The C1x and C++11 concurrency model

Mark Batty

University of Cambridge

January 16, 2013

C11 and C++11 Memory Model

A DRF model with the option to expose relaxed behaviour in exchange for high performance.

C11 takes it's model directly from C++11.

Allows for relaxed behaviour on target architectures, and compiler optimisation.

C++11: the next C++

C++11: the next C++

1300 page prose specification defined by the ISO.

C++11: the next C++

1300 page prose specification defined by the ISO.

The design is a detailed compromise:

- hardware/compiler implementability
- useful abstractions
- broad spectrum of programmers

C++11: the next C++

1300 page prose specification defined by the ISO.

The design is a detailed compromise:

- hardware/compiler implementability
- useful abstractions
- broad spectrum of programmers

We fixed serious problems in both C++11 and C1x, both now finalised.

What does C++11 look like?

```
std::atomic<int> flag0(0),flag1(0),turn(0);

void lock(unsigned index) {
  if (0 == index) {
    flag0.store(1, std::memory_order_relaxed);
    turn.exchange(1, std::memory_order_acq_rel);
    while (flag1.load(std::memory_order_acquire)
      && 1 == turn.load(std::memory_order_relaxed))
      std::this_thread::yield();
  } else {
    flag1.store(1, std::memory_order_relaxed);
    turn.exchange(0, std::memory_order_acq_rel);
    while (flag0.load(std::memory_order_acquire)
      && 0 == turn.load(std::memory_order_relaxed))
   std::this_thread::yield();
  }

void unlock(unsigned index) {
  if (0 == index) {
    flag0.store(0, std::memory_order_release);
  } else {
    flag1.store(0, std::memory_order_release);
  }
```

Atomic accesses take a n ordering parameter

From most relaxed to most like DRF-SC:

`memory_order_relaxed`

`memory_order_release/memory_order_acquire`

`memory_order_release/memory_order_consume`

`memory_order_seq_cst`

mo_seq_cst

The compiler must ensure that `mo_seq_cst` atomics have SC semantics.

```
x.store(1, mo_seq_cst);     | y.store(1, mo_seq_cst);
r1 = y.load(mo_seq_cst);    | r2 = x.load(mo_seq_cst);
```

The program above cannot end with `r1 = r2 = 0`.

mo_seq_cst

The compiler must ensure that mo_seq_cst atomics have SC semantics.

```
x.store(1, mo_seq_cst);     | y.store(1, mo_seq_cst);
r1 = y.load(mo_seq_cst);    | r2 = x.load(mo_seq_cst);
```

The program above cannot end with r1 = r2 = 0.

...so, MP is forbidden over mo_seq_cst. So are all other relaxed behaviours.

mo_release / mo_acquire

Supports fast implementation of the message passing idiom.

```
x = 1;                          | r1 = y.load(mo_acquire);
y.store(1, mo_release);         | r2 = x;
```

The program above cannot end with `r1 = 1` and `r2 = 0`.

mo_release / mo_acquire

Supports fast implementation of the message passing idiom.

```
x = 1;                          | r1 = y.load(mo_acquire);
y.store(1, mo_release);         | r2 = x;
```

The program above cannot end with `r1 = 1` and `r2 = 0`.

...so, MP is forbidden using `mo_release` and `mo_acquire`. SB and IRIW are allowed though.

mo_release / mo_consume

Supports faster implementation of the message passing idiom on Power.

`x = 1;`	`r1 = y.load(mo_consume);`
`y.store(&x, mo_release);`	`r2 = *r1;`

The program above cannot end with `r1 = &x` and `r2 = 0`.

The two loads must have an address dependency.

mo_relaxed

Very fast access, but also lots of strange behaviour.

```
r1 = x.load(mo_relaxed);  | r2 = y.load(mo_relaxed);
y.store(1, mo_relaxed);   | x.store(1, mo_relaxed);
```

The program above can end with `r1 = 1` and `r2 = 1`.

mo_relaxed

Very fast access, but also lots of strange behaviour.

```
r1 = x.load(mo_relaxed);  | r2 = y.load(mo_relaxed);
y.store(1, mo_relaxed);   | x.store(1, mo_relaxed);
```

The program above can end with `r1 = 1` and `r2 = 1`.

...so, LB is allowed using `mo_relaxed`. We will see that these accesses are more relaxed than Power even.

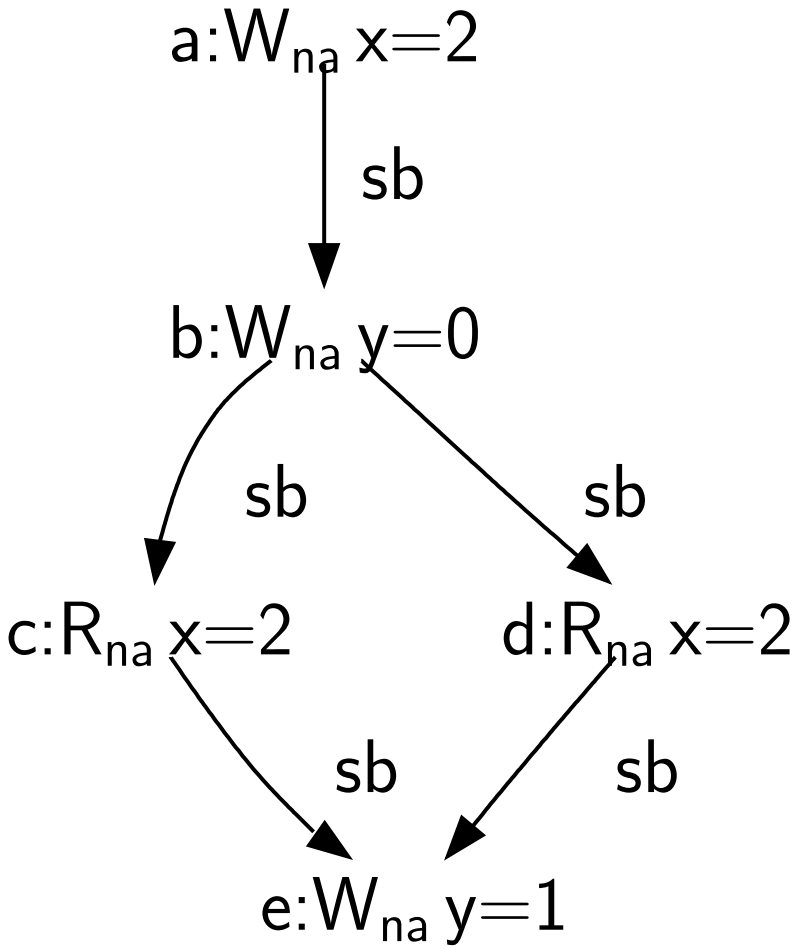
The C1x/C++11 memory model

The C1x/C++11 memory model

- sequential execution
- simple concurrency
- expert concurrency
- very expert concurrency

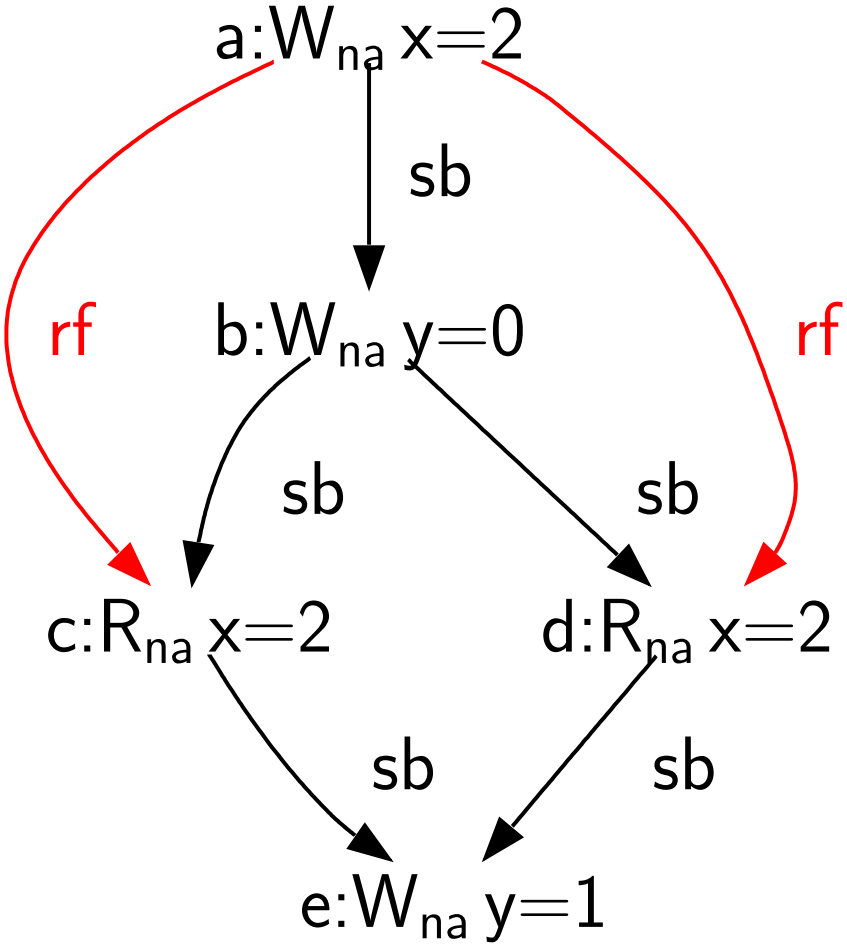
A single threaded program

```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0; }
```



A single threaded program

```
int main() {
   int x = 2;
   int y = 0;
   y = (x==x);
   return 0; }
```



The relations of a pre-execution

Each symbolic execution, E_i, contains:

- **sb** – *sequenced before*
- **asw** – *additional synchronizes with*
- **dd** – *data-dependence*

The relations of a pre-execution

Each symbolic execution, E_i, contains:

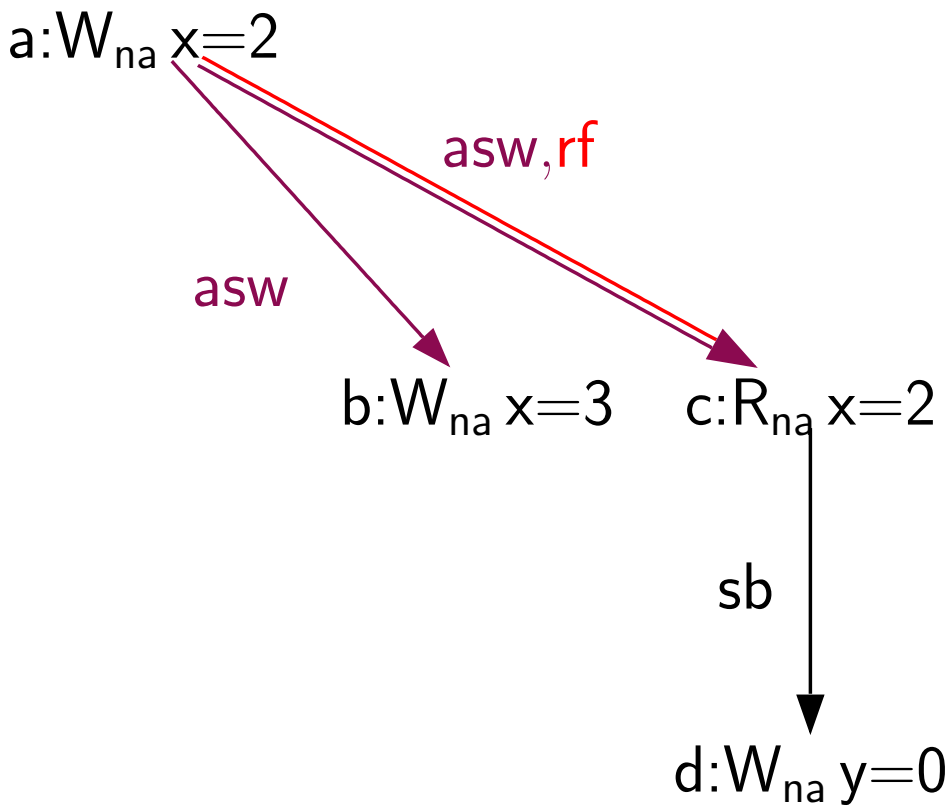
- **sb** – *sequenced before*
- **asw** – *additional synchronizes with*
- **dd** – *data-dependence*

Each full execution, X_{ij}, also has:

- **rf** – *reads from*
- **sc** – *SC order*
- **mo** – *modification order*

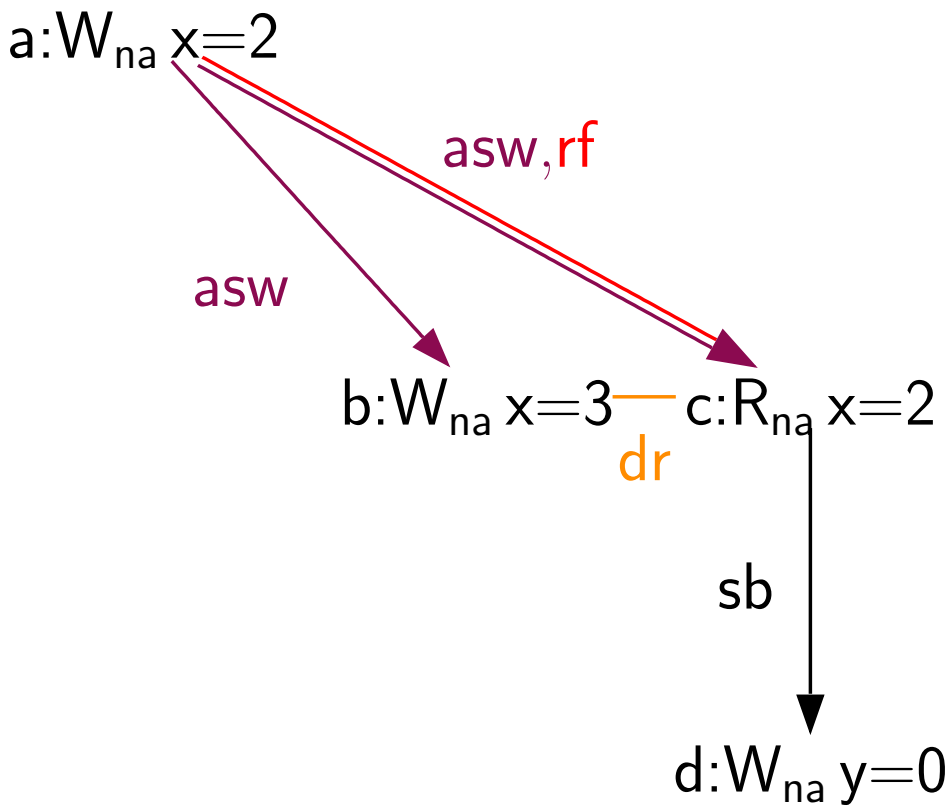
A data race

```
int y, x = 2;
x = 3;              | y = (x==3);
```



A data race

```
int y, x = 2;
x = 3;          | y = (x==3);
```

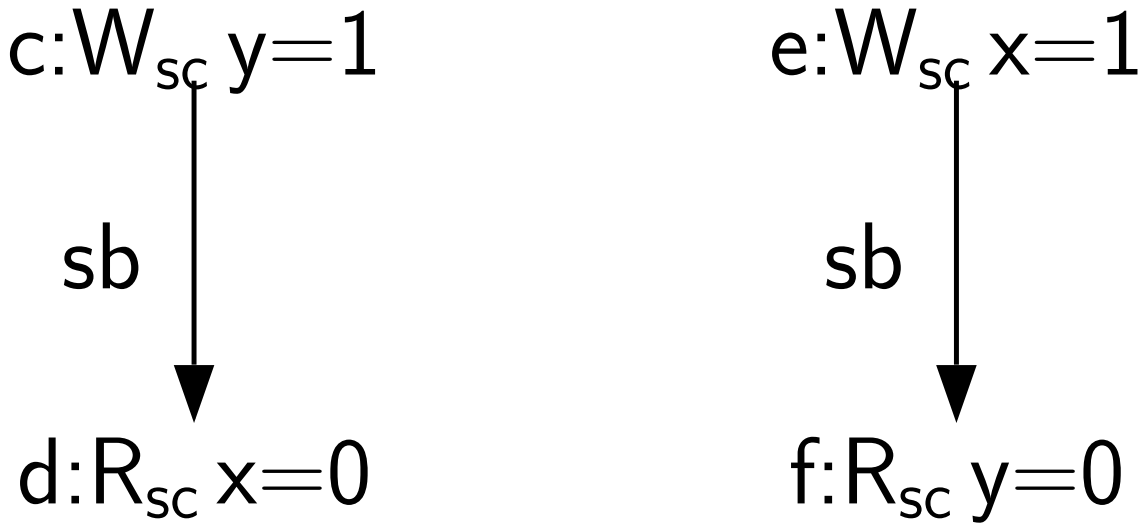


Simple concurrency: Decker's example and SC

```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);  | y.store(1, seq_cst);
y.load(seq_cst);      | x.load(seq_cst);
```

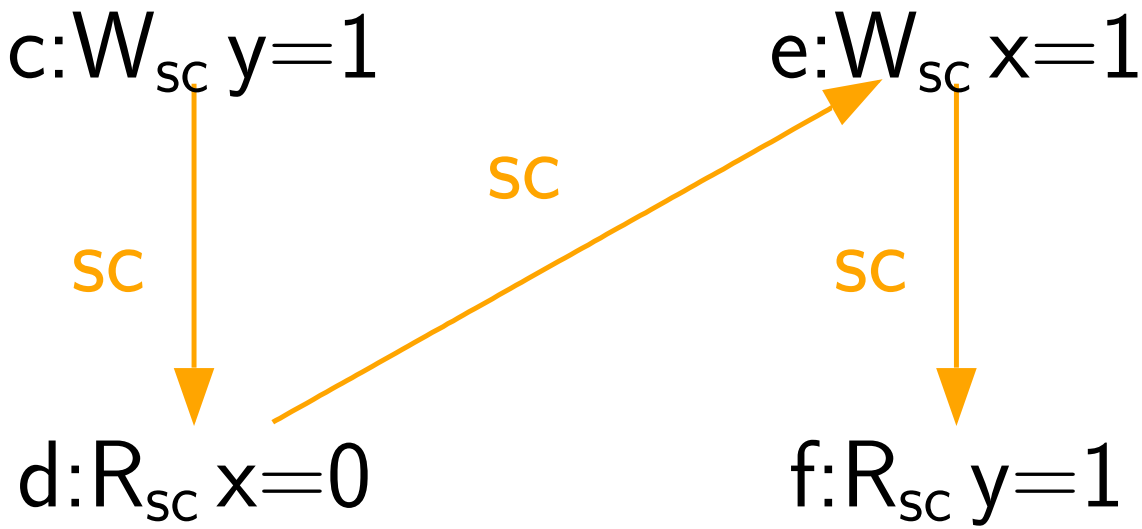
Simple concurrency: Decker's example and SC

```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);  |  y.store(1, seq_cst);
y.load(seq_cst);      |  x.load(seq_cst);
```



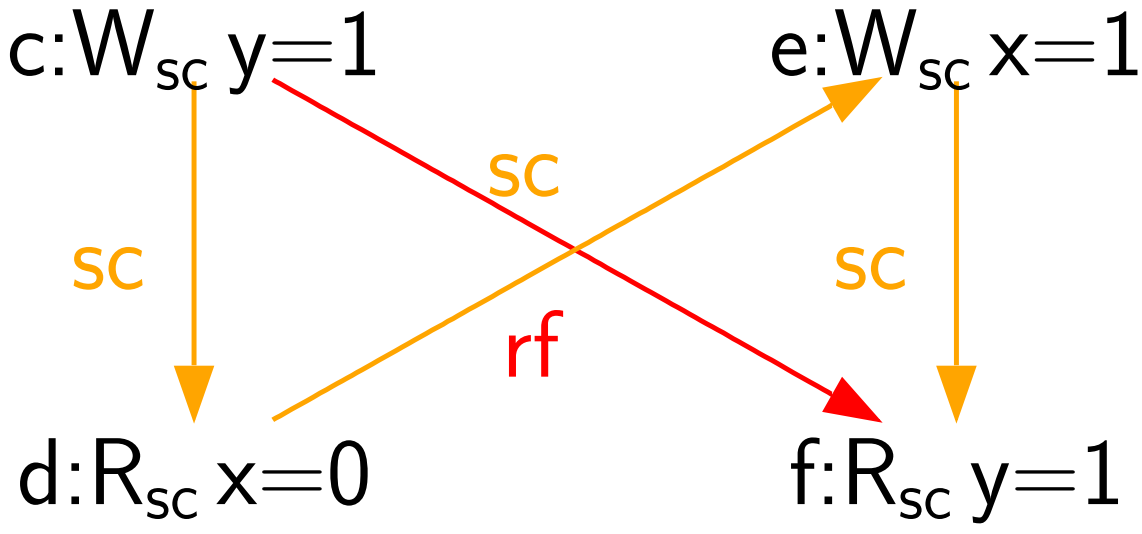
Simple concurrency: Decker's example and SC

```
atomic_int x = 0;
atomic_int y = 0;

x.store(1, seq_cst);  |  y.store(1, seq_cst);
y.load(seq_cst);      |  x.load(seq_cst);
```



SC atomics

Read the last write in SC order.

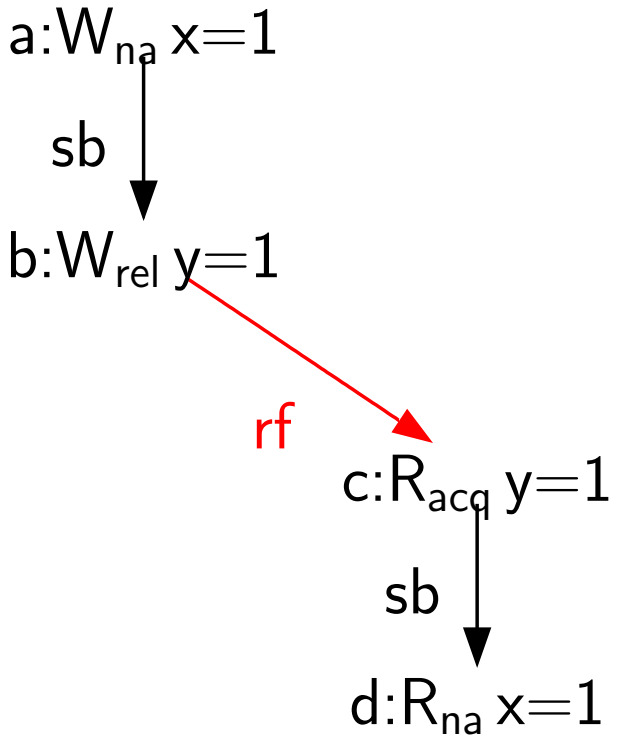


Using only `seq_cst` reads and writes gives SC.

(Initialization is not `seq_cst` though...)

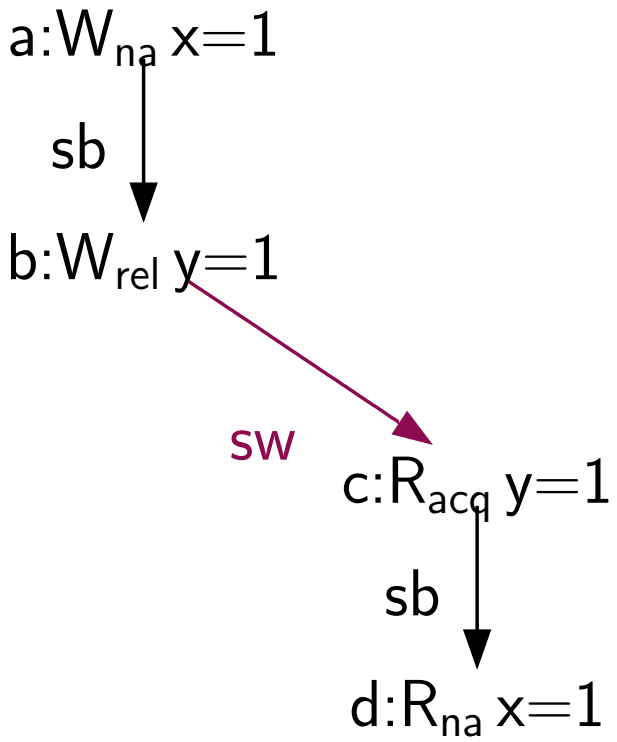
Expert concurrency: The release-acquire idiom

```
// sender                          | // receiver
x = ...                            | while (0 == y.load(acquire));
y.store(1, release);               | r = x;
```



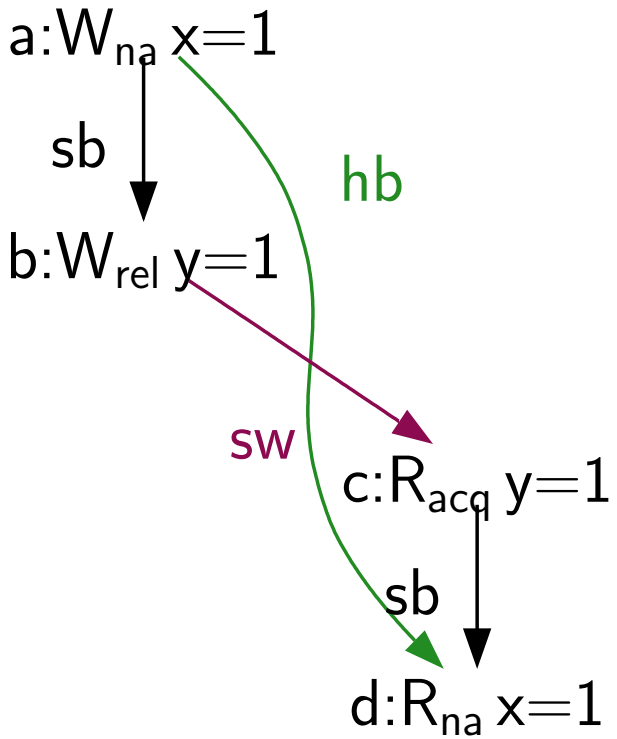
Expert concurrency: The release-acquire idiom

```
// sender                          | // receiver
x = ...                            | while (0 == y.load(acquire));
y.store(1, release);               | r = x;
```



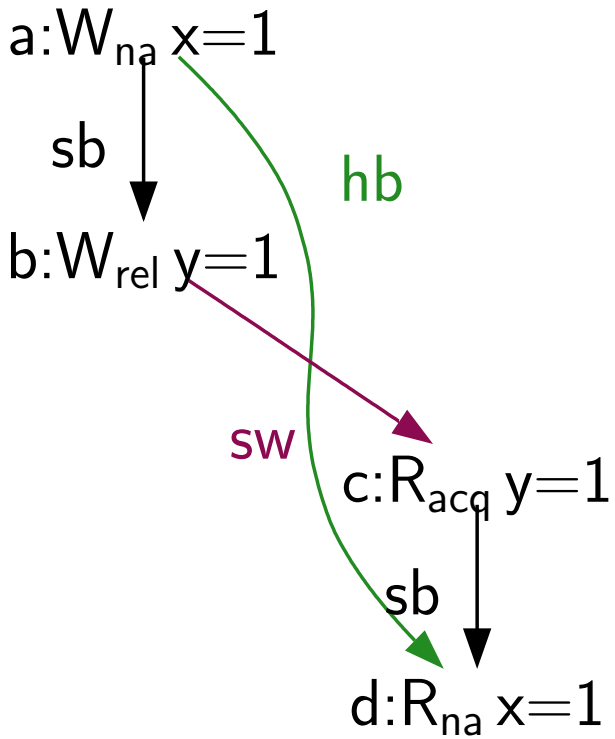
Expert concurrency: The release-acquire idiom

```
// sender                          | // receiver
x = ...                            | while (0 == y.load(acquire));
y.store(1, release);               | r = x;
```



Expert concurrency: The release-acquire idiom

```
// sender                          // receiver
x = ...                            while (0 == y.load(acquire));
y.store(1, release);               r = x;
```



$$\xrightarrow{\textit{simple-happens-before}} = \left(\xrightarrow{\textit{sequenced-before}} \cup \xrightarrow{\textit{synchronizes-with}}\right)^{+}$$

Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

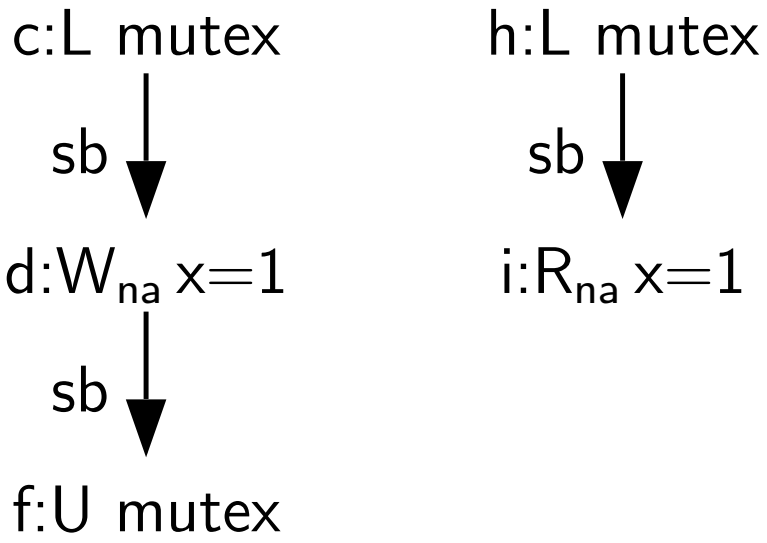
`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	

Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	

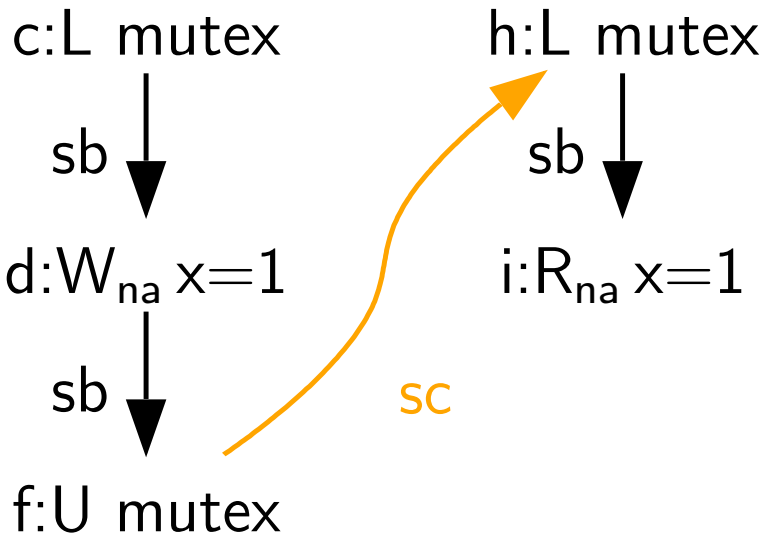


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	

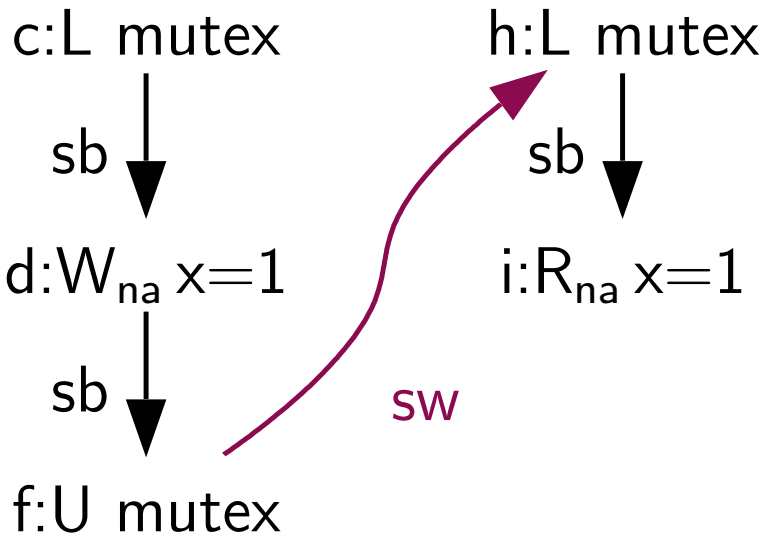


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	

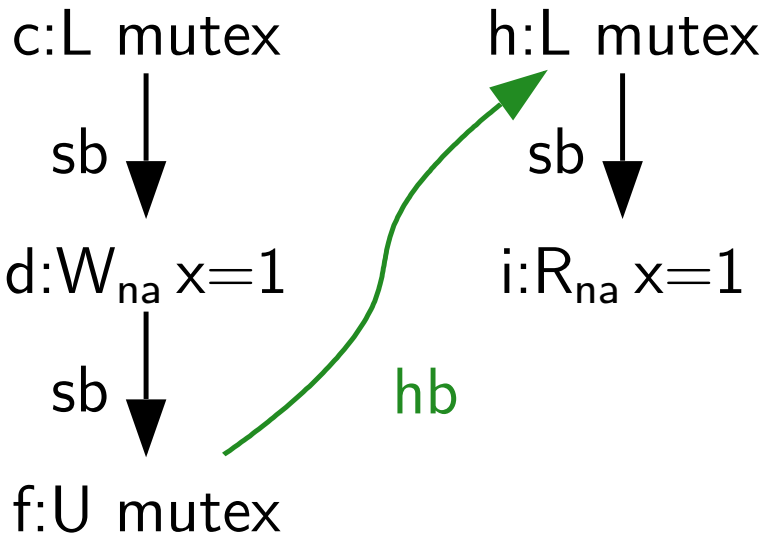


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	

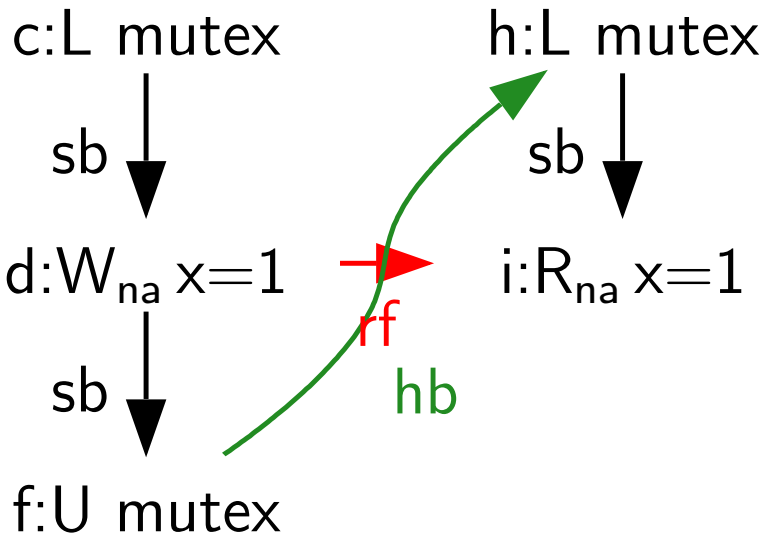


Locks and unlocks

Unlocks and locks synchronise too:

```
int x, r;
mutex m;
```

`m.lock();`	`m.lock();`
`x = ...`	`r = x;`
`m.unlock();`	



Happens-before is key to the model

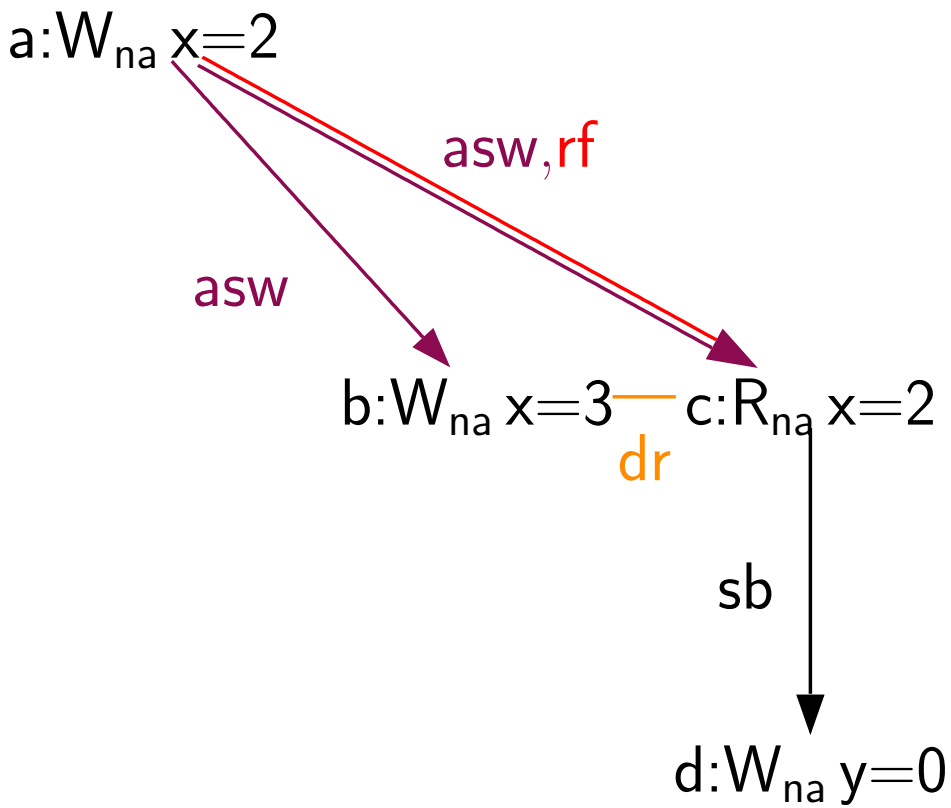
Non-atomic loads read the most recent write in happens-before. (This is unique in DRF programs)

The story is more complex for atomics, as we shall see, but we cannot read from the future, in happens-before.

Data races are defined as an absence of happens-before.

A data race

```
int y, x = 2;
x = 3;          | y = (x==3);
```



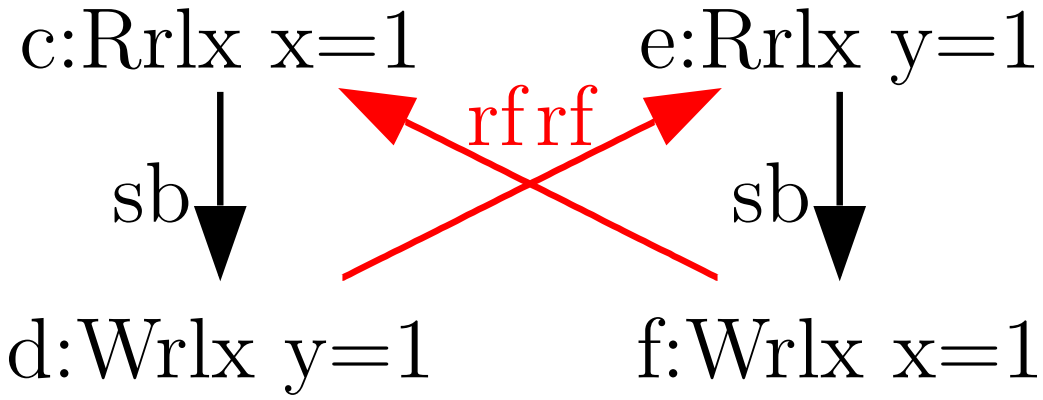
Data race definition

$$
\begin{aligned}
&\text{let } \mathit{data\_races}\ \mathit{actions}\ \mathit{hb} = \\
&\quad \{\ (a,\ b) \mid \forall\ a{\in}\mathit{actions}\ b{\in}\mathit{actions} \mid \\
&\qquad \neg\ (a = b)\ \wedge \\
&\qquad \mathrm{same\_location}\ a\ b\ \wedge \\
&\qquad (\mathrm{is\_write}\ a\ \vee\ \mathrm{is\_write}\ b)\ \wedge \\
&\qquad \neg\ (\mathrm{same\_thread}\ a\ b)\ \wedge \\
&\qquad \neg\ (\mathrm{is\_atomic\_action}\ a\ \wedge\ \mathrm{is\_atomic\_action}\ b)\ \wedge \\
&\qquad \neg\ ((a,\ b) \in \mathit{hb}\ \vee\ (b,\ a) \in \mathit{hb})\ \}
\end{aligned}
$$

A program with a data race has undefined behaviour.

Relaxed writes: load buffering

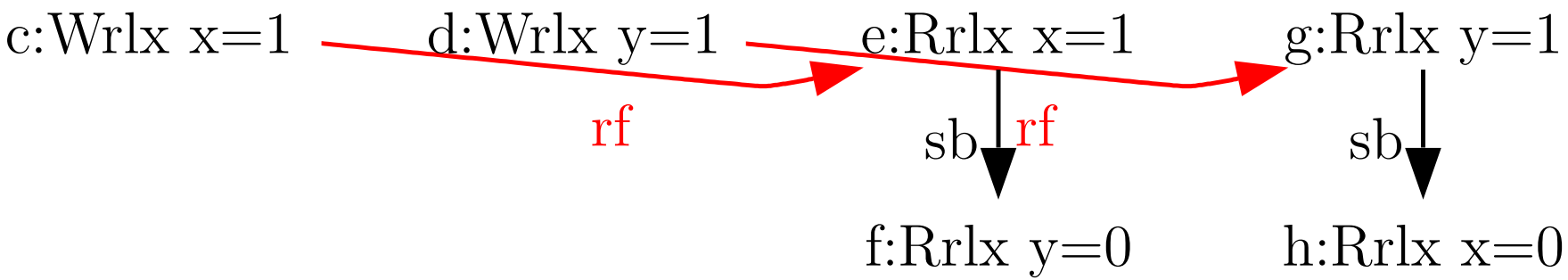
```
x.load(relaxed);          | y.load(relaxed);
y.store(1, relaxed);      | x.store(1, relaxed);
```



No synchronisation cost, but weakly ordered.

Relaxed writes: independent reads, independent writes

```
atomic_int x = 0;
atomic_int y = 0;
x.store(1, relaxed); | y.store(2, relaxed); | x.load(relaxed); | y.load(relaxed);
                                           | y.load(relaxed); | x.load(relaxed);
```



Expert concurrency: fences avoid excess synchronisation

```
// sender                          | // receiver
x = ...                            | while (0 == y.load(acquire));
y.store(1, release);               | r = x;
```

Expert concurrency: fences avoid excess synchronisation

```
// sender
x = ...
y.store(1, release);
```

```
// receiver
while (0 == y.load(acquire));
r = x;
```

```
// sender
x = ...
y.store(1, release);
```

```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```

Expert concurrency: The fenced release-acquire idiom

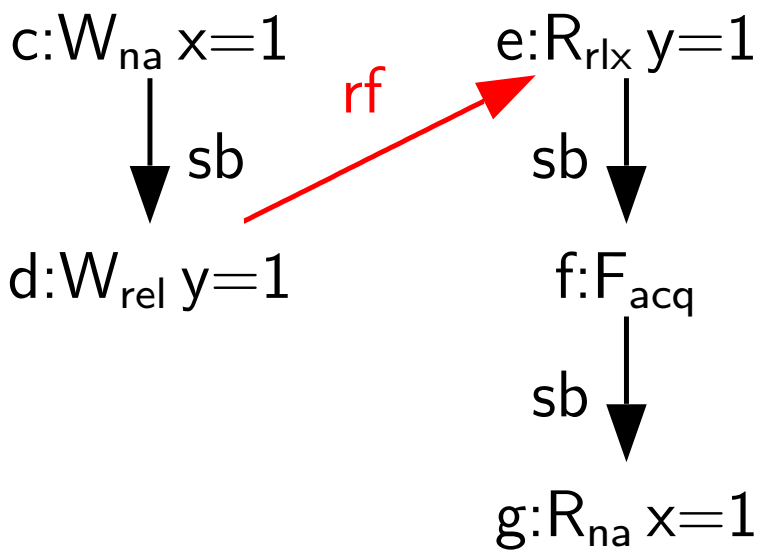
```
// sender
x = ...
y.store(1, release);
```

```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```

Expert concurrency: The fenced release-acquire idiom

```
// sender
x = ...
y.store(1, release);
```

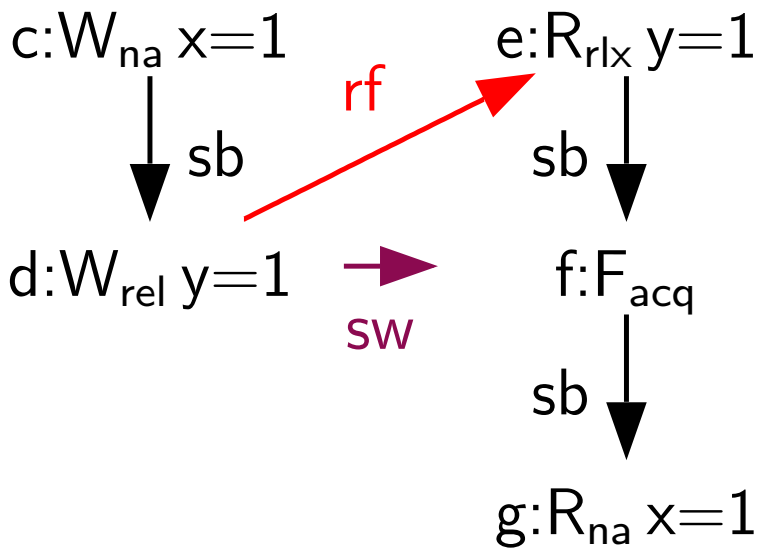
```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```



Expert concurrency: The fenced release-acquire idiom

```
// sender
x = ...
y.store(1, release);
```

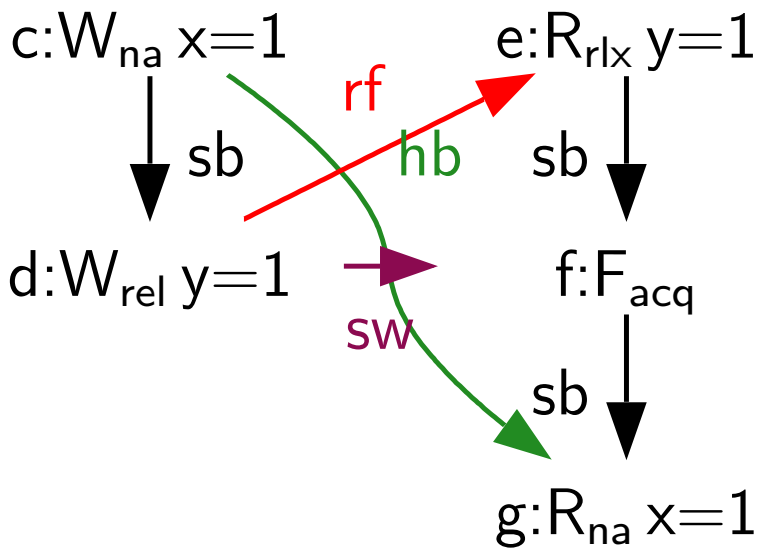
```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```



Expert concurrency: The fenced release-acquire idiom

```
// sender
x = ...
y.store(1, release);
```

```
// receiver
while (0 == y.load(relaxed));
fence(acquire);
r = x;
```



Expert concurrency: modification order

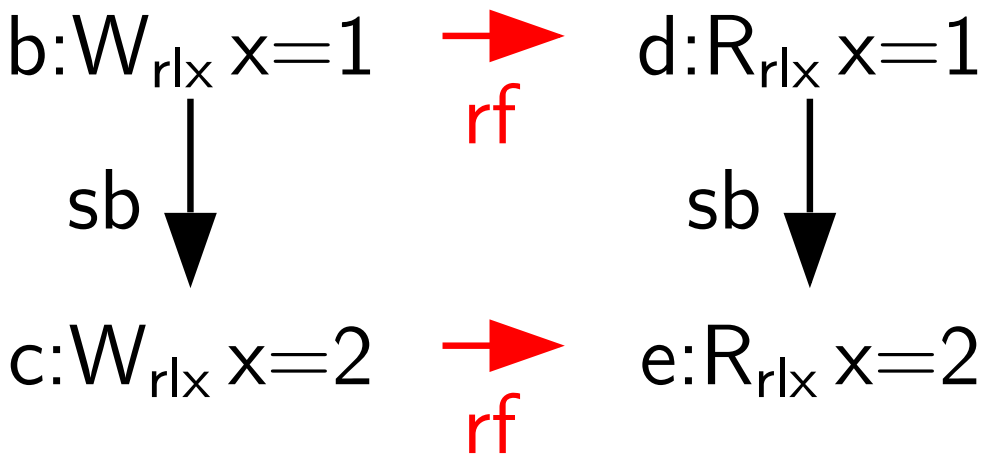
Modification order is a per-location total order over atomic writes of any memory order.

```
x.store(1, relaxed);  |  x.load(relaxed);
x.store(2, relaxed);  |  x.load(relaxed);
```

Expert concurrency: modification order

Modification order is a per-location total order over atomic writes of any memory order.

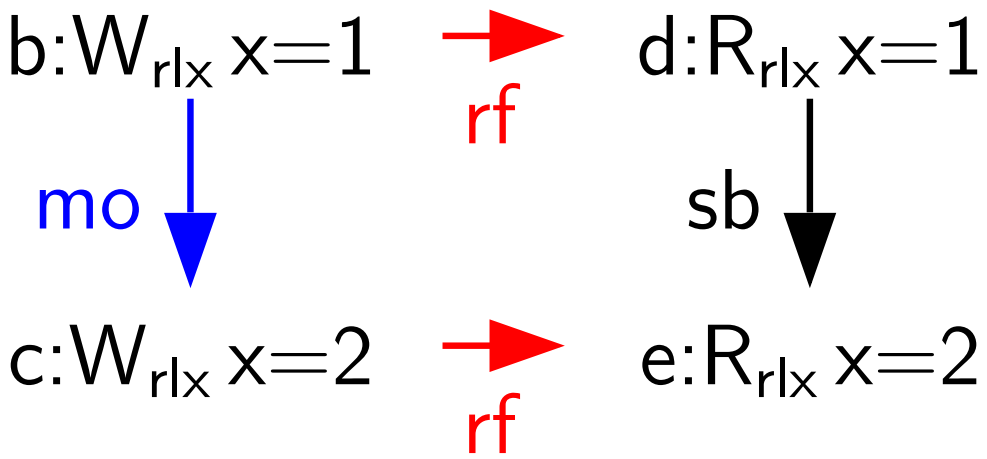
```
x.store(1, relaxed);  |  x.load(relaxed);
x.store(2, relaxed);  |  x.load(relaxed);
```



Expert concurrency: modification order

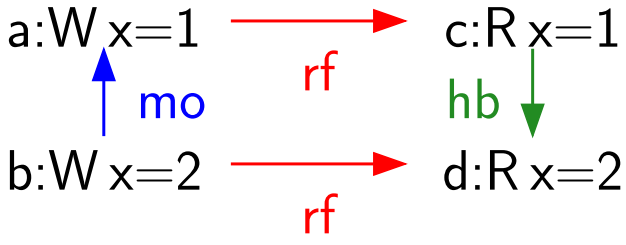
Modification order is a per-location total order over atomic writes of any memory order.

```
x.store(1, relaxed);  |  x.load(relaxed);
x.store(2, relaxed);  |  x.load(relaxed);
```

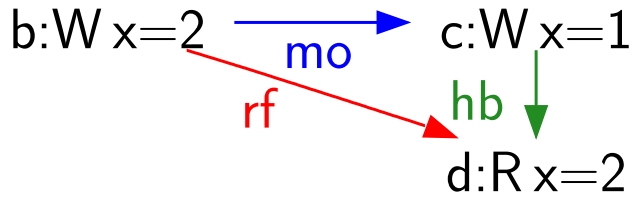


Coherence and atomic reads

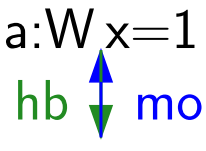
All forbidden!



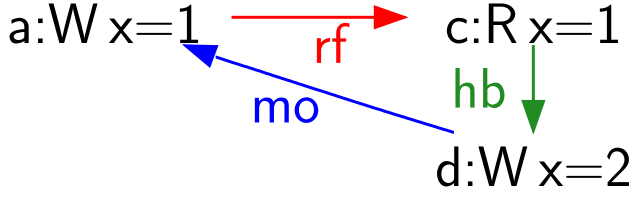
CoRR



CoWR



CoWW



CoRW

Atomics cannot read from later writes in happens before.

Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

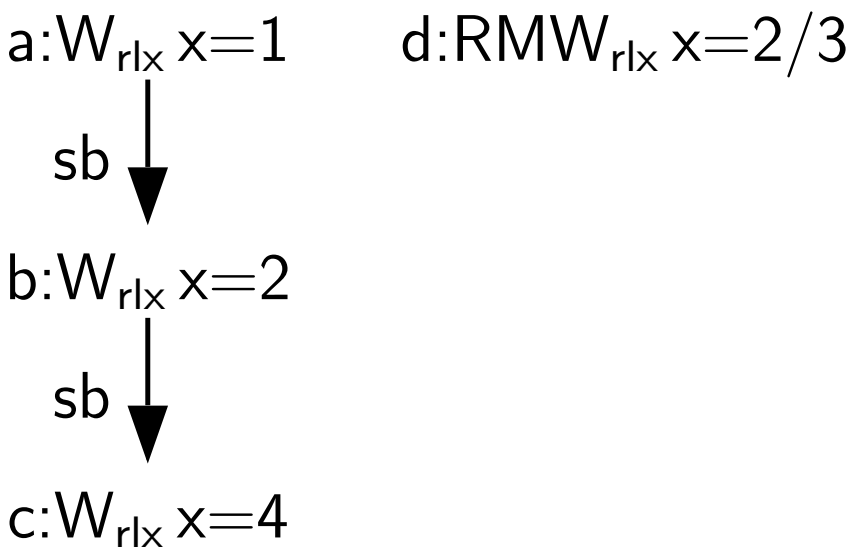
```
x.store(1, relaxed); | compare_exchange(&x, 2, 3, relaxed, relaxed);
x.store(2, relaxed); |
x.store(4, relaxed); |
```

Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

```
x.store(1, relaxed); | compare_exchange(&x, 2, 3, relaxed, relaxed);
x.store(2, relaxed); |
x.store(4, relaxed); |
```

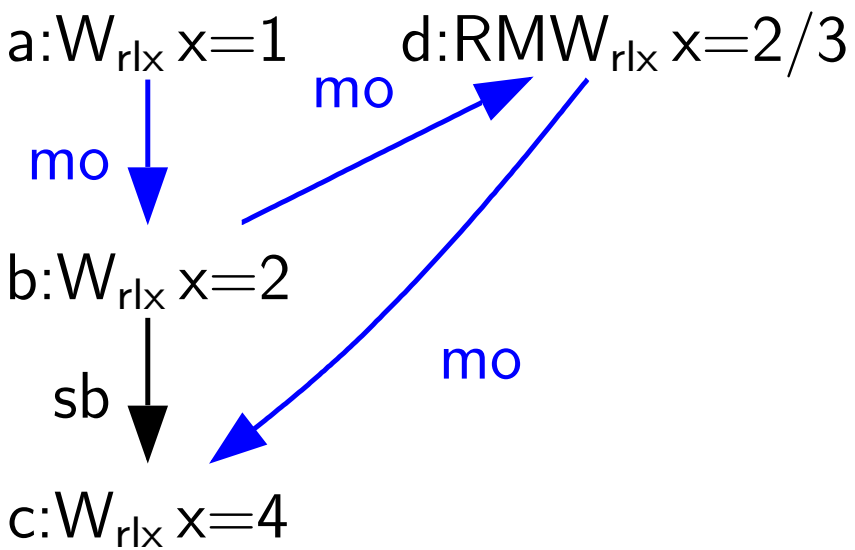


Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

```
x.store(1, relaxed); | compare_exchange(&x, 2, 3, relaxed, relaxed);
x.store(2, relaxed); |
x.store(4, relaxed); |
```

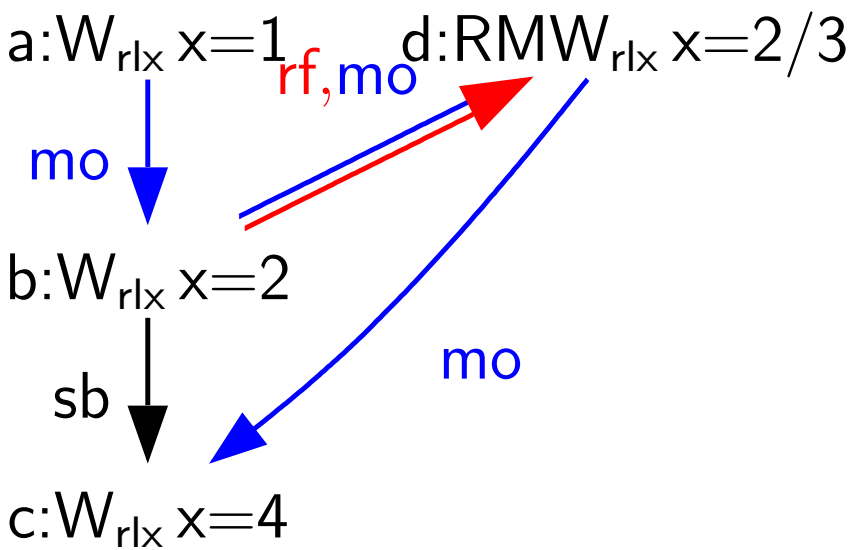


Read-modify-writes

A successful `compare_exchange` is a read-modify-write.

Read-modify-writes read the last write in mo:

```
x.store(1, relaxed); | compare_exchange(&x, 2, 3, relaxed, relaxed);
x.store(2, relaxed); |
x.store(4, relaxed); |
```



Very expert concurrency: consume

Weaker than acquire

Stronger than relaxed

Non-transitive happens before! (only fully transitive through data dependence, dd)

How may a program execute in the model?

1. $P \mapsto E_1, ..., E_n$

How may a program execute in the model?

1. $P \mapsto E_1, ..., E_n$

— find memory accesses with thread local semantics

How may a program execute in the model?

1. $P \mapsto E_1, ..., E_n$

— find memory accesses with thread local semantics

2. $E_i \mapsto X_{i1}, ..., X_{im}$

How may a program execute in the model?

1. $P \mapsto E_1, ..., E_n$

— find memory accesses with thread local semantics

2. $E_i \mapsto X_{i1}, ..., X_{im}$

— calculate happens before, check the rules

How may a program execute in the model?

1. $P \mapsto E_1, ..., E_n$

— find memory accesses with thread local semantics

2. $E_i \mapsto X_{i1}, ..., X_{im}$

— calculate happens before, check the rules

3. is there an X_{ij} with a race?

How may a program execute in the model?

1. $P \mapsto E_1, ..., E_n$

— find memory accesses with thread local semantics

2. $E_i \mapsto X_{i1}, ..., X_{im}$

— calculate happens before, check the rules

3. is there an X_{ij} with a race?

— if so then have undefined behaviour

CPPMEM - demo!

Code in, all executions out

CPPMEM - demo!

Code in, all executions out

How may a program execute in CPPMEM?

1. $P \mapsto E_1, ..., E_n$ — tracking constraints

2. $E_i \mapsto X_{i1}, ..., X_{im}$ — automatically uses formal model

3. is there an X_{ij} with a race?

The model as a whole

C1x and C++11 support many modes of programming:

- sequential

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire
- with relaxed, fences and the rest

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire
- with relaxed, fences and the rest
- with all of the above plus consume

The model as a whole

C1x and C++11 support many modes of programming:

- sequential
- concurrent with locks
- with `seq_cst` atomics
- with release and acquire
- with relaxed, fences and the rest
- with all of the above plus consume

Mathematizing C++ concurrency. M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. In Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), 2011.

The full model

Theorems

Are C1x and C++11 hopelessly complicated?

Programmers cannot be given this model!

With a formal definition, we can do proof, and even mechanise it.

What do we need to prove?

Are C1x and C++11 hopelessly complicated?

Programmers cannot be given this model!

With a formal definition, we can do proof, and even mechanise it.

What do we need to prove?

- implementability
- simplifications
- libraries

Implementability

Can we compile to x86?

Implementability

Can we compile to x86?

Operation	x86 Implementation
load(non-seq_cst)	mov
load(seq_cst)	mov
store(non-seq_cst)	mov
store(seq_cst)	mov; mfence
fence(non-seq_cst)	no-op
fence(seq_cst)	mfence

x86-TSO is stronger and simpler.

Top level comparison

Recall the C/C++ semantics for program P:

1. $P \mapsto E_1, ..., E_n$,

Top level comparison

Recall the C/C++ semantics for program P:

1. $P \mapsto E_1, ..., E_n$, each an E_{thread}

Top level comparison

Recall the C/C++ semantics for program P:

1. $P \mapsto E_1, ..., E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, ..., X_{im}$,

Top level comparison

Recall the C/C++ semantics for program P:

1. $P \mapsto E_1, ..., E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, ..., X_{im}$, collectively X_{witness}

Top level comparison

Recall the C/C++ semantics for program P:

1. $P \mapsto E_1, ..., E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, ..., X_{im}$, collectively X_{witness}
3. is there an X_{ij} with a race? (actually, several kinds...)

Top level comparison

Recall the C/C++ semantics for program P:

1. $P \mapsto E_1, ..., E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, ..., X_{im}$, collectively X_{witness}
3. is there an X_{ij} with a race? (actually, several kinds...)

In x86-TSO:

Events and dependencies, E_{x86} are analogous to E_{thread}.

Top level comparison

Recall the C/C++ semantics for program P:

1. $P \mapsto E_1, ..., E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, ..., X_{im}$, collectively X_{witness}
3. is there an X_{ij} with a race? (actually, several kinds...)

In x86-TSO:

Events and dependencies, E_{x86} are analogous to E_{thread}.
Execution witnesses, X_{x86} are analogous to X_{witness}.

Top level comparison

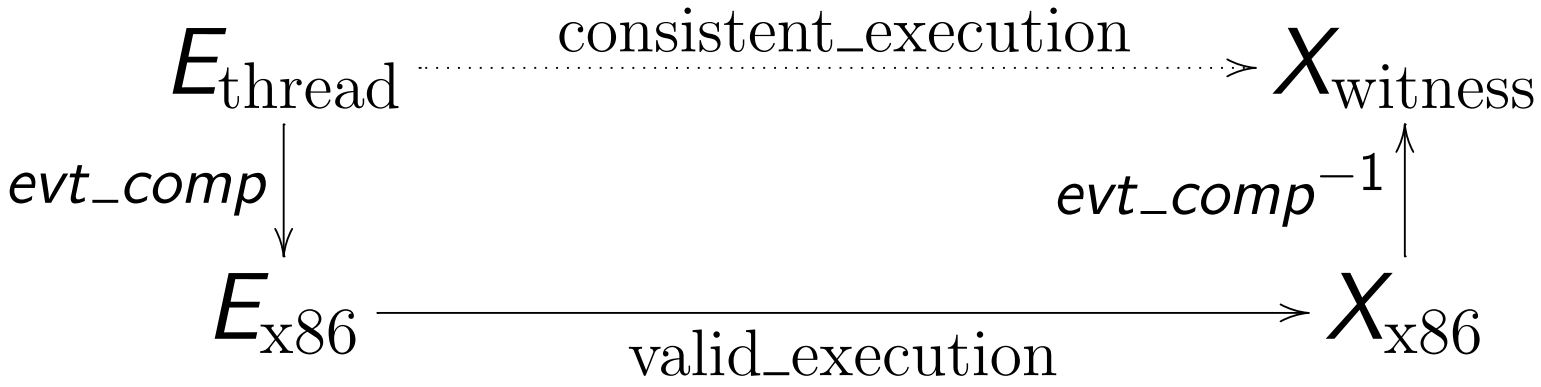
Recall the C/C++ semantics for program P:

1. $P \mapsto E_1, ..., E_n$, each an E_{thread}
2. $E_i \mapsto X_{i1}, ..., X_{im}$, collectively X_{witness}
3. is there an X_{ij} with a race? (actually, several kinds...)

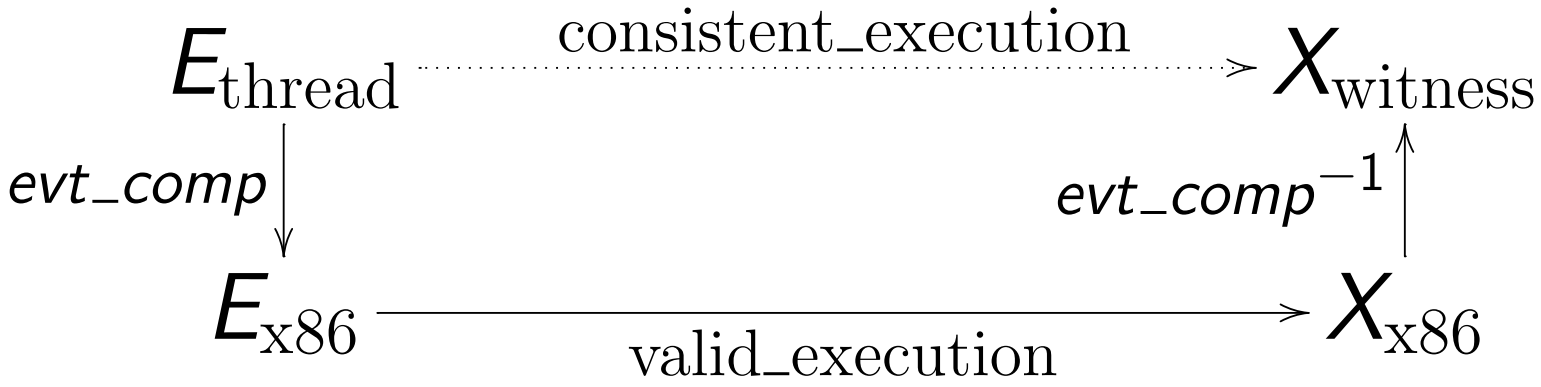
In x86-TSO:

Events and dependencies, E_{x86} are analogous to E_{thread}.
Execution witnesses, X_{x86} are analogous to X_{witness}.
There is not a DRF semantics.

Theorem


E_{thread}
consistent_execution
X_{witness}
evt_comp
evt_comp^{-1}
E_{x86}
valid_execution
X_{x86}

Theorem



We have a mechanised proof that C1x/C++11 behaviour is preserved.

Implementability

Can we compile to IBM Power?

Implementability

Can we compile to IBM Power?

C++0x Operation	POWER Implementation
Non-atomic Load	`ld`
Load Relaxed	`ld`
Load Consume	`ld` (and preserve dependency)
Load Acquire	`ld; cmp; bc; isync`
Load Seq Cst	`sync; ld; cmp; bc; isync`
Non-atomic Store	`st`
Store Relaxed	`st`
Store Release	`lwsync; st`
Store Seq Cst	`sync; st`

We have a hand proof that C1x/C++11 behaviour is preserved.

Implementability

Can we compile to IBM Power?

C++0x Operation	POWER Implementation
Non-atomic Load	`ld`
Load Relaxed	`ld`
Load Consume	`ld` (and preserve dependency)
Load Acquire	`ld; cmp; bc; isync`
Load Seq Cst	`sync; ld; cmp; bc; isync`
Non-atomic Store	`st`
Store Relaxed	`st`
Store Release	`lwsync; st`
Store Seq Cst	`sync; st`

We have a hand proof that C1x/C++11 behaviour is preserved.

Clarifying and compiling C/C++ concurrency: from C++0x to POWER. M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. In Proc. 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2012.

mo_seq_cst

The compiler must ensure that `mo_seq_cst` atomics have SC semantics.

```
x.store(1, mo_seq_cst);          | y.store(1, mo_seq_cst);
r1 = y.load(mo_seq_cst);         | r2 = x.load(mo_seq_cst);
```

The program above cannot end with `r1 = r2 = 0`.

Sample compilation on x86:
store: `mov; mfence`
load: `mov`

Sample compilation on Power:
store: `sync; st`
load: `sync; ld; cmp; bc; isync`

mo_release / mo_acquire

Supports fast implementation of the message passing idiom.

```
x = 1;                          | r1 = y.load(mo_acquire);
y.store(1, mo_release);         | r2 = x;
```

The program above cannot end with `r1 = 1` and `r2 = 0`.

Accesses to the data could be reordered/optimised with `mo_relaxed`.

Sample compilation on x86:
store: `mov`
load: `mov`

Sample compilation on Power:
store: `lwsync; st`
load: `ld; cmp; bc; isync`

mo_release / mo_consume

Supports faster implementation of the message passing idiom on Power.

```
x = 1;                              | r1 = y.load(mo_consume);
y.store(&x, mo_release);            | r2 = *r1;
```

The program above cannot end with `r1 = &x` and `r2 = 0`.

The two loads have an address dependency - Power won't reorder them.

Sample compilation on x86:
store: `mov`
load: `mov`

Sample compilation on Power:
store: `lwsync; st`
load: `ld`

Refinements to the model and standards

Simplifications and meta-theorems

Full model – *visible sequences of side effects* are unneeded.

Simplifications and meta-theorems

Full model – *visible sequences of side effects* are unneeded.

Derivative models:

- without consume, happens-before is transitive.
- DRF programs using only `seq_cst` atomics are SC (false).

Simplifications and meta-theorems

Full model – *visible sequences of side effects* are unneeded.

Derivative models:

- without consume, happens-before is transitive.
- DRF programs using only `seq_cst` atomics are SC (false).

```
atomic_int x = 0;
atomic_int y = 0;
if (1 == x.load(seq_cst))  |  if (1 == y.load(seq_cst))
  atomic_init(&y, 1);      |    atomic_init(&x, 1);
```

`atomic_init` is a non-atomic write, and in C1x/C++11 they race...

The current state of the standard

Fixed:

- Happens-before
- Coherence
- `seq_cst` atomics were broken

The current state of the standard

Fixed:

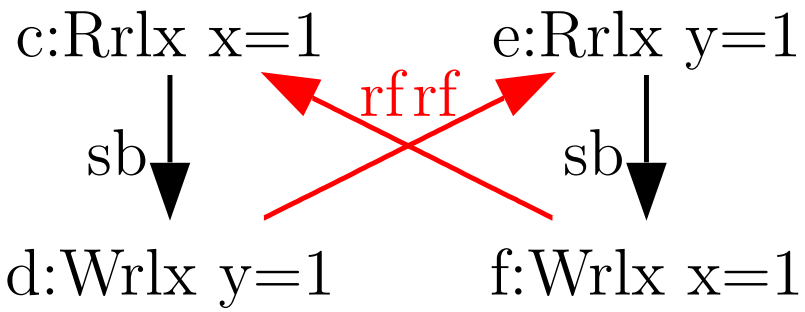
- Happens-before
- Coherence
- `seq_cst` atomics were broken

Not fixed:

- Self satisfying conditionals

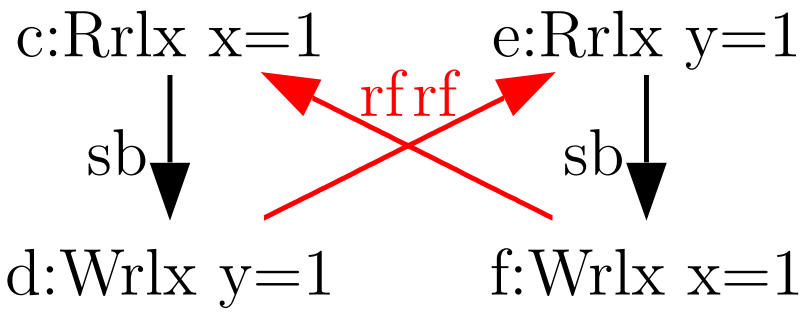
Self-satisfying conditionals

```
r1 = x.load(mo_relaxed);          | r2 = y.load(mo_relaxed);
if (r1 == 42)                     | if (r2 == 42)
  y.store(r1, mo_relaxed);        |   x.store(42, mo_relaxed);
```



Self-satisfying conditionals

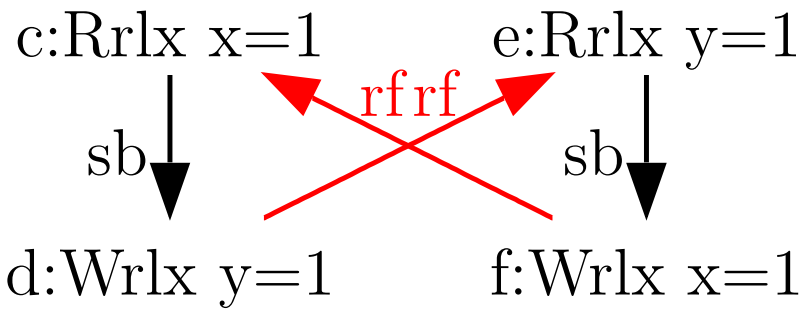
```
r1 = x.load(mo_relaxed);          | r2 = y.load(mo_relaxed);
if (r1 == 42)                     | if (r2 == 42)
  y.store(r1, mo_relaxed);        |   x.store(42, mo_relaxed);
```



"However, implementations **should** not allow such behavior."

Self-satisfying conditionals

```
r1 = x.load(mo_relaxed);          | r2 = y.load(mo_relaxed);
if (r1 == 42)                     | if (r2 == 42)
  y.store(r1, mo_relaxed);        |   x.store(42, mo_relaxed);
```



"However, implementations **should** not allow such behavior."

"should not" means "is allowed to" in the standard!

...but it's not all bad!

Syntactic divide supported by simpler memory models.

Increasingly reasonable, consistent specification.

Remaining problems far less serious than Java.

Implementable above key architectures.

Thanks!