CppMem: Non-Atomic Variables

This lesson gives an overview of non-atomic variables used in the context of CppMem.

WE'LL COVER THE FOLLOWING

Using the run button immediately shows that there is a data race. To be more precise, it has two data races. Neither the access to the variable `x` nor to the variable `y` are protected. As a result, the program has undefined behavior. In C++ jargon, this means that the program has the so-called catch fire semantic; therefore, all results are possible. Your PC can even catch fire.

So, we are not able to draw conclusions about the values of x and y.

Guarantees for int variables

Most of the mainstream architectures guarantee that access to an `int` variable is atomic as long as the `int` variable is aligned naturally. Naturally aligned means that the `int` variable on a 32-bit architecture must have an address divisible by 4; On a 64-bit architecture, it's divisible by 8. I mention this so explicitly because you can adjust the alignment of your data types with C++.

I have to emphasize that I'm not advising you to use an `int` as an atomic `int`. I only want to point out that the compiler guarantees more in this

> case than the C++11 standard. If you rely on the compiler guarantee,
>
> your program will not be compliant with the C++ standard and, therefore, may break in the future.

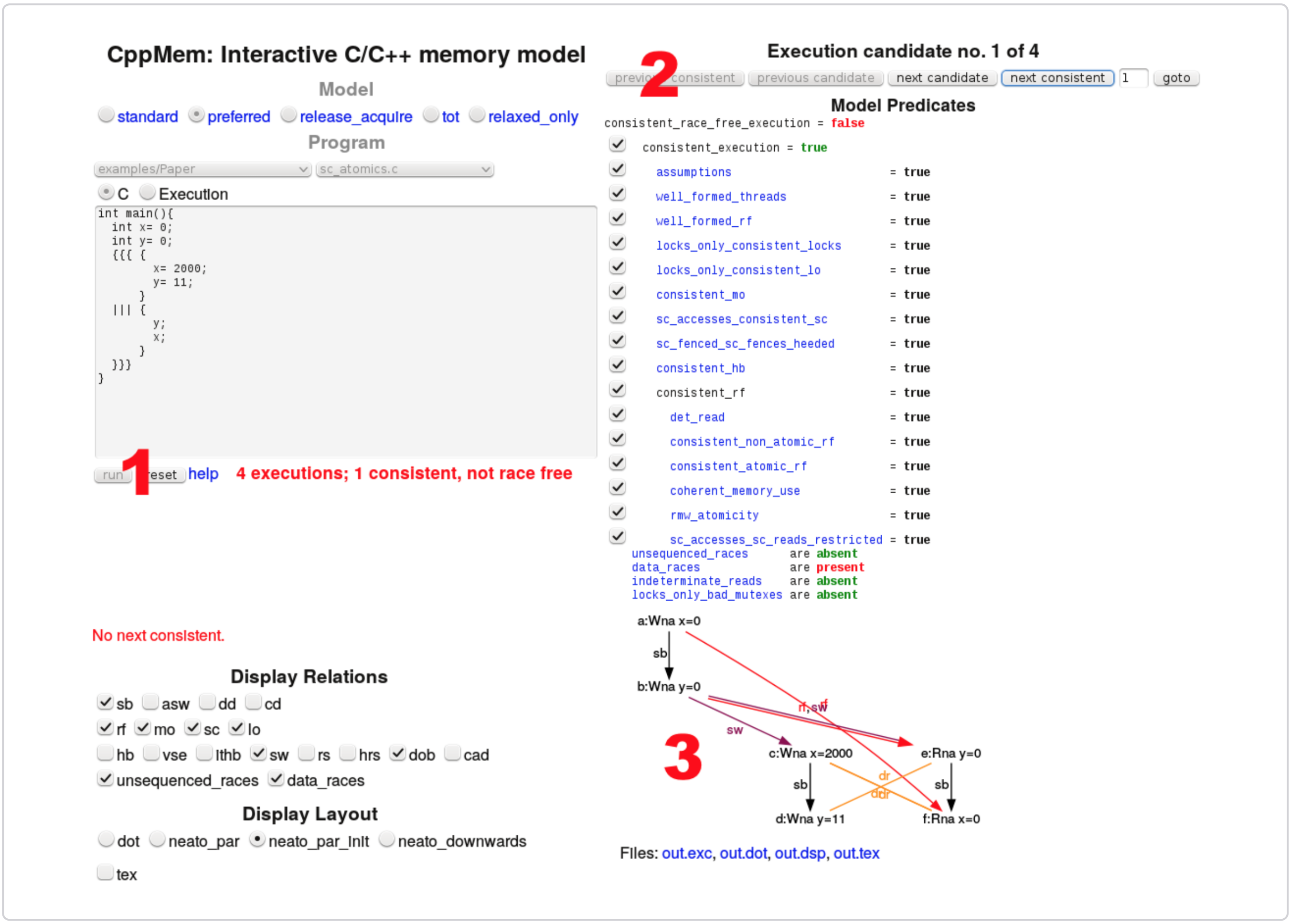
This was my reasoning. Now we should have a look at what CppMem will report about the undefined behavior of the program. As it stands, CppMem allows me to reduce the program to its bare minimum.

```
int main() {
  int x = 0;
  int y = 0;
  {{{ {
        x = 2000;
        y = 11;
      }
  ||| {
        y;
        x;
      }
  }}}
}
```

You can just define a thread in CppMem with the curly braces (lines 4 and 12) and the pipe symbol (line 8). The additional curly braces in lines 4 and 7 or lines 8 and 11 define the work package of the thread. Because I'm not interested in the output of the variables `x` and `y`, I only read them in lines 9 and 10. That was the theory for CppMem, now to the practice.

The Analysis

When I execute the program, CppMem complains (1) (in red) that one of the four possible interleavings of threads is not race free. Only the first execution is consistent. Now I can use CppMem to switch between the four executions (2) and analyze the annotated graph (3).



CppMem: Interactive C/C++ memory model

Model

standard [✓ preferred] release_acquire tot relaxed_only

Program

examples/Paper | sc_atomics.c

[✓ C] Execution

```
int main(){
  int x= 0;
  int y= 0;
  {{{ {
        x= 2000;
        y= 11;
      }
  ||| {
        y;
        x;
      }
  }}}
}
```

run 1 reset help **4 executions; 1 consistent, not race free**

No next consistent.

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

tex

Execution candidate no. 1 of 4

2 previous consistent | previous candidate | next candidate | next consistent | 1 | goto

Model Predicates

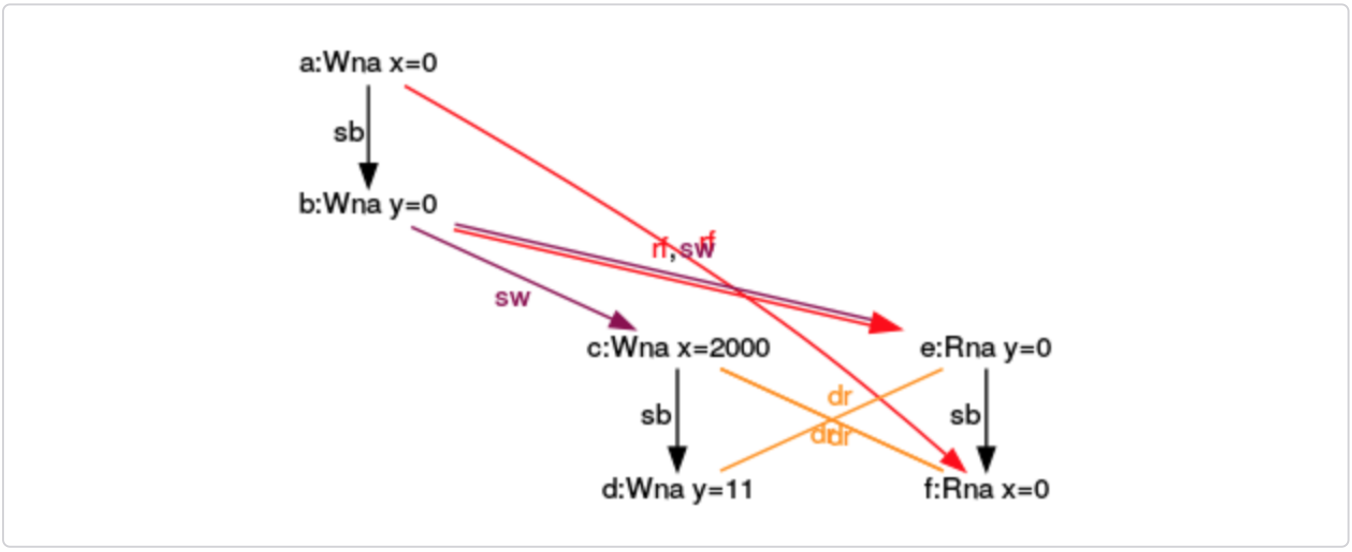
```
consistent_race_free_execution = false
✓ consistent_execution = true
✓   assumptions                       = true
✓   well_formed_threads               = true
✓   well_formed_rf                    = true
✓   locks_only_consistent_locks       = true
✓   locks_only_consistent_lo          = true
✓   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 present
  indeterminate_reads   are absent
  locks_only_bad_mutexes are absent
```

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

You get the most out of CppMem by analyzing the various graphs.

First Execution

What conclusions can we derive from the following graph?



The nodes of the graph represent the expressions of the program, and the edges represent the relationship between the expressions. In my explanation edges represent the relationship between the expressions. In my explanation,

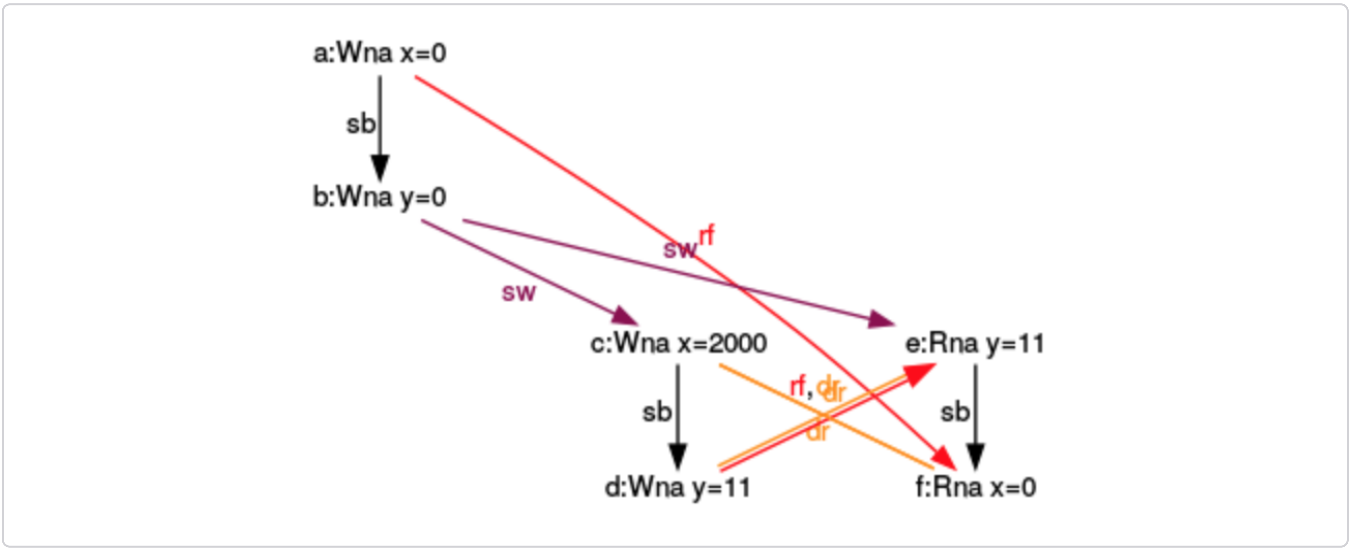
I will refer to the names (a) to (f). What can I derive from the annotations in this graph?

- **a:Wna x** = **0**: Is the first expression (a), which is a non-atomic write of `x`.
- **sb (sequenced-before)**: The writing of the first expression (a) is sequenced before the writing of the second expression (b). These relations also holds between the expressions (c) and (d), and (e) and (f).
- **rf (read from)**: The expression (e) reads the value of `y` from the expression (b). Accordingly, (f) reads from (a).
- **sw (synchronizes-with)**: The expression (a) synchronises with (f). This relation holds true because the expression (f) takes place in a separate thread. Creation of a thread is a synchronization point; everything that happens before the thread creation is visible in the thread. For symmetry reasons, the same argument holds true between (b) and (e).
- **dr (data race)**: Here is the data race between the reading and writing of the variables `x` and `y`. The program has undefined behavior.

> **Why is the execution consistent?**
>
> The execution is consistent because the values `x` and `y` are read from the values in the main thread (a) and (b). If the values were read from `x` and `y` using a separate thread (not main-thread) in the expressions (c) and (d), it can happen that the values of `x` and `y` in (e) and (f) are only partial reads; this is not consistent. Or to say it differently, in the concrete execution `x` and `y` get the value 0. You can see the values of `x` and `y` that were read in the expressions (e) and (f).

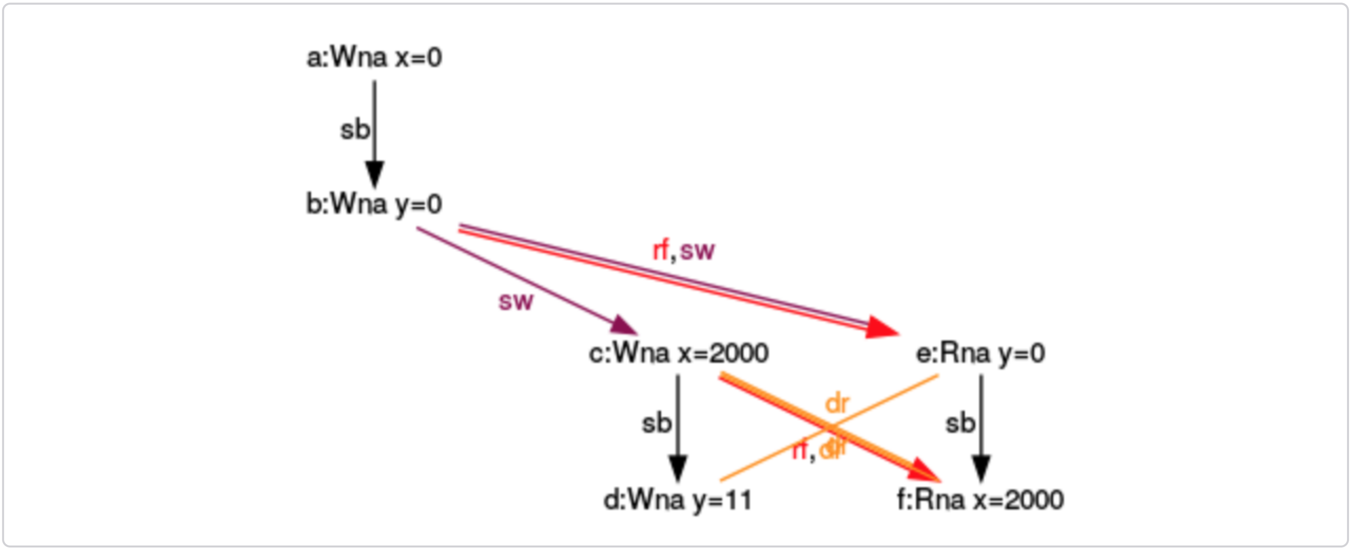
The next three executions are not consistent.

Second Execution



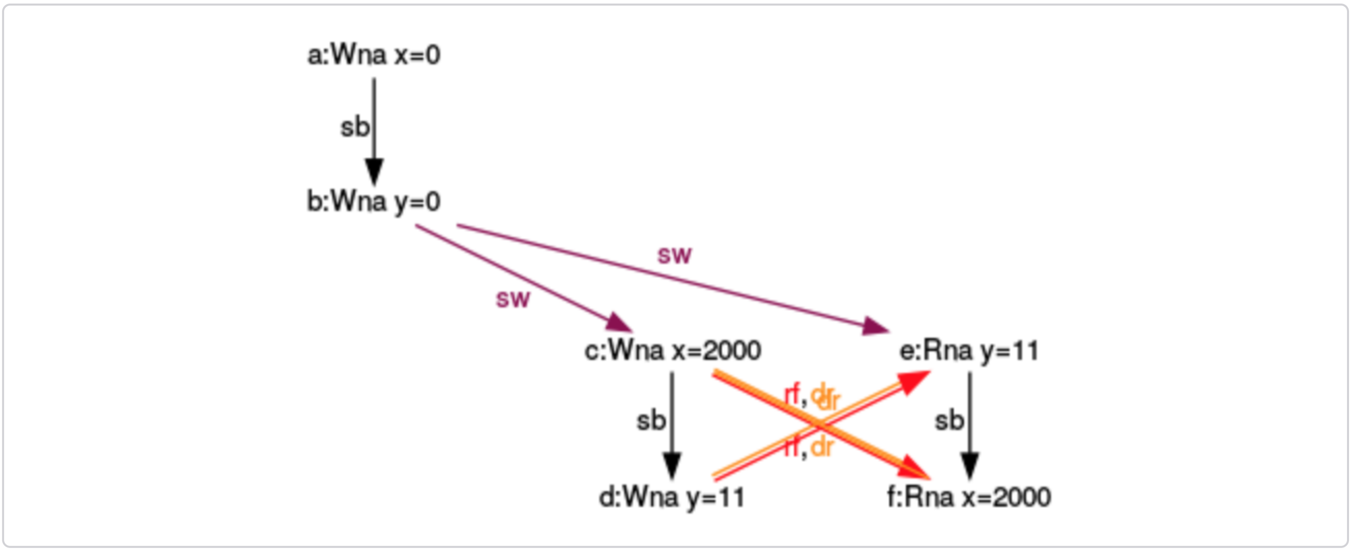
The expression (e) reads the value of y from the expression (d) in this non-consistent execution. Also, the writing of (d) happens concurrently with the reading of (e).

Third Execution



This execution is symmetric to the previous execution. That being said, the expression (f) reads concurrently from expression (c).

Fourth Execution



Now everything goes wrong. The expressions (e) and (f) read from the expressions (d) and (c) concurrently.

A Short Conclusion

Although I just used the default configuration of CppMem, I got a lot of valuable information and insight. In particular, the graphs from CppMem showed:

- All four combinations of `x` and `y` are possible: (0,0), (11,0), (0,2000), (11,2000).
- The program has at least one data race and, therefore, has undefined behavior.
- Only one of the four possible executions is consistent.

Using volatile

Using the qualifier `volatile` for `x` and `y` makes no difference from the memory model perspective, compared with using non-synchronised access to `x` and `y` .

```
int main() {
  volatile int x = 0;
  volatile int y = 0;
  {{{ {
        x = 2000;
        y = 11;
      }
  ||| {
       y;
       x;
      }
  }}}
}
```

CppMem will generate identical graphs to those seen in the previous example. The reason is quite simple: In C++, `volatile` has no multithreading semantic.

The access to `x` and `y` in this example was not synchronized and we got a data race; therefore, undefined behavior. The most obvious way for synchronization is to use locks.