CppMem: Atomics with Non-Atomics

This lesson highlights atomics with non-atomics used in the context of CppMem.

WE'LL COVER THE FOLLOWING

- CppMem

A typical misunderstanding in the application of the acquire-release semantic is to assume that the acquire operation is waiting for the release operation. Based on this wrong assumption, you may think that `x` does not have to be an atomic variable and we can further optimize the program.

```
// ongoingOptimisationAcquireReleaseBroken.cpp

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

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

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

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

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

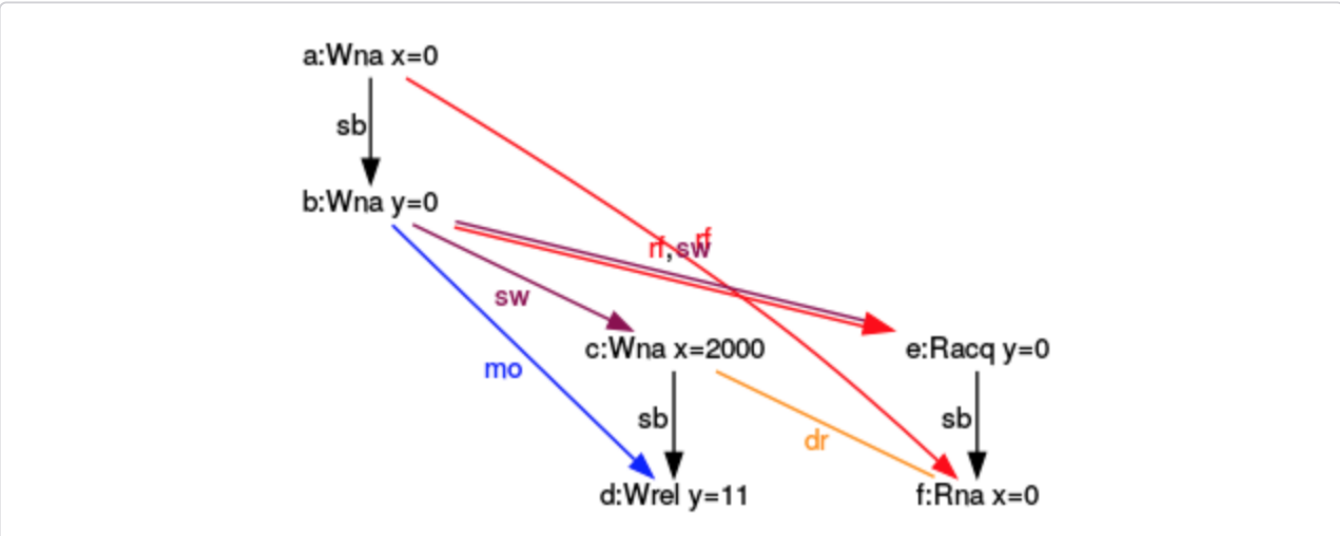
The program has a data race on `x` and, therefore, undefined behavior. The acquire-release semantic guarantees that if `y.store(11, std::memory_order_release)` (line 12) is executed before `y.load(std::memory_order_acquire)` (line 16), then `x = 2000` (line 11) will be executed before the reading of `x` in line 17. If not, the reading of `x` will be executed at the same time as the writing of `x`. So, we have concurrent access to a shared variable and one of them is a write operation. That is by definition a ***data race***.

To make my point more clear, let me use CppMem.

CppMem

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

The data race occurs when one thread is writing `x = 2000` and the other thread is reading `x`. Therefore, we get a **dr** symbol (data race) on the corresponding yellow arrow.



The final step in the process of ongoing optimization is still missing: *relaxed semantic*.