CppMem: Atomics with Sequential Consistency

This lesson gives an overview of atomics with sequential consistency used in the context of CppMem.

WE'LL COVER THE FOLLOWING

If you don't specify the memory model, sequential consistency will be applied. Sequential consistency guarantees two properties: each thread executes its instructions in source code order, and all threads follow the same global order. Here is the optimized version of the program using atomics.

```
// ongoingOptimisationSequentialConsistency.cpp

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

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

void writing(){
  x.store(2000);
  y.store(11);
}

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

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

Let’s analyze the program. The program is data race free because `x` and `y` are atomics. Therefore, only one question is left to answer: What values are possible for `x` and `y`? The question is easy to answer. Thanks to the sequential consistency, all threads have to follow the same global order.

It holds true:

- `x.store(2000);` **happens-before** `y.store(11);`
- `std::cout << y.load() << " ";` **happens-before** `std::cout << x.load() << std::endl;`

Hence the value of `x.load()` cannot be `0` if `y.load()` has the value `11`, because `x.store(2000)` happens before `y.store(11)`.

All other values for `x` and `y` are possible. Here are three possible interleavings resulting in the three different values for `x` and `y`.

1. `thread1` will be completely executed before `thread2`.
2. `thread2` will be completely executed before `thread1`.
3. `thread1` will execute its first instruction `x.store(2000)` before `thread2` will be completely executed.

Here are all the values for `x` and `y`.

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

11	2000	Yes

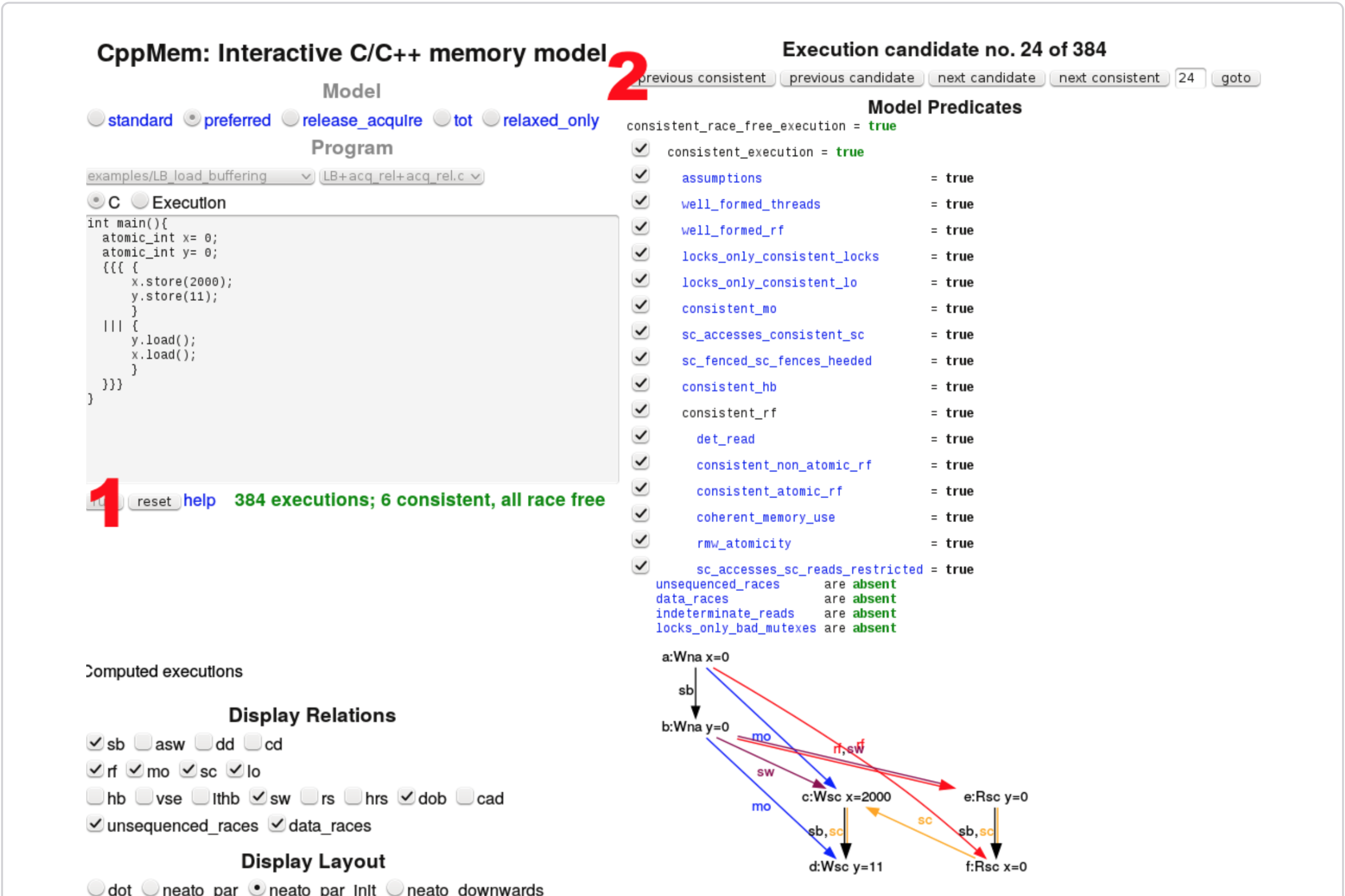
Let me verify my reasoning with CppMem.

CppMem

Here is the corresponding program in CppMem.

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

First, a little bit of syntax. CppMem uses the `typedef atomic_int` for `std::atomic<int>` in lines 2 and 3. When I execute the program, I'm overwhelmed by the number of execution candidates.



CppMem: Interactive C/C++ memory model

Model

standard [✓ preferred] release_acquire tot relaxed_only

Program

examples/LB_load_buffering LB+acq_rel+acq_rel.c

[✓ C] Execution

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

reset help **384 executions; 6 consistent, all race free**

Computed executions

Display Relations

[✓ sb] asw dd cd
[✓ rf] [✓ mo] [✓ sc] [✓ lo]
hb vse ithb [✓ sw] rs hrs [✓ dob] cad
[✓ unsequenced_races] [✓ data_races]

Display Layout

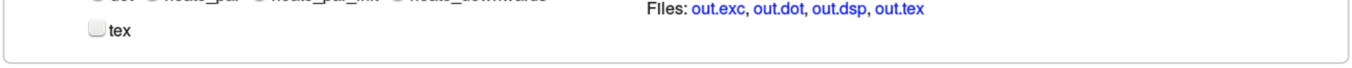
dot neato_par [✓ neato_par_init] neato_downwards

Execution candidate no. 24 of 384

previous consistent | previous candidate | next candidate | next consistent | 24 | goto

Model Predicates

```
consistent_race_free_execution = true
✓ consistent_execution = true
✓   assumptions                       = true
✓   well_formed_threads               = true
✓   well_formed_rf                    = true
✓   locks_only_consistent_locks       = true
✓   locks_only_consistent_lo          = true
✓   consistent_mo                     = true
✓   sc_accesses_consistent_sc         = true
✓   sc_fenced_sc_fences_heeded        = true
✓   consistent_hb                     = true
✓   consistent_rf                     = true
✓     det_read                        = true
✓     consistent_non_atomic_rf        = true
✓     consistent_atomic_rf            = true
✓     coherent_memory_use             = true
✓     rmw_atomicity                   = true
✓     sc_accesses_sc_reads_restricted = true
    unsequenced_races     are absent
    data_races            are absent
    indeterminate_reads   are absent
    locks_only_bad_mutexes are absent
```

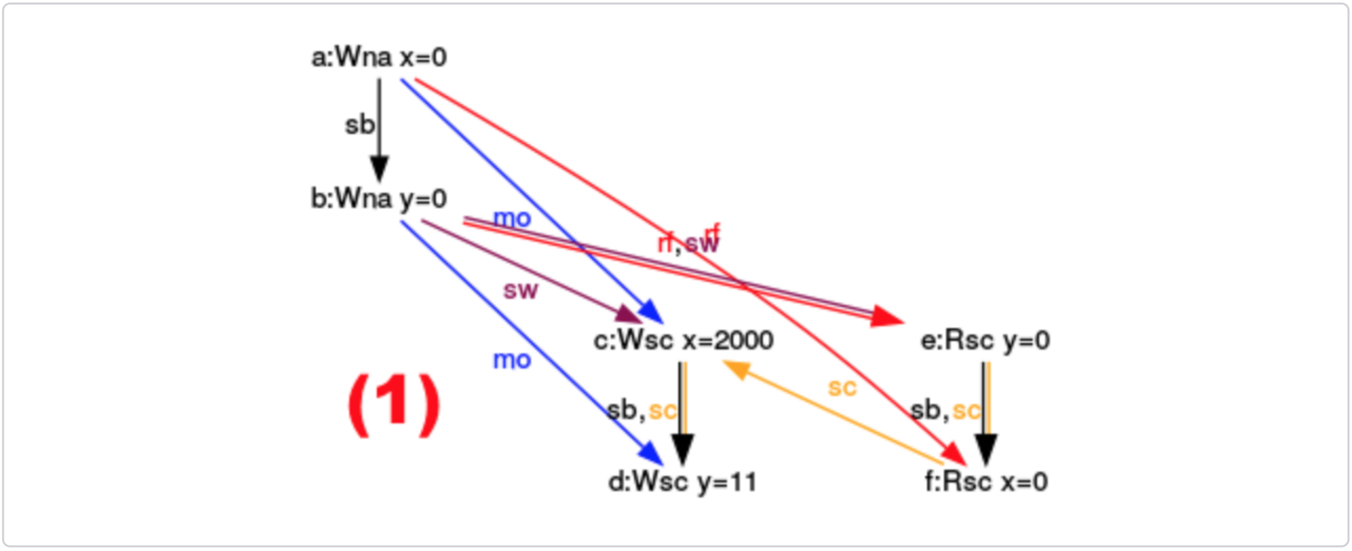

☐ tex

Files: out.exc, out.dot, out.dsp, out.tex

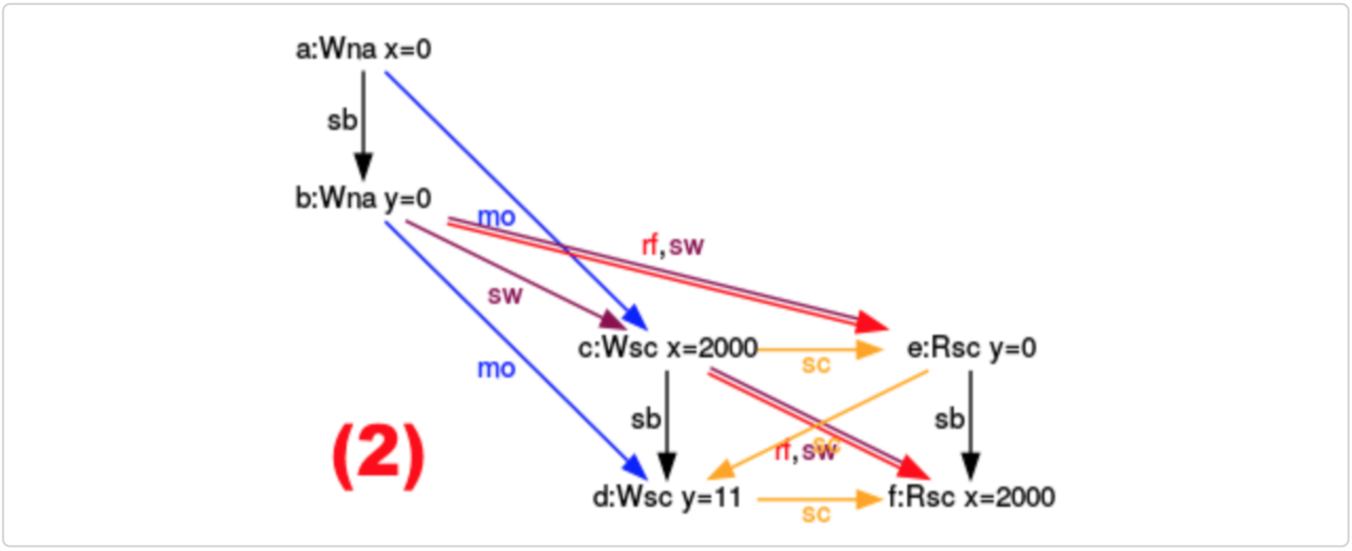
There are 384 **(1)** possible execution candidates, but only 6 of them are consistent; no candidate has a *data race*. I'm only interested in the 6 consistent executions and ignore the other 378 non-consistent executions. Non-consistent means, for example, that they will not respect the *modification order*. I use the interface **(2)** to get the six annotated graphs.

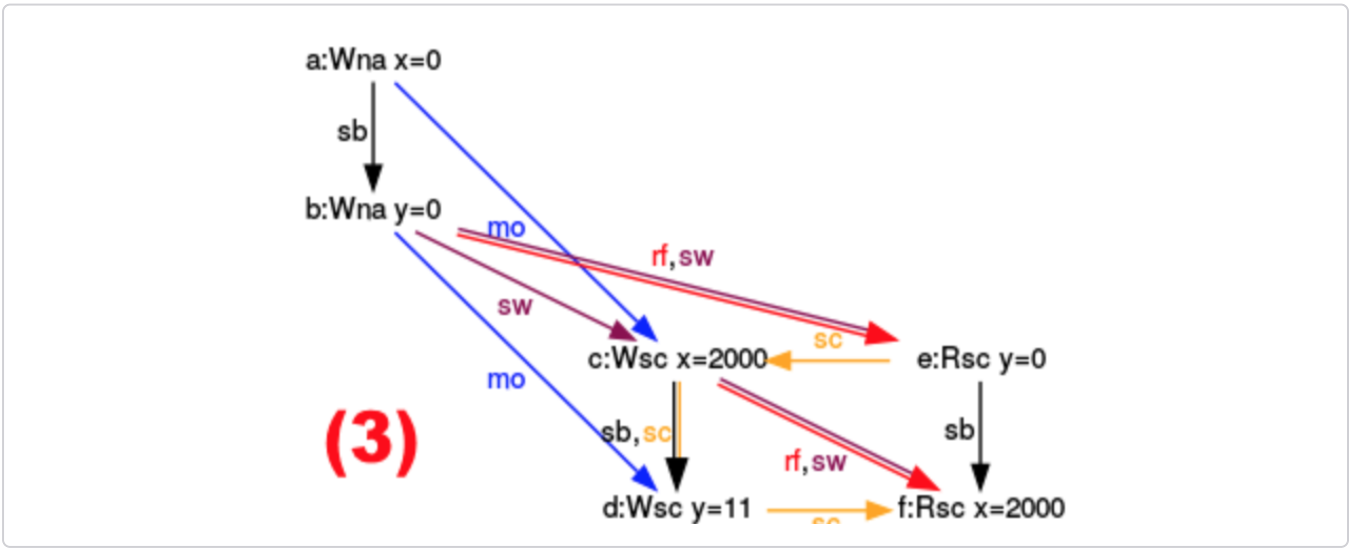
We already know that all values for `x` and `y` are possible except for `y = 11` and `x = 0`. This is because of the sequential consistency. Now I'm curious, which interleaving of threads will produce which values for `x` and `y`?

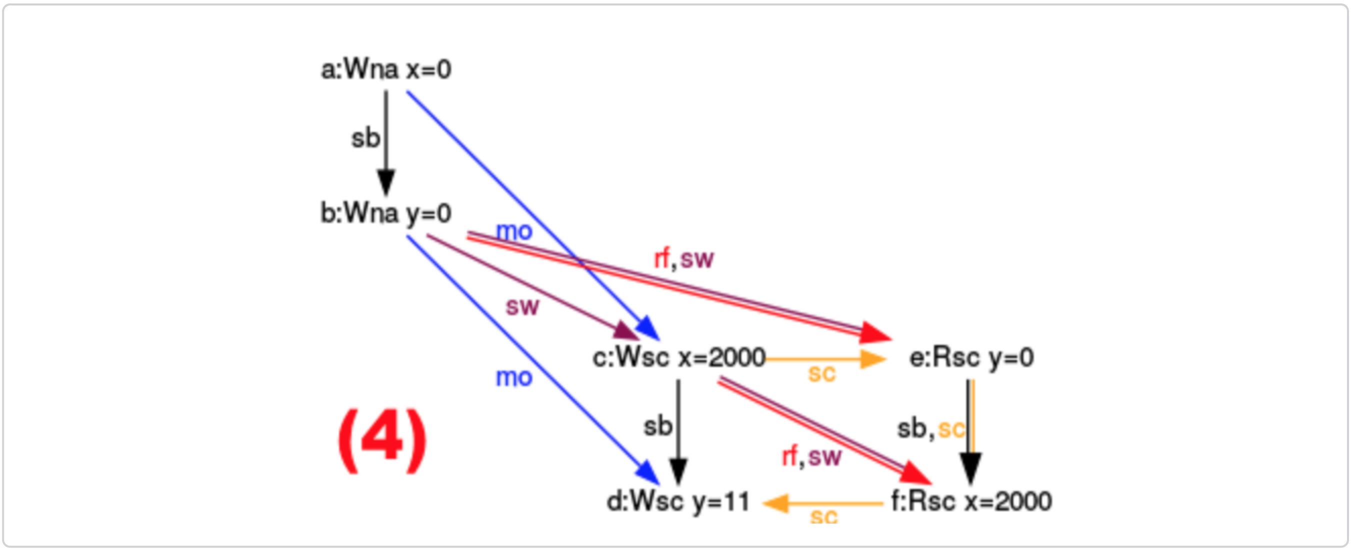
Execution for (y = 0, x = 0)

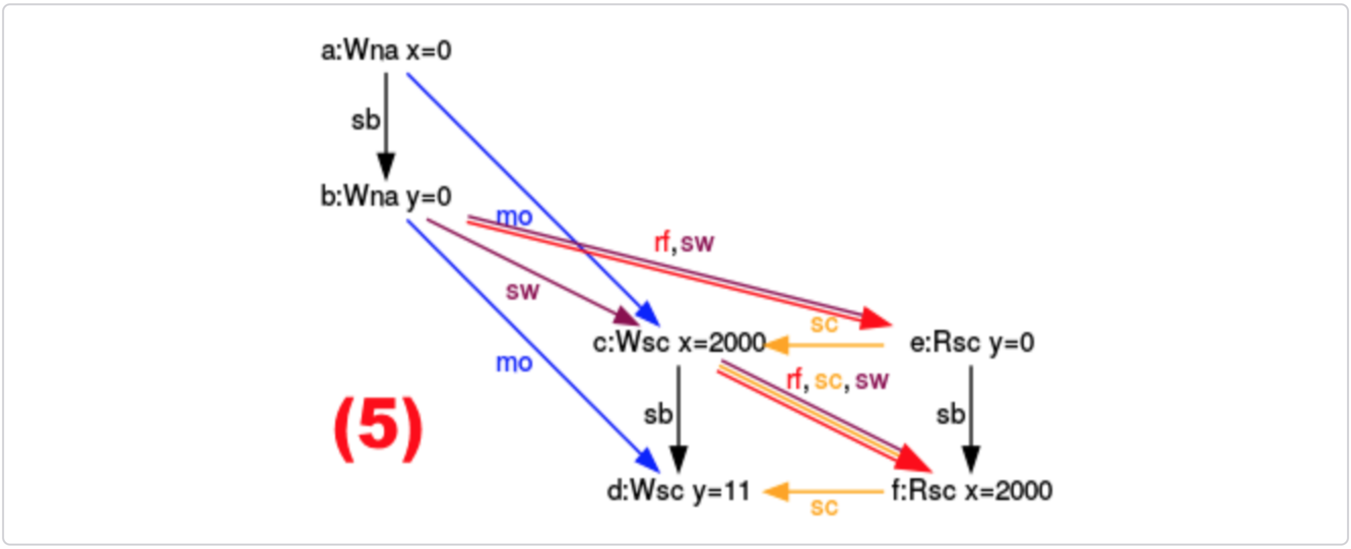


Executions for (y = 0, x = 2000)

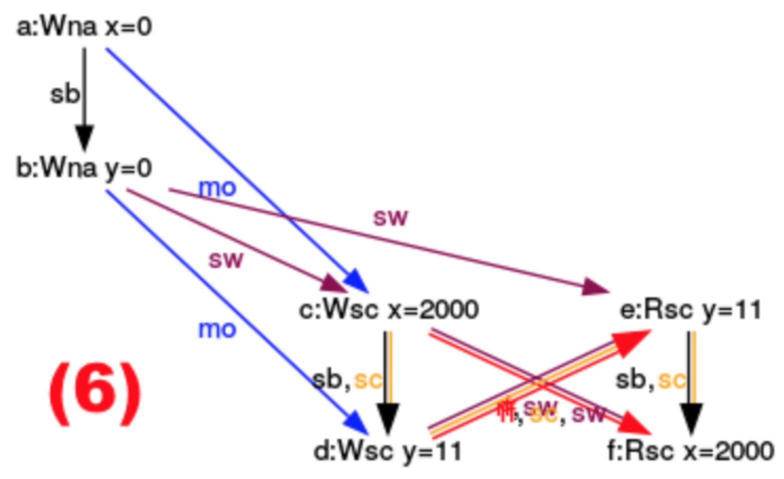








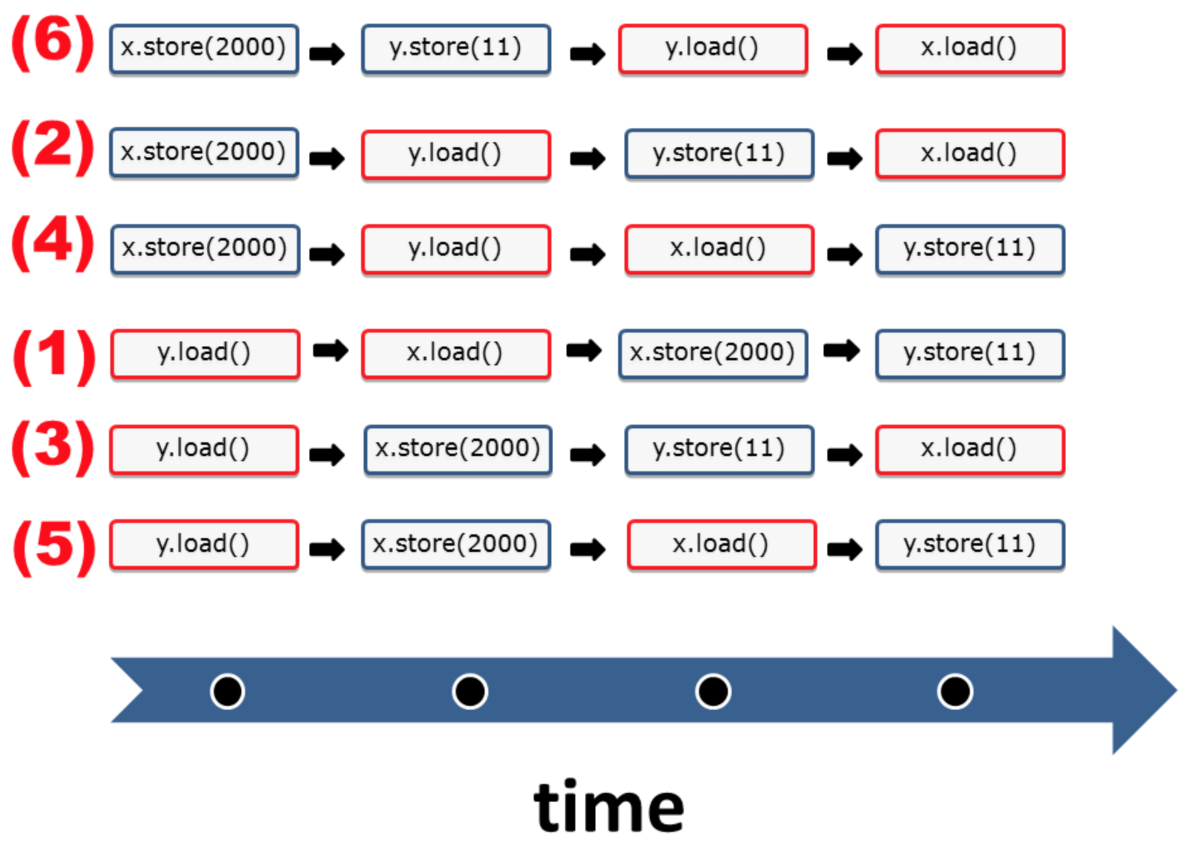
Execution for (y = 11, x = 2000)



I'm not done with my analysis. I'm interested in which sequence of instructions corresponds to which of the six graphs.

Sequence of Instructions

I have assigned each graph to its corresponding sequence of instructions.


(6) x.store(2000) ➡ y.store(11) ➡ y.load() ➡ x.load()
(2) x.store(2000) ➡ y.load() ➡ y.store(11) ➡ x.load()
(4) x.store(2000) ➡ y.load() ➡ x.load() ➡ y.store(11)
(1) y.load() ➡ x.load() ➡ x.store(2000) ➡ y.store(11)
(3) y.load() ➡ x.store(2000) ➡ y.store(11) ➡ x.load()
(5) y.load() ➡ x.store(2000) ➡ x.load() ➡ y.store(11)
time

Cases:

Let me start with the simpler cases.

- **(1)**: It's quite simple to assign the graph (1) to the sequence (1). In the sequence (1) `x` and `y` have the values 0 because `y.load()` and `x.load()` are executed before the operations `x.store(2000)` and `y.store(11)`.
- **(6)**: The reasoning for the execution (6) is similar. `y` has the value 11 and `x` the value 2000 because all load operations happen after all store operations.
- **(2), (3), (4), (5)**: Now to the more interesting cases in which `y` has the value 0 and `x` has the value `2000`. The yellow arrows (sc) in the graph are the key to my reasoning because they stand for the sequence of instructions. For example, let's look at execution (2).
  - **(2)**: Here is the sequence of the yellow arrows (sc) in the graph (2): write `x = 2000` => read `y = 0` => write `y = 11` => read `x = 2000`. This sequence corresponds to the sequence of instructions of the second interleaving of threads (2).

Let's break the sequential consistency with the acquire-release semantic in the next lesson.