并发进阶
Advanced Concurrency

现代C++基础
Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- **Memory Order Basics**

- **Atomic Variable Details**

- **Advanced Memory Order**

- **Coroutine**

Initially, I think that most of the committee members underestimated the problem. We knew that Java had a good memory model [Pugh 2004] and hoped to adopt that. I was highly amused to find that representatives from Intel and IBM effectively vetoed that idea by pointing out that by adopting the Java memory model for C++ we would slow down all JVMs by a factor of at least two. Consequently, to preserve the performance of Java, we had to adopt a far more complex model for C++. Ironically and predictably, C++ was then criticized for having a more complicated memory model than Java.

最开始，我想大多数委员都小瞧了这个问题。我们知道 Java 有一个很好的内存模型 [Pugh 2004]，并曾希望采用它。令我感到好笑的是，来自英特尔和 IBM 的代表坚定地否决了这一想法，他们指出，如果在 C++ 中采用 Java 的内存模型，那么我们将使所有 Java 虚拟机的速度减慢至少一半。因此，为了保持 Java 的性能，我们不得不为 C++ 采用一个复杂得多的模型。可以想见而且讽刺的是，C++ 此后因为有一个比 Java 更复杂的内存模型而受到批评。

Advanced Concurrency

Memory Order Basics

“Even with C++11 support, I consider lock-free programming expert-level work.”
-- Bjarne Stroustrup, HoPL4, P33

Advanced Concurrency

- Memory Order Basics
  - Overview
  - Sequentially consistent model
  - Acquire-release model
  - Relaxed model

  - There also exists consume-release model, but since it’s very difficult for users to annotate and for compilers to analyze better optimizations, all compilers strengthen consume-release model to acquire-release model.
    - C++20: [*Note 1*: Prefer `memory_order::acquire`, which provides stronger guarantees than `memory_order::consume`. Implementations have found it infeasible to provide performance better than that of `memory_order::acquire`. Specification revisions are under consideration. — *end note*]
    - C++26: consume operations are deprecated.

Defang and deprecate
memory_order::consume

Doc. No: P3475R1
Contact: Hans Boehm (hboehm@google.com)
Audience: LEWG, CWG, LWG
Date: Jan 11, 2025
Target: C++26

Memory order

- Current programming world stands on the foundation of sequential execution…
  - Compiler / JIT may do aggressive optimization…
    - Here we will "cache" global variables to registers, and eliminate redundant expressions (i.e. `b = addend + 1`).

```
int a, b, addend;

void test()
{
    b = addend + 1;
    a = b - 5;
    b = addend + 3;
```

```
void test(int addend)
{
    int tempb = addend;     // mov esi, DWORD PTR addend[rip]
    int tempa = tempb - 4;  // lea edi, [rsi - 4]
    tempb += 4;             // add esi, 4

    a = tempa;              // mov DWORD PTR a[rip], edi
    b = tempb;              // mov DWORD PTR b[rip], esi
```

  - Processors may do out-of-order execution and speculative computation…
  - Each processor may have its own L1/L2 cache…

Memory order

- These optimizations are smart and correct in sequential world, but when it comes to parallelism, some assumptions are not that intuitive…

```
int a, b, addend;

void test()
{
    b = addend + 1;
    a = b - 5;
    b = addend + 3;
```

```
void test(int addend)
{
    int tempb = addend;     // mov esi, DWORD PTR addend[rip]
    int tempa = tempb - 4;  // lea edi, [rsi - 4]
    tempb += 4;             // add esi, 4

    a = tempa;              // mov DWORD PTR a[rip], edi
    b = tempb;              // mov DWORD PTR b[rip], esi
```

- What if there is another thread that modifies addend here?
  - b can be something other than tempb + 4, but compiler optimizations make it impossible.

Memory order

- Among so many compiler optimizations, processor ISA regulations, cache coherence protocols…
  - We need to find a way to unify "as-if" behaviors by abstraction!
- That is what *memory order* for in C++.
  - Three types of memory order:
    - Sequentially consistent model (`seq_cst`)
    - Acquire-release model (`acq_rel`)
    - Relaxed model (`relaxed`)
  - BTW, Rust has completely same regulations as C++.

Rust pretty blatantly just inherits the memory model for atomics from C++20. This is not due to this model being particularly excellent or easy to understand. Indeed, this model is quite complex and known to have several flaws. Rather, it is a pragmatic concession to the fact that *everyone* is pretty bad at modeling atomics. At very least, we can benefit from existing tooling and research around the

Atomics - The Rustonomicon

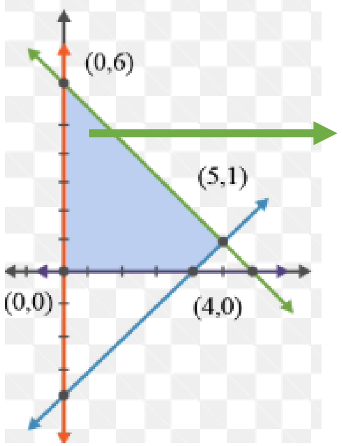
[1]: A Concurrency Semantics for Relaxed Atomics that Permits Optimisation and Avoids Thin-air Executions | POPL'16, Jean & Peter from Univ. of Cambridge
POPL is Top Academic Conference in Programming Language Design.

Memory order

- But, how to describe memory order is still **an unsolved problem** even in academia (even `seq_cst` model has bug fix in C++20).
  - And C++ is pioneer in this field, so the standard has been revised nearly in every version.
  - But normally this is defect in theoretical model; real-world behaviors are not severely affected.
- The key problem is that memory order is *axiomatic*[1], which is rather weak and cannot exactly describe what we want.
  - Memory order gives constraints, and every outcome that can fulfill the constraint is a valid solution.
    - While some solutions are not really valid…we'll see them later.

What memory order regulates:

$$\textit{constraints:}\ \begin{cases} x + y \geq 6 \\ x - y \geq 4 \\ x \geq 0 \\ y \geq 0 \end{cases}$$



Results that memory order allows

Memory order

Formally, this is regulated by RR/RW/WR/WW coherence in standard; we rephrase it here.

- There are some intuitive basic regulations in memory model.

1. Modification order: for a **single *atomic*** variable, all threads see the same operation sequences.
   - So can `r1 == 1 && r2 == 2 && r3 == 2 && r4 == 1`?
   - No!
   - Reason: r4 cannot read value newer than r3, and r2 cannot read value newer than r1.
     - `r1 == 1 && r2 == 2`: 2 is newer than 1;
     - `r3 == 2 && r4 == 1`: 1 is newer than 2; Conflict!
     - Compilers are not allowed to reorder.
   - But, operations for different atomic variables may have different orders in different threads.

```
-- Initially --
std::atomic<int> x{0};

-- Thread 1 --
x.store(1);

-- Thread 2 --
x.store(2);

-- Thread 3 --
int r1 = x.load();
int r2 = x.load();

-- Thread 4 --
int r3 = x.load();
int r4 = x.load();
```

Memory order

2. Sequenced before: we've covered evaluation order previously…

Expression

- Then, it's order of expression evaluation that computes the whole tree.
  - It is only determined that before the evaluation of root, the left child and the right child will be evaluated first; the order is **unspecified**.
  - e.g. `f1()` $+_1$ `f2()` $+_2$ `f3()`, it's `root(`$+_2$`) -> lChild(f1() + f2()) -> rChild(f3())`, while `lChild` is `root(`$+_1$`) -> lChild(f1()) -> rChild(f2())`;
    - We can know before $+_1$ is evaluated, lChild and rChild is first evaluated.
    - However, you can evaluate in the sequence of:
      - `lChild` evaluates `f1()`
      - `rChild` evaluates `f3()`, gets the value.
      - `lChild` evaluates `f2()`, gets the value.
      - This still obeys our rules, e.g. `f1()` and `f2()` evaluated before `lChild`.
    - So if we output `a` in `f1()`, `b` in `f2()`, `c` in `f3()`, any permutation of `abc` is possible!
  - To sum up, evaluation order is hugely determined by how compiler computes the tree.

Memory order

- So if an evaluation `A` definitely computes before another one `B`, then we say `A` is ***sequenced before*** `B`.
  - For example, for different statements.

```
a += 1; // #1 happens before #2
b += 2; // #2
```

  - In the same statement:

```
a += 1, b += 2; // a += 1 is evaluated first
                // then it's sequenced before 'b += 2'
// Yet another smelly example.
a += b++; // b++ is evaluated first since C++17,
          // so here 'b++' sequenced before 'a +='.
```

  - And function parameters are *indeterminately sequenced* since C++17, so there is some order but it’s unspecified;
  - And some evaluations are not regulated at all, which means they’re *unsequenced* (e.g. `a = b++ + b` is UB, since `b++` and `b` are unsequenced while `b++` has side effect).
- Again, such order is in the sequential view…

Memory order

Data races occur when non-atomic operations on the same memory location do NOT have some certain happens-before relationship.

3. Happens before: in parallel world, which evaluation is executed first is regulated by ***happens-before***.
   - If A is sequenced before B, then A happens before B (single-thread case);
   - If A ***synchronizes with*** B, then A happens before B (inter-thread case);
   - Or A happens before B & B happens before C, then A happens before C.

   - For non-atomic variables, only when A happens before B will effects of A be visible to B.
     - So compilers can do aggressive optimizations, as long as they aren't visible.
   - For atomic variables, HB order is part of MO; if two operations have no HB relationship, then their order in MO is also random.
     - Namely, if B doesn't happen before A, then effects of A may be visible to B.
   - Memory order mainly regulates such "synchronize-with" relationship.

Note: actually, what we teach here is happens-before since C++26; before that (since C++20) this is called simply-happens-before, but it's equivalent to happens-before (since C++11) when no consume operation is involved (and again, we've said that consume operations are never implemented).

Advanced Concurrency

- Memory Order Basics
  - Overview
  - Sequentially consistent model
  - Acquire-release model
  - Relaxed model

Sequential Consistency

- In real world, all events are sequenced in some way, and all observers will see the same sequence.
- Similarly, we may think operations to have some total order, and all threads observe the same order.
  - This is the core of sequentially consistent model!
  - Back to our example:

```
x.store(1)
x.store(2)
x.load()    x.load()
x.load()    x.load()
```

```
x.load()      r1=0
x.store(1)
x.load()      r3=1
x.store(2)
x.load()      r4=2
x.load()      r2=2
```

Interleaving them randomly,
we get a total order.

```
-- Initially --
std::atomic<int> x{0};

-- Thread 1 --
x.store(1);

-- Thread 2 --
x.store(2);

-- Thread 3 --
int r1 = x.load();
int r2 = x.load();

-- Thread 4 --
int r3 = x.load();
int r4 = x.load();
```

Sequential Consistency

- Formally, when an atomic load operation B loads a value that's stored by an atomic store operation A, then A synchronizes with B.
  - Then all previous outcomes are visible since B.
- For example:

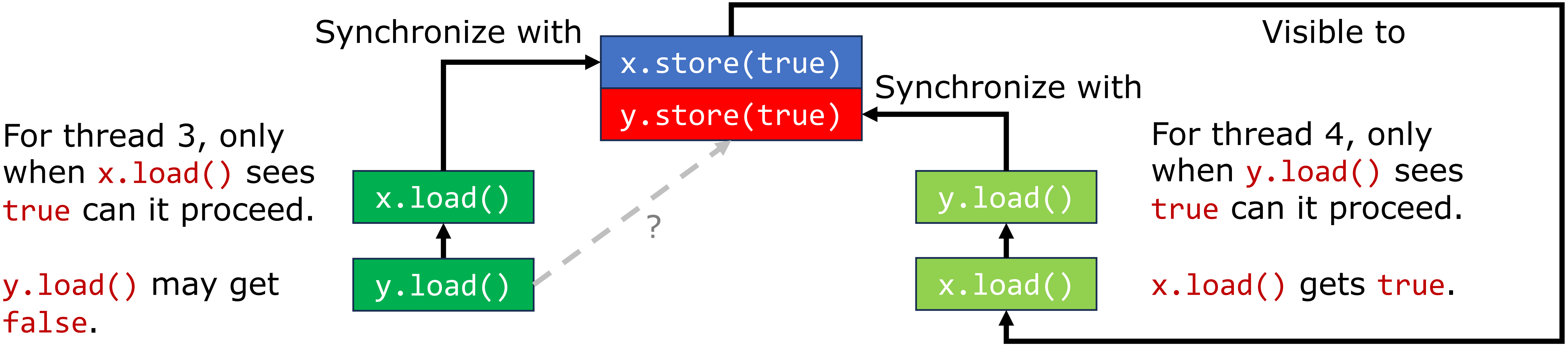
```cpp
std::atomic<bool> x{false}, y{false};
std::atomic<int> z{0};

void write_x() { x.store(true); }
void write_y() { y.store(true); }
void read_x_then_y()
{
    while (!x.load());
    if (y.load())
        ++z;
}
void read_y_then_x()
{
    while (!y.load());
    if (x.load())
        ++z;
}
```

```cpp
int main()
{
    { // 等四个线程全部结束。
        std::jthread a{ write_x }, b{ write_y }, c{ read_x_then_y },
                     d{ read_y_then_x };
    }
    assert(z.load() != 0);
}
```

Can this assert fire?

- No, `z.load()` is always non-zero.
- Reason: there is a total order, so either `x.store(true)` or `y.store(true)` occurs first.
  - Let's assume `x.store(true)` happens first since it's completely symmetric.



- Synchronization is implicitly established through reading value.
- Note: `x.store(true)` and `y.store(true)` **do NOT have happens-before relationship**; the order is imposed by total order.

seq_cst model actually uses strongly-happens-before relationship but here they're equivalent; we'll cover it in the next section.

Sequential Consistency

- Note 2: start of threads & joining threads will also establish synchronize-with relationship with function start & return.
  - So here thread joining happens before `z.load()`, and function return happens before thread joining, and `++z` happens before function return. Thus `z.load()` can get 1 or 2 correctly.
- Note 3: operations on atomic variables are indivisible (and thus prevent data races), which is not affected by memory order.
  - Called *atomicity*.
  - Our example in the last lecture:
    - If `a` is atomic variable, then lock protection is not needed.

```
void Inc(int& a, std::mutex& mut) {
    for (int i = 0; i < 100000; i++)
    {
        std::lock_guard _{ mut };
        a++;
    }
}
```

Advanced Concurrency

- Memory Order Basics
  - Overview
  - Sequentially consistent model
  - Acquire-release model
  - Relaxed model

Acquire-release Model

- In many architectures like RISC-V, ARM and Power, such total-order assumption is quite expensive, while they support weaker model better.
  - Acquire-release is a commonly supported order!

- So what does acquire-release model guarantee?
  - Only read operations can be “acquire”, and only write operations can be “release”.
  - For an acquire operation `B`, if it reads the value from a release operation `A`, **then `A` synchronizes with `B`** (and thus `A` happens before B).
  - **There is no total order**.

Acquire-release Model

- For example:

Sequenced before store, thus happens before store.

Only when ptr loads some value will the program proceed, then store synchronizes with load (and thus happens before load).

Sequenced after load, thus load happens before asserts.

```
std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire)))
        ;
    assert(*p2 == "Hello"); // never fires
    assert(data == 42); // never fires
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

Through three happens-before, we know that data is always 42.

Acquire-release Model

- On the other hand, since it doesn't have total order:

```
void write_x() { x.store(true, std::memory_order_release); }
void write_y() { y.store(true, std::memory_order_release); }
void read_x_then_y()
{
    while (!x.load(std::memory_order_acquire));
    if (y.load(std::memory_order_acquire))
        ++z;
}
void read_y_then_x()
{
    while (!y.load(std::memory_order_acquire));
    if (x.load(std::memory_order_acquire))
        ++z;
}
```

Store of x and y have no happens-before relationship.

Here we only know that x is true (synchronize-with), while `y.load` and `y.store` don't necessarily have happens-before relationship.

Similarly, `x.load` and `x.store` don't necessarily have happens-before relationship.

Thus, `z.load()` can be 0 here.

Acquire-release Model

- Another example for transitivity:
  - SB(#0, #1)
  - SW(#1, #2)
    - As only when #2 reads true can thread_2 proceed.
  - SB(#2, #3)
  - SW(#3, #4)
  - SB(#4, #5)
- Thus we know HB(#0, #5).

Obviously, acquire-release model can be used to implement spinlock.

```
int data = 0;
std::atomic<bool> sync1{ false },sync2{ false };

void thread_1()
{
    data = 442;                                        // #0
    sync1.store(true,std::memory_order_release);       // #1
}

void thread_2()
{
    while(!sync1.load(std::memory_order_acquire)); // #2
    sync2.store(true,std::memory_order_release);       // #3
}

void thread_3()
{
    while(!sync2.load(std::memory_order_acquire)); // #4
    assert(data == 442);                               // #5
}
```

Acquire-release Model

- By happens-before relationship, acquire-release model implicitly disables compiler reorder optimization.
  - An acquire operation `B` may happen after another release operation `A`…
    - If a compiler reorders statements `S1` after `B` to before `B`;
    - Or if a compiler reorders statements `S2` before `A` to after `A`;
    - Then `S1` may fail to observe results in `S2`.
  - Thus, acquire & release offers a **one-way instruction barrier** implicitly.
    - All operations that will cause side effects (that may be used by another threads) cannot go below beyond a release operation;
    - All operations that may rely on side effects cannot go above beyond an acquire operation.
  - Intuitively, acquire-release forms some critical section; you cannot move out code in between.

Advanced Concurrency

- Memory Order Basics
  - Overview
  - Sequentially consistent model
  - Acquire-release model
  - Relaxed model

Relaxed Model

- Sometimes we may want even weaker order…
  - That is, we only need to maintain atomicity; no synchronize-with relationship is needed.
  - This is relaxed model.
- For example:

```cpp
std::atomic<int> x{0}, y{0};

void read_y_then_write_x(int& r1)
{
    r1 = y.load(std::memory_order_acquire); // #1
    x.store(r1, std::memory_order_release); // #2
}

void read_x_then_write_y(int& r2)
{
    r2 = x.load(std::memory_order_acquire); // #3
    y.store(42, std::memory_order_release); // #4
}
```

```cpp
int main()
{
    int r1 = 0, r2 = 0;
    std::jthread{ read_y_then_write_x, std::ref(r1) },
                { read_x_then_write_y, std::ref(r2) };
    assert(!(r1 == 42 && r2 == 42));
}
```

Exercise: Can this assert fire?

Relaxed Model

```cpp
std::atomic<int> x{0}, y{0};

void read_y_then_write_x(int& r1)
{
    r1 = y.load(std::memory_order_acquire); // #1
    x.store(r1, std::memory_order_release); // #2
}

void read_x_then_write_y(int& r2)
{
    r2 = x.load(std::memory_order_acquire); // #3
    y.store(42, std::memory_order_release); // #4
}
```

- No!
- Assuming that `r1 == 42`,
  - Then #1 reads value from #4, and acquire-release model makes SW(#4, #1).
  - And SB(#3, #4), SB(#1, #2), thus we know HB(#3, #2).
  - Thus, effects of #2 are not visible to #3, and r2 is definitely 0.
- Then what about relaxed model?
  - This assertion may fire…
  - That is, `r1 == 42 && r2 == 42` may be `true`.

```cpp
void read_y_then_write_x(int& r1)
{
    r1 = y.load(std::memory_order_relaxed); // #1
    x.store(r1, std::memory_order_relaxed); // #2
}

void read_x_then_write_y(int& r2)
{
    r2 = x.load(std::memory_order_relaxed); // #3
    y.store(42, std::memory_order_relaxed); // #4
}
```

Relaxed Model

```cpp
void read_y_then_write_x(int& r1)
{
    r1 = y.load(std::memory_order_relaxed); // #1
    x.store(r1, std::memory_order_relaxed); // #2
}

void read_x_then_write_y(int& r2)
{
    r2 = x.load(std::memory_order_relaxed); // #3
    y.store(42, std::memory_order_relaxed); // #4
}
```

- Since relaxed model doesn't establish any synchronize-with relationship…
- Remember our effect rules?
    - For atomic variables, HB order is part of MO; if two operations have no HB relationship, then their order in MO is also random.
        - Namely, if B doesn't happen before A, then effects of A may be visible to B.
    - So here #1 doesn't happen before #4, then effects of #4 can be read by #1 so `r1 == 42` can be true.
        - And #1 happens before #2, so x can store 42.
    - And #3 doesn't happen before #2, then effects of #2 can be read by #3 so `r2 == 42` can be true.
- Thus, `r1 == 42 && r2 == 42` can be true.

Relaxed Model

- Note 1: again, we emphasize that there is no total order.
  - If there is, then in thread 2 SB(#3, #4) prevents any possible order to make `r2 == 42`.
  - In practice, compilers are allowed to reorder #3 and #4, since destroying such HB doesn't affect any visible effects.
- Note 2: this outcome doesn't violate modification order constraint of a single atomic variable.

All threads see this same modification order.

x	0	r1	#3 can read r1, so it can be 42.
y	0	42	#1 can read 42, so r1 can be 42.

Notice that here it's ***can*** instead of ***must***; #1 and #3 can read older values.

Relaxed Model

```cpp
std::atomic<int> x{0}, y{0}, z{0};
std::atomic<bool> start{false};

constexpr unsigned int loop_num = 10;
struct ValueStatus { int x, y, z; };

using ValueContainer = std::array<ValueStatus, loop_num>;
```

- Another complex example:

```cpp
void increment(std::atomic<int>* var, ValueContainer* values)
{
    start.wait(false);
    for (unsigned int i = 0; i < loop_num; i++)
    {
        values[i].x = x.load(std::memory_order_relaxed);
        values[i].y = y.load(std::memory_order_relaxed);
        values[i].z = z.load(std::memory_order_relaxed);

        var->store(i + 1, std::memory_order_relaxed);
    }
}

void read_status(ValueContainer* values)
{
    start.wait(false);
    for (unsigned int i = 0; i < loop_num; i++)
    {
        values[i].x = x.load(std::memory_order_relaxed);
        values[i].y = y.load(std::memory_order_relaxed);
        values[i].z = z.load(std::memory_order_relaxed);
    }
}
```

Like a spinlock, covered later.

```cpp
int main()
{
    std::array<ValueContainer, 5> values;
    {
        std::jthread a{ increment, &x, &values[0] }, b{ increment, &y,
&values[1] }, c{ increment, &z, &values[2] };
        std::jthread d{ read_status, &values[3] }, e{ read_status, &values[4] };

        start.store(true);
        start.notify_all();
    }

    for (const auto& cont: values)
    {
        std::print("[");
        for (auto val : cont)
            std::print("({}, {}, {}) ", val.x, val.y, val.z);
        std::println("]");
    }
}
```

All threads start now.

Relaxed Model

- So what it does is:
  - Three threads, with each one only modifying one of the atomic variables, and reading all of them;
  - Two threads that only read all atomic variables.
- It can only guarantee that:
  - The thread that modifies the variable will see it increases one by one, constrained by happens-before relationship.
    - For example, `values[0]` will have `(0, …, …)`, `(1, …, …)`, `…`, `(9, …, …)`.
  - And constrained by single-atomic modification order, other variables that are not modified by itself will have non-decreasing values.
    - That is, once a value is read (not necessarily the newest), values older than it cannot be read.

15 [*Note 16*: 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. — *end note*]

Relaxed Model

- Courtesy of *C++ Concurrency in Action, 2nd ed.* by Anthony Williams.

One possible output from this program is as follows:

```
(0,0,0),(1,0,0),(2,0,0),(3,0,0),(4,0,0),(5,7,0),(6,7,8),(7,9,8),(8,9,8),
(9,9,10)
(0,0,0),(0,1,0),(0,2,0),(1,3,5),(8,4,5),(8,5,5),(8,6,6),(8,7,9),(10,8,9),
(10,9,10)
(0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4),(0,0,5),(0,0,6),(0,0,7),(0,0,8),
(0,0,9)
(1,3,0),(2,3,0),(2,4,1),(3,6,4),(3,9,5),(5,10,6),(5,10,8),(5,10,10),
(9,10,10),(10,10,10)
(0,0,0),(0,0,0),(0,0,0),(6,3,7),(6,5,7),(7,7,7),(7,8,7),(8,8,7),(8,8,9),
(8,8,9)
```

Relaxed Model

- Relaxed model may cause very astonishing results, so it needs to be used with extreme caution…
  - Usually it either cooperates with other sync operations (like acquire-release model)…
  - Or it’s used to do very simple job that only needs atomicity.
    - For example, `std::shared_ptr` has a counter to count its copies; when all copies are destructed, the memory is finally freed.
    - We can check the shared count by `.use_count()`, which is normally a relaxed load since it doesn’t need to participate in synchronization.