Relaxed Semantic

This lesson gives an overview of relaxed semantic which is used in C++ for concurrency.

WE'LL COVER THE FOLLOWING

The relaxed semantic is the other end of the spectrum. It's the weakest of all memory models and only guarantees that the operations on the same atomic data type in the same thread won't be reordered. That guarantee is called modification order consistency. Other threads can see these operations in a different order.

No Synchronization & Ordering constraints?

This is quite easy; if there are no rules, we cannot violate them. But that is too easy, as the program should have *well-defined* behavior. In particular, this means that data races are not allowed. To guarantee this you typically use synchronization and ordering constraints of stronger memory models to control operations with relaxed semantic. How does this work? A thread can see the effects of another thread in arbitrary order, so you have to make sure there are points in your program where all operations on all threads get synchronized.

A typical example of an atomic operation, in which the sequence of operations doesn't matter, is a counter. The key observation for a counter is not in which order the different threads increment the counter; it's that all increments are atomic and all threads' tasks are done at the end. Have a look at the following example.

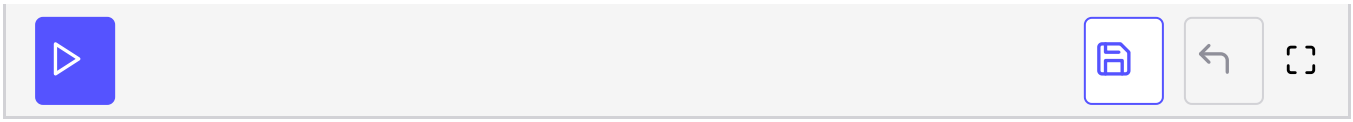
```
// relaxed.cpp

#include <vector>
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<int> count = {0};

void add()
{
    for (int n = 0; n < 1000; ++n) {
        count.fetch_add(1, std::memory_order_relaxed);
    }
}

int main()
{
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n) {
        v.emplace_back(add);
    }
    for (auto& t : v) {
        t.join();
    }
    std::cout << "Final counter value is " << count << '\n';
}
```



The three most interesting lines are 13, 24, and 26. In line 13, the atomic number `count` is incremented using the relaxed semantic, so we have a guarantee that the operation is atomic. The `fetch_add` operation establishes an ordering on `count`. The function `add` (lines 10 - 15) is the work package of the threads. Each thread gets its work package on line 21. Thread creation is one synchronization point. The other one being `t.join()` on line 24.

The creator thread synchronizes with all its children in line 24. It waits with the `t.join()` call until all its children are done. `t.join()` is the reason that the results of the atomic operations are published. To say it more formally, `t.join()` is a release operation.

Conclusion

In conclusion, there is a *happens-before* relation between the increment operation in line 13 and the reading of the counter `count` in line 26. The result is that the program always returns 10000. Boring? No, it's reassuring!

A typical example of an atomic counter which uses the relaxed semantic is the reference counter of `std::shared_ptr`. This will only hold for the increment operation. The key property for incrementing the reference counter is that the operation is atomic; the order of the increment operations does not matter. This will not hold for the decrement of the reference counter. These operations need an acquire-release semantic for the destructor.

> **The `add` algorithm is wait_free**
>
> Have a closer look at function `add` in line 10. There is no synchronisation involved in the increment operation (line 13). The value 1 is just added to the atomic `count`. Therefore, the algorithm is not only ***lock-free*** but it is also ***wait-free***.

The key idea of `std::atomic_thread_fence` is to establish synchronization and ordering constraints between threads without any atomic operations.