# CppMem: Atomics with an Acquire-Release Semantic

This lesson gives an overview of atomics with acquire-release semantic used in the context of CppMem.



The synchronization in the acquire-release semantic takes place between atomic operations on the same atomic. This is in contrast to the sequential consistency where we have synchronization between threads. Due to this fact, the acquire-release semantic is more lightweight and, therefore, faster.

Here is the program with acquire-release semantic:

```
// ongoingOptimisationAcquireRelease.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_release);
}

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

int main(){
```

```
    std::thread thread1(writing);
    std::thread thread2(reading);
    thread1.join();

    thread2.join();
};
```

On first glance you will notice that all operations are atomic, so the program is well-defined. But the second glance shows more; the atomic operations on `y` are attached with the flag `std::memory_order_release` (line 12) and `std::memory_order_acquire` (line 16). In contrast to that, the atomic operations on `x` are annotated with `std::memory_order_relaxed` (lines 11 and 17), so there are no synchronizations and ordering constraints for `x`. The answer to the possible values for `x` and `y` can only be given by `y`.

It holds:

- `y.store(11,std::memory_order_release)` **synchronizes-with** `y.load(std::memory_order_acquire)`
- `x.store(2000,std::memory_order_relaxed)` **is visible before** `y.store(11, std::memory_order_release)`
- `y.load(std::memory_order_acquire)` **is visible before** `x.load(std::memory_order_relaxed)`

## Explanation #

I will elaborate a little bit more on these three statements. The key idea is that the store of `y` in line 12 synchronizes with a load of `y` in line 16. This is due to the fact that the operations take place on the same atomic and they use the acquire-release semantic. `y` uses `std::memory_order_release` in line 12 and `std::memory_order_acquire` in line 16. The pairwise operation on `y` has another very interesting property. They establish a kind of barrier relative to `y`; so, `x.store(2000, std::memory_order_relaxed)` cannot be executed after `y.store(std::memory_order_release)` and x.load() cannot be executed before `y.load()`.

The reasoning in the case of the acquire-release semantic is more sophisticated than in the case of the previous sequential consistency, but the sophisticated than in the case of the previous sequential consistency, but the

possible values for `x` and `y` are the same; Only the combination `y == 11` and `x == 0` is not possible.

There are three different interleavings of the threads possible, which produce the three different combinations of the values `x` and `y`.

- `thread1` will be executed before `thread2`.
- `thread2` will be executed before `thread1`.
- `thread1` executes `x.store(2000)` before `thread2` will be executed.

To make a long story short, here are all possible values for `x` and `y`.

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

Once more, let's verify our thinking with CppMem.

## CppMem

Here is the corresponding program:

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

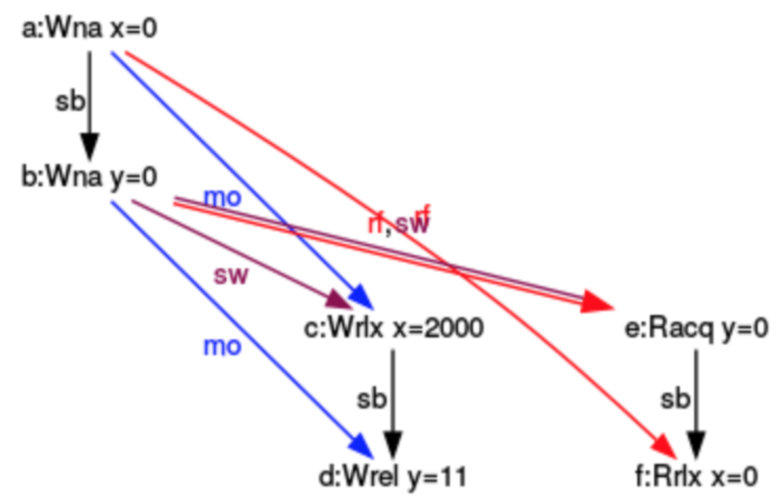
```
  ||| {
      y.load(memory_order_acquire);
      x.load(memory_order_relaxed);
      }
  }}}
}
```

We already know that all results are possible except for (y = 11, x = 0).

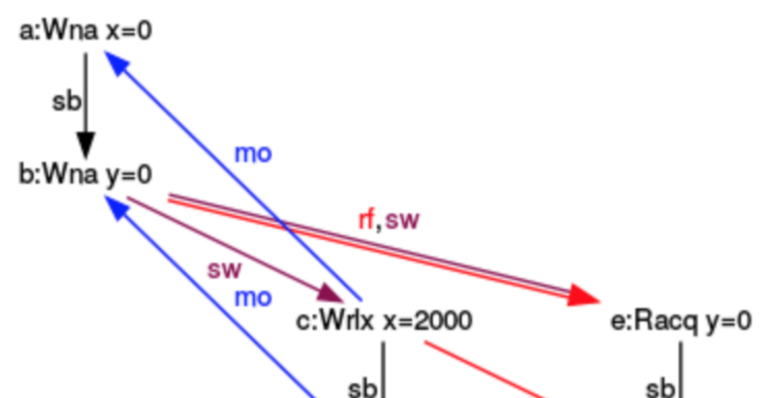
## Possible Executions

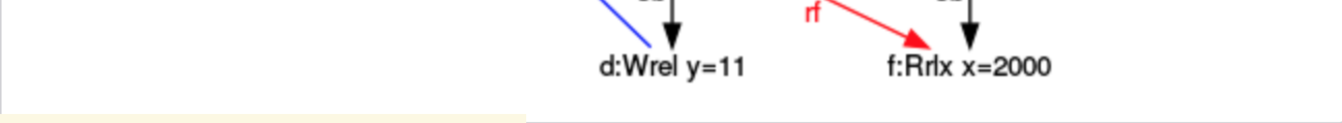
I will only refer to the three graphs with consistent execution. The graphs show that there is an acquire-release semantic between the store-release of `y` and the load-acquire operation of `y`. It will not make any difference if the reading of `y` **(rf)** takes places in the main thread or in a separate thread. The graphs show the synchronizes-with relation using a **sw** annotated arrow.

### Execution for (y = 0, x = 0)

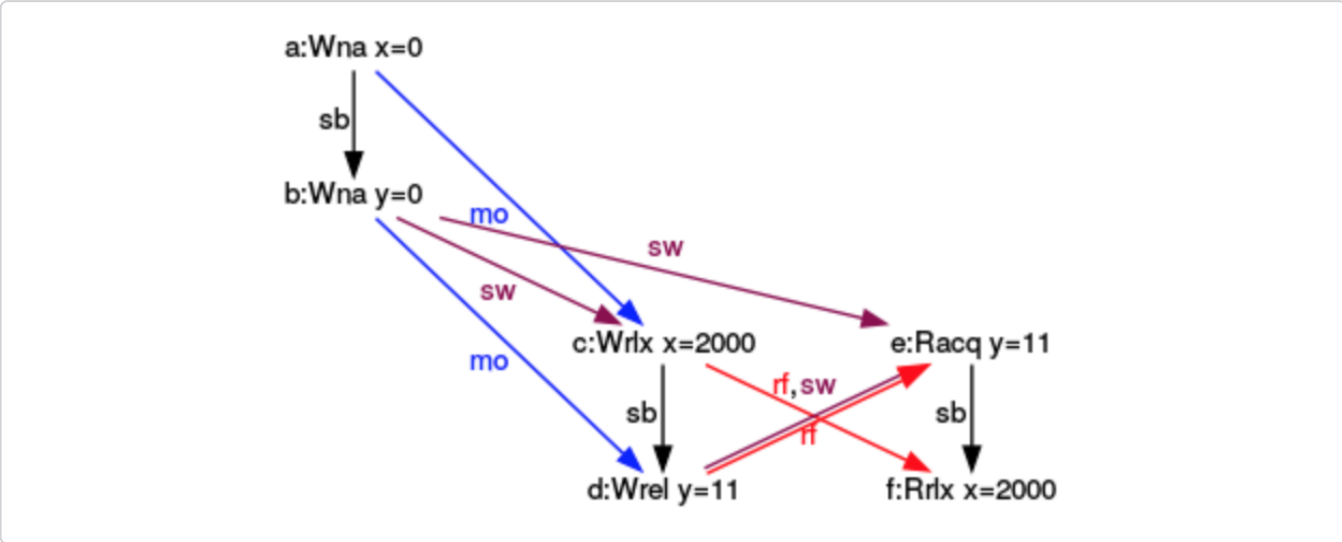



### Execution for (y = 0, x = 2000)






### Execution for (y = 11, x = 2000)




`x` does not have to be atomic. This was my first and wrong assumption; see why in the next lesson. :)