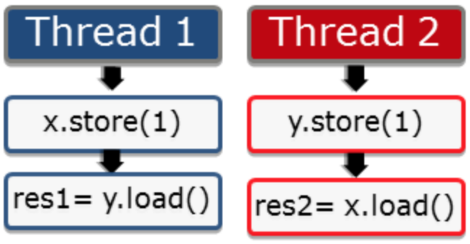
Weak Memory Model

This lesson gives a brief overview of the weak memory model regarding concurrency in C++.



Let's refer once more to the contract between the programmer and the system.

The programmer uses atomics in this particular example; He obeys his part of the contract. The system guarantees *well-defined* program behavior without data races. In addition to that, the system can execute the four operations in each combination. If the programmer uses the relaxed semantic, the pillars of the contract dramatically change. On one hand, it is a lot more difficult for the programmer to understand possible interleavings of the two threads. On the other hand, the system has a lot more optimization possibilities.

With Relaxed Semantic

With the relaxed semantic - also called Weak Memory Model - many more combinations of the four operations are possible. The *counter-intuitive* behavior is that thread 1 can see the operations of thread 2 in a different order, so there is no view of a global clock. From the perspective of thread 1, it is possible that the operation `res2= x.load()` overtakes `y.store(1)`. It is even

possible that thread 1 or thread 2 do not perform their operations in the order defined in the source code. For example, thread 2 can first execute `res2= x.load()` and then `y.store(1)`.

With Acquire-Release Semantic

There are a few models between the sequential consistency and the relaxed-semantic. The most important one is the acquire-release semantic. With the acquire-release semantic, the programmer has to obey weaker rules than with sequential consistency. In contrast, the system has more optimization possibilities. The acquire-release semantic is the key to a deeper understanding of synchronization and partial ordering in multithreading programming because the threads will be synchronized at specific synchronization points in the code. Without these synchronization points, it's not possible to have *well-defined* behavior of threads, tasks, or condition variables possible.

In the last section, I introduced sequential consistency as the default behavior of atomic operations. But what does that mean? You can specify the memory order for each atomic operation. If no memory order is specified, sequential consistency is applied - meaning that the flag `std::memory_order_seq_cst` is implicitly applied to each operation on an atomic. So, the following piece of code is equivalent to the latter piece of code:

```
x.store(1);
res = x.load();
```

is equivalent to the following piece of code:

```
x.store(1, std::memory_order_seq_cst);
res = x.load(std::memory_order_seq_cst);
```

For simplicity, I will use the last form in this course. Now it's time to take a deeper look into the atomics of the C++ memory model. In the next lesson, we will start with the elementary `std::atomic_flag`.