-write operations with memory\_order\_acq\_rel semantics behave as both an acquire and a release, so a prior store can synchronize with such an operation, and it can synchronize with a subsequent load, as is the case in this example

***your acquire and release operations have to be on the same variable to ensure an ordering.***

**数据依赖**

**The concept of a data dependency is relatively straightforward: there is a data**

**dependency between two operations if the second one operates on the result of the**

**first. There are two new relations that deal with data dependencies: *dependency-ordered-before* and *carries-a-dependency-to*. Like sequenced-before, carries-a-dependency-to applies strictly within a single thread and models the data dependency between operations; if the result of an operation (A) is used as an operand for an operation (B), then A carries a dependency to B. If the result of operation A is a value of a scalar type such as an int , then the relationship still applies if the result of A is stored in a variable, and that variable is then used as an operand for operation B. This operation is also transitive, so if A carries a dependency to B, and B carries a dependency to C, then A carries a dependency to C. On the other hand, the *dependency-ordered-before relationship can apply between threads*. It’s introduced by using atomic load operations tagged with memory\_order\_consume . This is a special case of memory\_order\_acquire that limits the synchronized data to direct dependencies; a store operation (A) tagged with memory\_order\_release, memory\_order\_acq\_rel, or memory\_order\_seq\_cst is dependency-ordered-before a load operation(B)tagged with memory\_order\_consume if the consume reads the value stored. This is** **as opposed to(截然相反) the synchronizes-with relationship you get if the load uses memory\_order\_acquire . If this operation (B) then carries a dependency to some operation (C), then A is also dependency-ordered-before C.**

**This wouldn’t do you any good for synchronization purposes if it didn’t affect the**

**inter-thread happens-before relation, but it does: if A is dependency-ordered-before**

**B, then A also inter-thread happens-before B**

**If the store is tagged with memory\_order\_release , memory\_order\_acq\_rel , or memory\_order\_seq\_cst, and the load is tagged with memory\_order\_consume , memory\_order\_acquire, or memory\_order\_seq\_cst , and each operation in the chain loads the value written by the previous operation, then the chain of operations constitutes a release sequence and the initial store synchronizes with (for memory\_order\_acquire or memory\_order\_seq\_cst ) or is dependency-ordered-before (for memory\_order\_consume ) the final load. Any atomic read-modify-write operations in the chain can have any memory ordering (even memory\_order\_relaxed )**