CppMem: Atomics with a Relaxed Semantic

This lesson gives an overview of atomics with a relaxed semantic used in the context of CppMem.

WE'LL COVER THE FOLLOWING

- CppMem

With the relaxed semantic, we don't have synchronization or ordering constraints on atomic operations; only the atomicity of the operations is guaranteed.

```cpp
// ongoingOptimisationRelaxedSemantic.cpp

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

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

void writing(){
  x.store(2000, std::memory_order_relaxed);
  y.store(11, std::memory_order_relaxed);
}

void reading(){
  std::cout << y.load(std::memory_order_relaxed) << " ";
  std::cout << x.load(std::memory_order_relaxed) << std::endl;
}

int main(){
  std::thread thread1(writing);
  std::thread thread2(reading);
  thread1.join();
  thread2.join();
};
```

For the relaxed semantic, my key questions are very easy to answer. These are my questions:

1. Does the program have well-defined behavior?
2. Which values for x and y are possible?

On one hand, all operations on `x` and `y` are atomic, so the program is well-defined. On the other hand, there are no restrictions on the possible interleavings of threads. The result may be that `thread2` sees the operations on `thread1` in a different order. This is the first time in our process of ongoing optimizations that `thread2` can display `x == 0` and `y == 11` and all combinations of `x` and `y` are therefore possible.

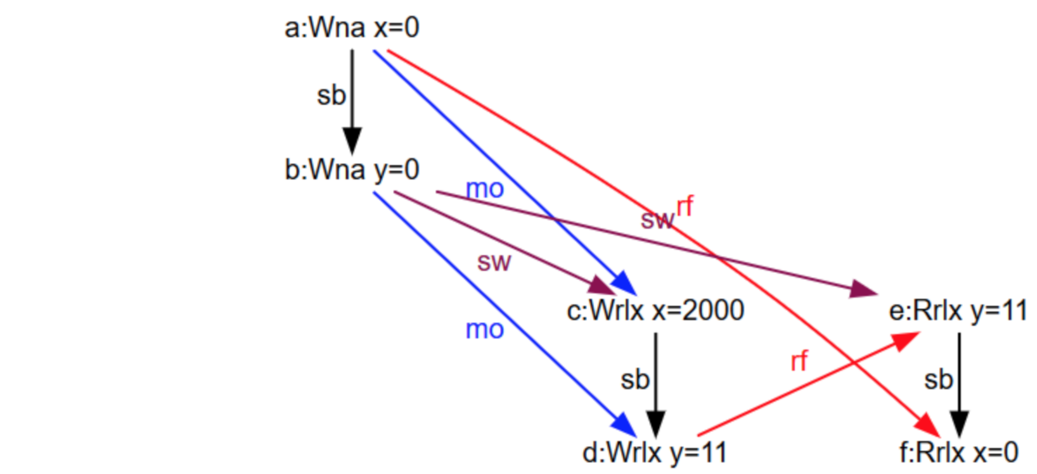
y	x	Values possible?
0	0	Yes
11	0	Yes
0	2000	Yes
11	2000	Yes

Now I'm curious how the graph of CppMem will look like for `x == 0` and `y == 11`?

CppMem

```
int main(){
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ {
      x.store(2000, memory_order_relaxed);
      y.store(11, memory_order_relaxed);
      }
  ||| {
      y.load(memory_order_relaxed);
      x.load(memory_order_relaxed);
      }
  }}}
  }}}
}
```

That was the CppMem program. Now, let’s go to the graph that produces counter-intuitive behavior.



`x` reads the value 0 (line 10), but `y` reads the value 11 (line 9); this happens, although the writing of `x` (line 5) is sequenced before the writing of `y` (line 6).