Relaxed Memory:
The Specification Design Space

Mark Batty

University of Cambridge

Fortran meeting, Delft, 25 June 2013

An ideal specification

Unambiguous

Easy to understand

Sound w.r.t. experimentally observable behaviour

Supportive of desirable programming idioms

Efficiently implementable now and in the future

An ideal specification

Unambiguous

Easy to understand

Sound w.r.t. experimentally observable behaviour

Supportive of desirable programming idioms

Efficiently implementable now and in the future

Testable?

How to define a memory model

Separate *instruction semantics* and the *memory model*

There are two prevailing styles of MM:

Operational models (x86, Power/ARM):
- Run stepwise through an execution
- Maintain a machine state (buffers, lists)

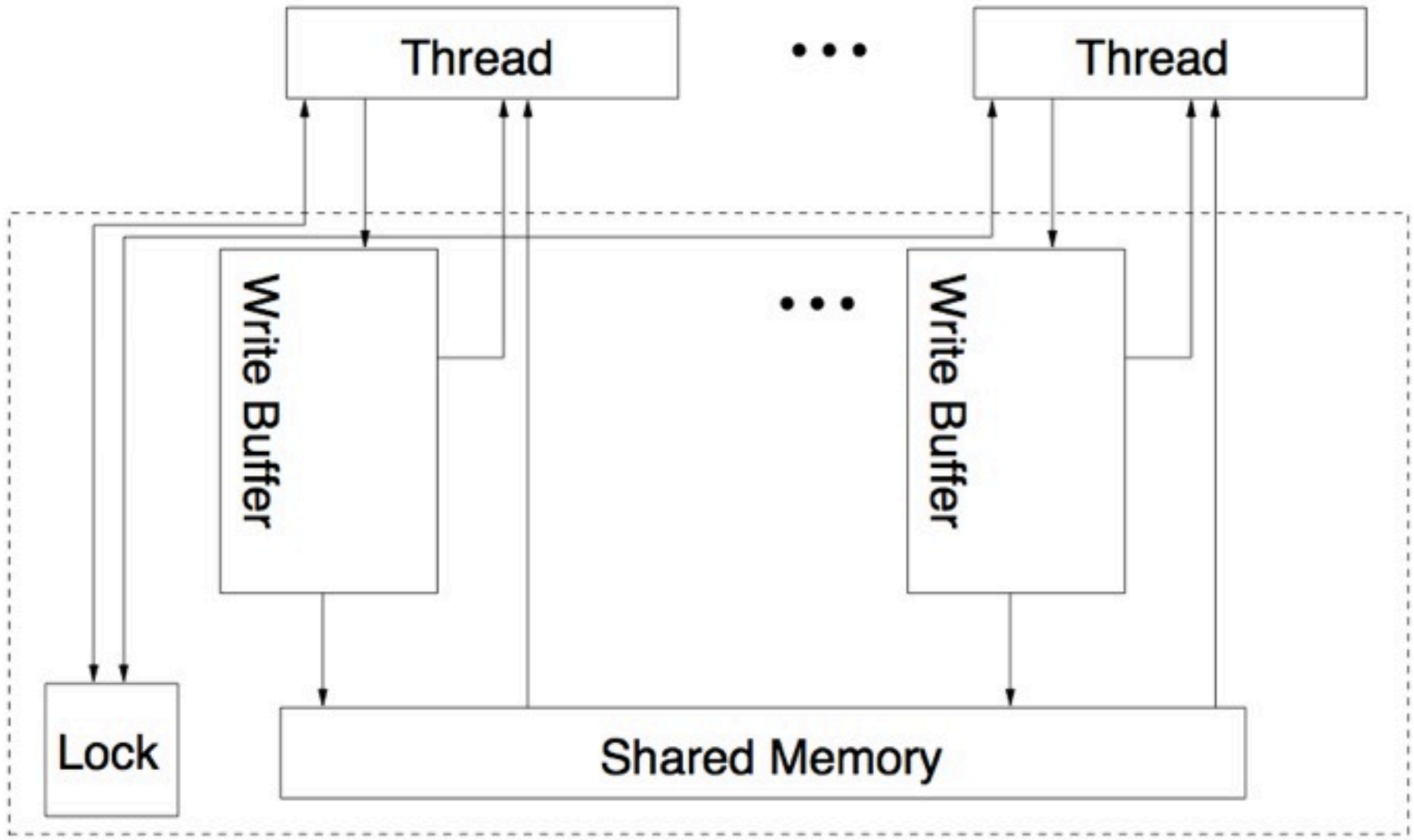
Axiomatic models (C/C++11, Java):
- Enumerate whole program executions
- The model decides which ones are allowed

X86 abstract machine

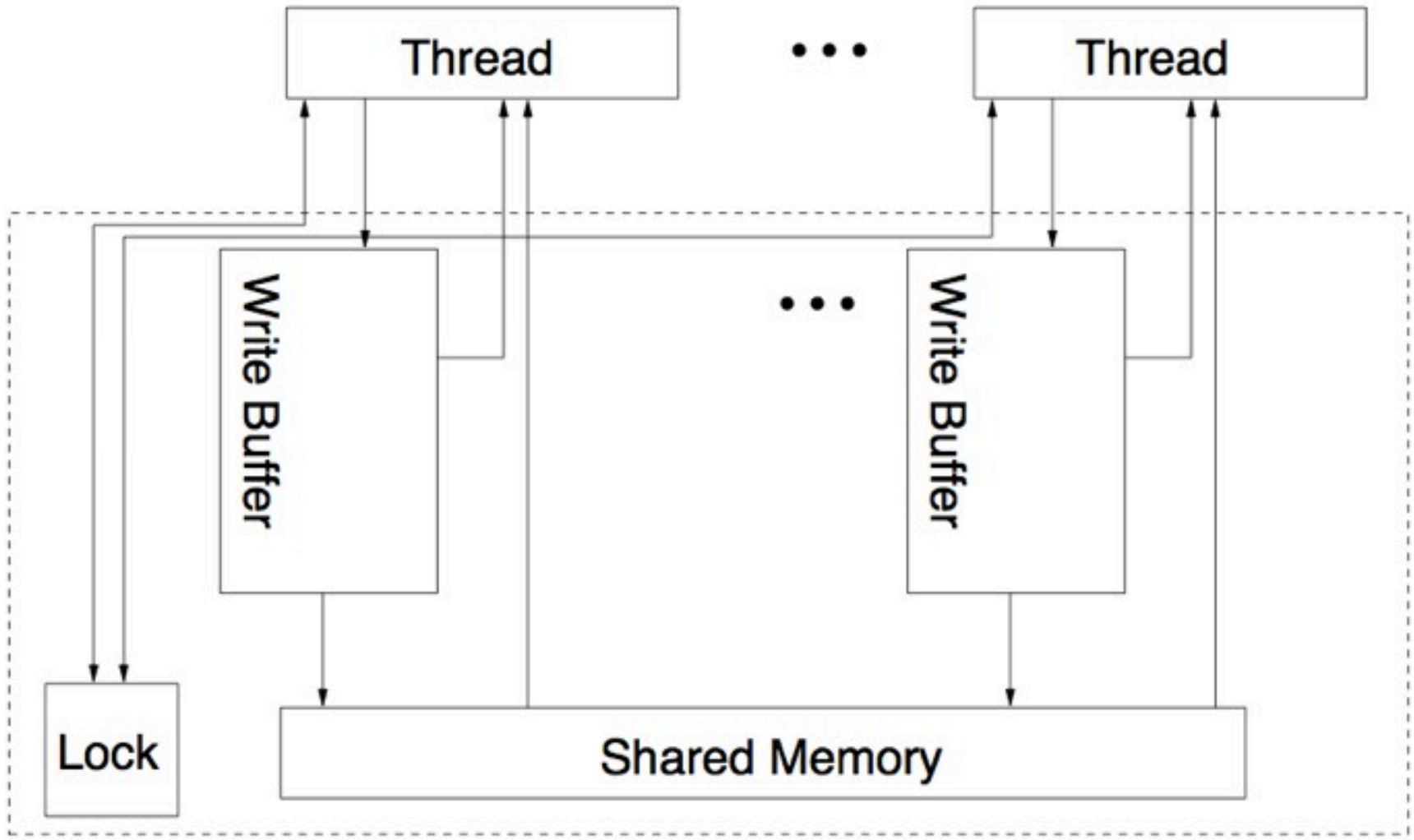
A very simple operational memory model

Nearly sequentially consistent

X86 abstract machine



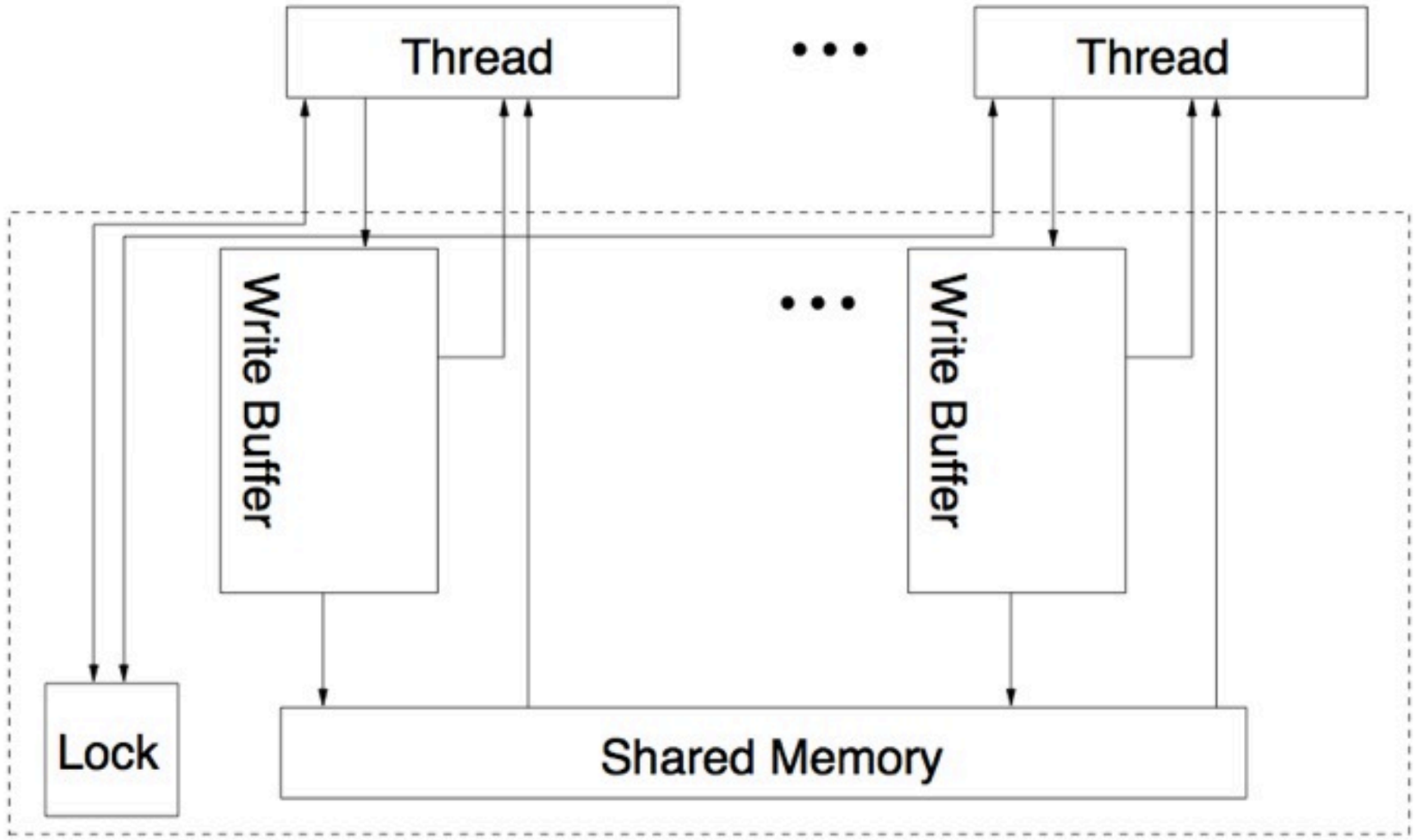
X86 abstract machine



SB

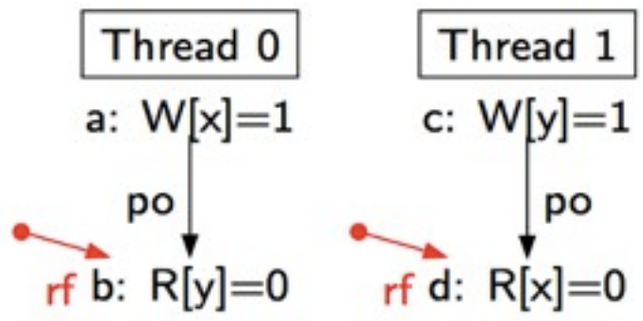
Thread 0	Thread 1
x=1 r1=y //reads 0	y=1 r2=x //reads 0

X86 abstract machine



SB

Thread 0	Thread 1
x=1 r1=y //reads 0	y=1 r2=x //reads 0

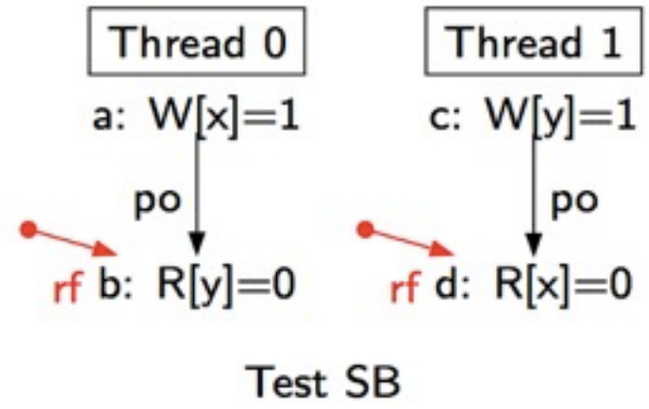


Test SB

Aside: a litmus test

SB

Thread 0	Thread 1
x=1 r1=y //reads 0	y=1 r2=x //reads 0



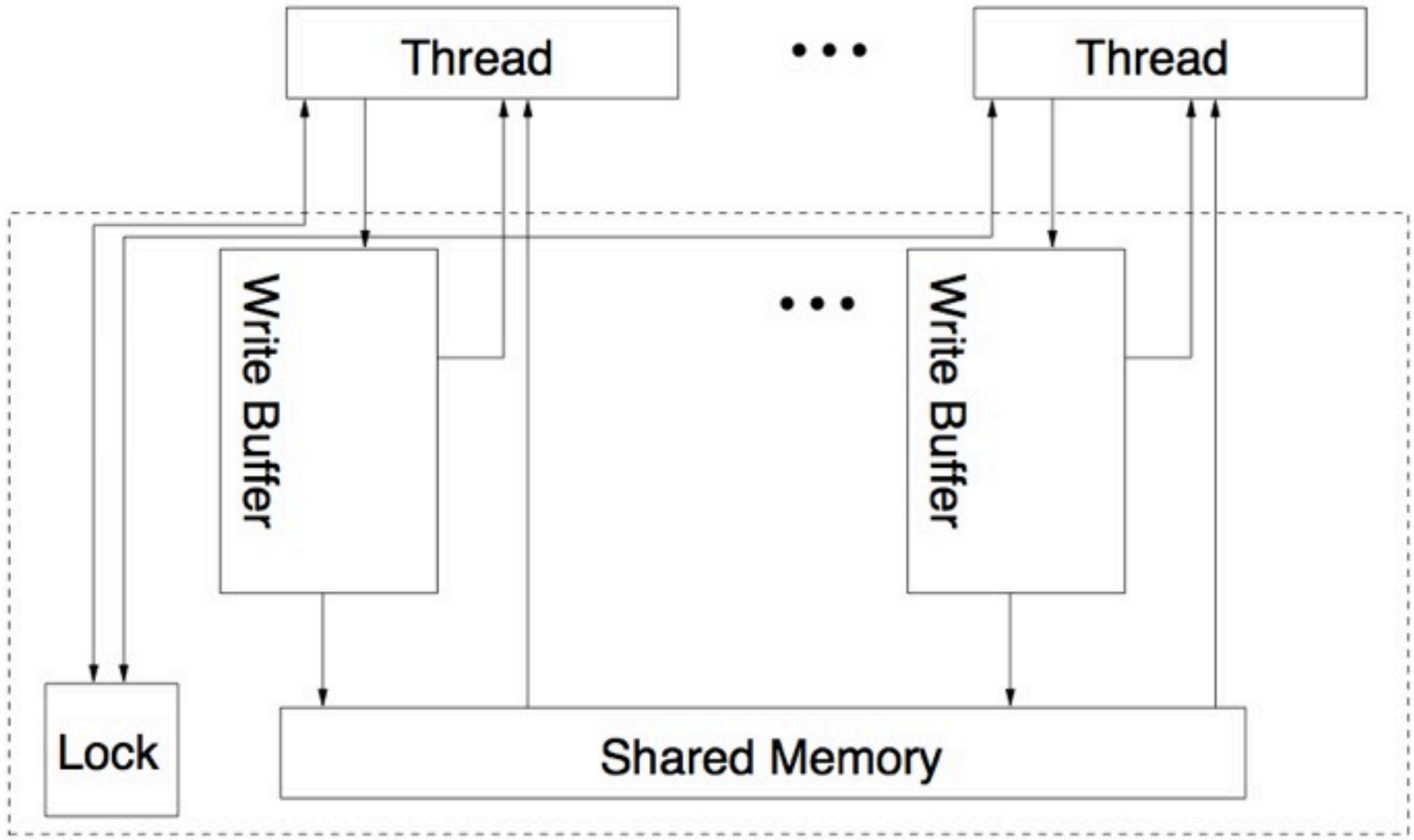
A minimal example showcasing some relaxed behaviour

Can be used to compare very different models

(observable in 11G/167G runs on Power)

SB is one of the most tame relaxed behaviours

Restoring SC with Fences



SB+MFENCE

Thread 0	Thread 1
x=1 MFENCE r1=y	y=1 MFENCE r2=x

Cannot read 0,0

IBM Power and ARM

Power and ARM share their memory model

Much more relaxed than x86

The programer can observe many relaxed behaviours

IBM Power is more relaxed

Thread 0	Thread 1
`x = 1`	`while (y==0) {};`
`y = 1`	`r = x`

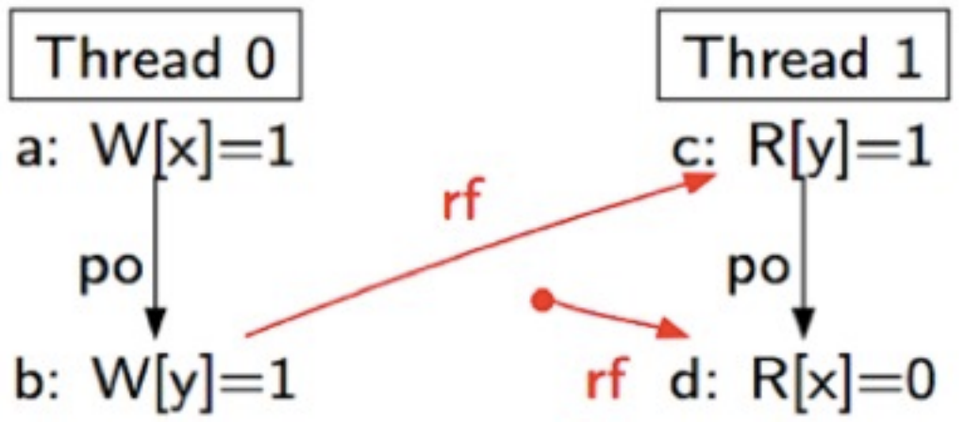
Observable behaviour: `r = 0`

Message passing (MP)

IBM Power is more relaxed

Thread 0	Thread 1
`x = 1`	`while (y==0) {};`
`y = 1`	`r = x`

Observable behaviour: `r = 0`



Forbidden on SC and x86-TSO
Allowed and **observed** on Power

1.7G/167G runs

IBM Power is more relaxed

Three possible reasons (at least) for y==1 and x==0

Thread 0	Thread 1
`x = 1`	`while (y==0) {};`
`y = 1`	`r = x`

Observable behaviour: `r = 0`

1. the two writes are performed in opposite order

reordering store buffers

2. the two reads are performed in opposite order

load reorder buffers / speculation

3. writes are propagated between threads out of order

interconnects partitioned by address (cache lines)

IBM Power is more relaxed

Three possible reasons (at least) for y==1 and x==0

Thread 0	Thread 1
`x = 1`	`while (y==0) {};`

Power does all three!

1. the two v

reordering store buffers

2. the two reads are performed in opposite order

load reorder buffers / speculation

3. writes are propagated between threads out of order

interconnects partitioned by address (cache lines)

Power and ARM programming

Visible behaviour much weaker and subtle than x86.

Basically, program order is **not preserved** unless:

- writes to the *same* memory location (coherence)
- there is an *address dependency* between two loads
  data-flow path through registers and arith/logical operations from the value of the first load to the address of the second
- there is an *address or data or control dependency* between a load and a store
  as above, or to the value store, or data flow to the test of an intermediate conditional branch
- you use a *synchronisation instruction* (ptesync, hwsync, lwsync, eieio, mbar, isync).

Programming language memory models

This is a harder problem. The model must:

- be ordered enough to program atop
- provide clear intuitions to programmers

- be efficiently implementable on **all** targets
- allow for compiler optimisations

Programming language memory models

This is a harder problem. The model must:

- be ordered enough to program atop
- provide clear intuitions to programmers

Strong

vs

- be efficiently implementable on **all** targets
- allow for compiler optimisations

Relaxed

Programming language memory models

This is a harder problem. The model must:

- be ordered enough to program atop
- provide clear intuitions to programmers

Strong

vs

- be efficiently implementable on **all** targets
- allow for compiler optimisations

Relaxed

Optimisations cause big problems in C/C++11!

C/C++11 concurrency

Formal model developed in discussion with WG21

An axiomatic data-race free model

Semantics defined by sets of execution graphs

More relaxed than Power

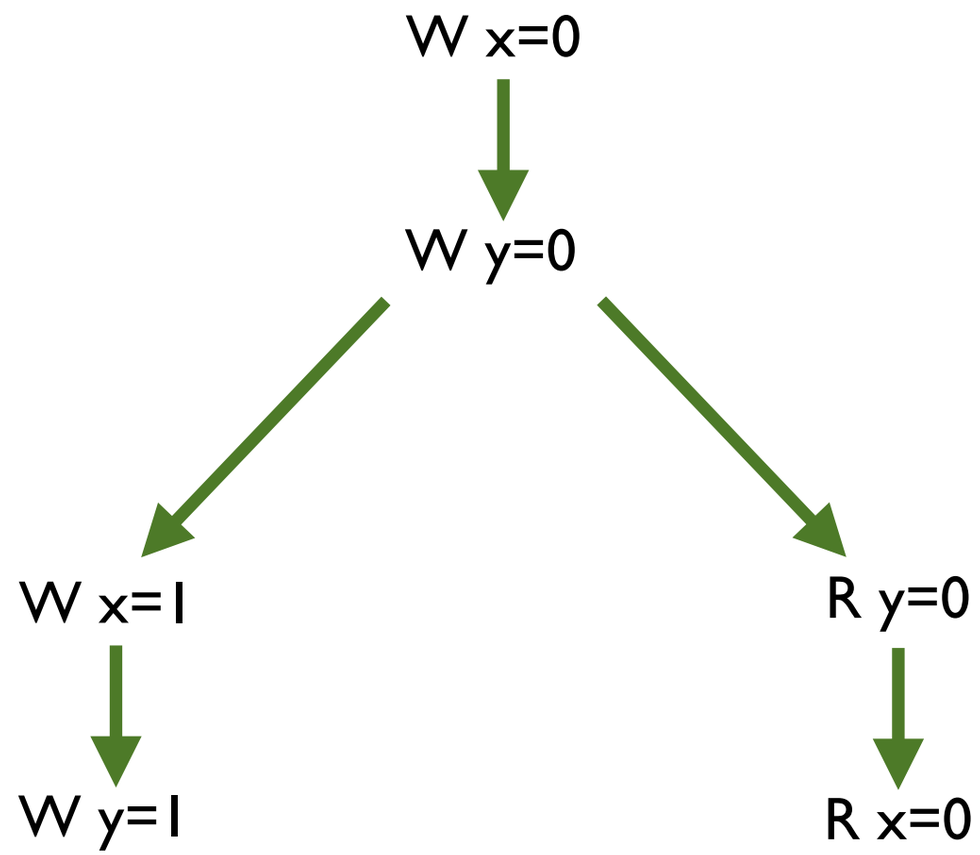
A simple program

```
int r;
int x=0;
int y=0;
x = 1; || r = y;
y = 1; || r = x;
```

Memory model

The semantics of a program is a set of execution graphs, here is one:

```
int r;
int x=0;
int y=0;
x = 1; || r = y;
y = 1; || r = x;
```

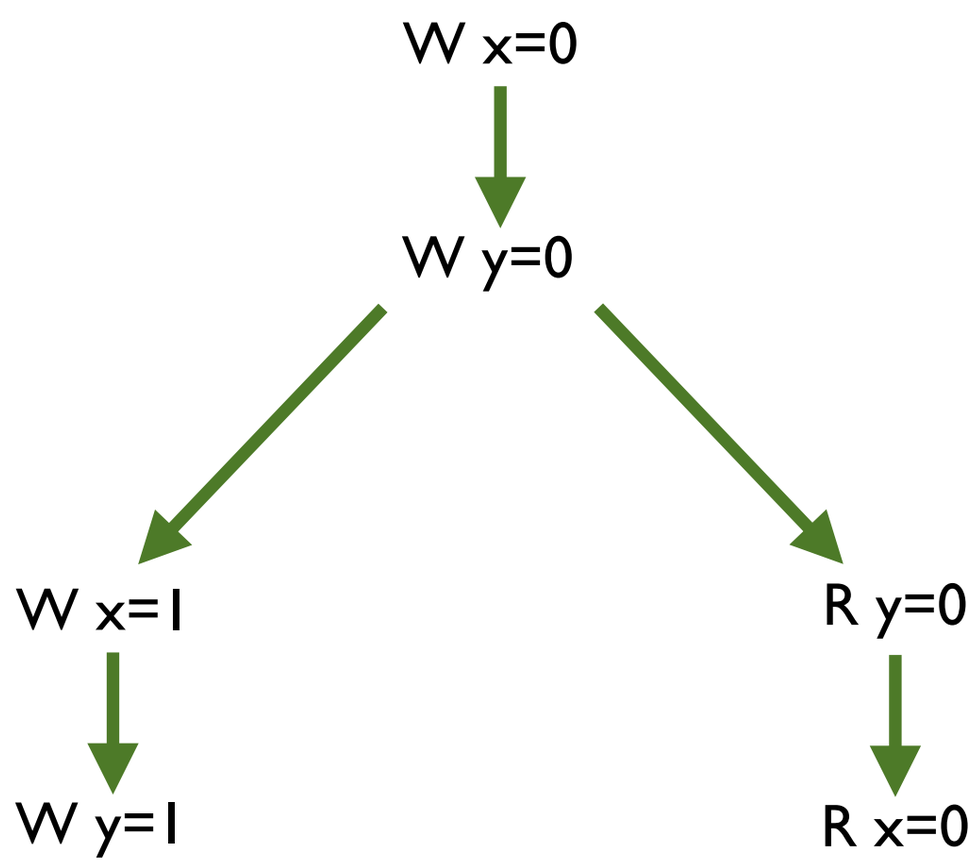


Memory model

The semantics of a program is a set of execution graphs, here is one:

```
int x=0;

int y=0;
```

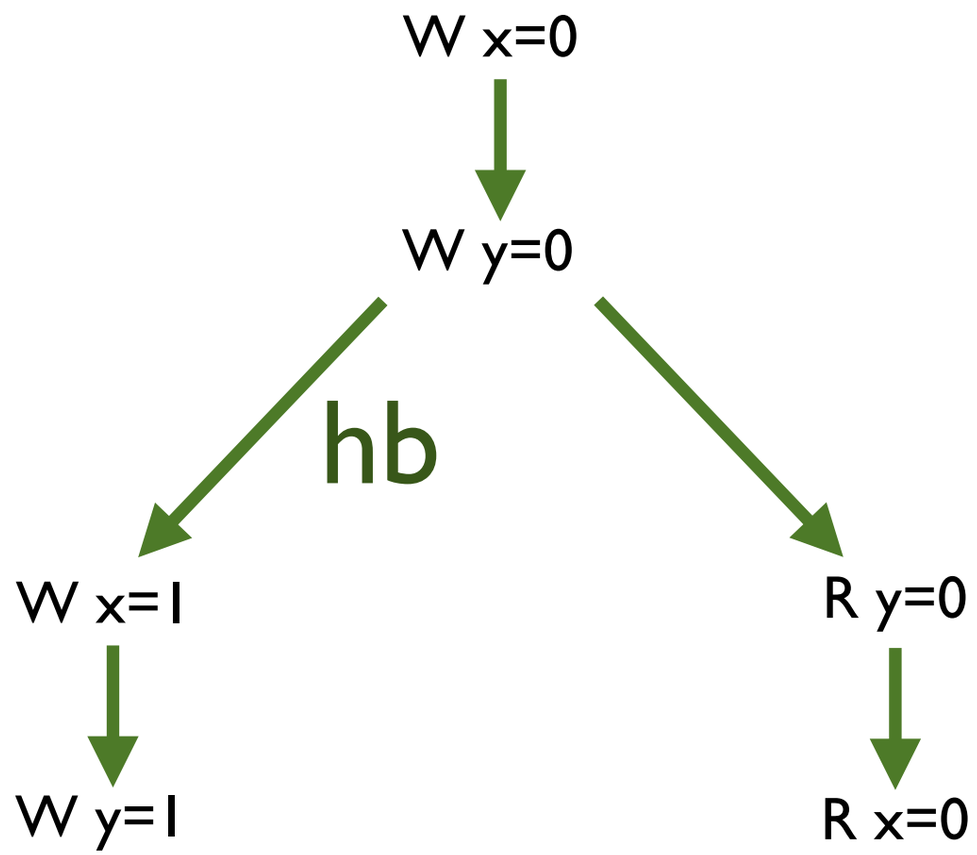
```
x = 1;  ||  r = y;

y = 1;  ||  r = x;
```



Happens-before

Happens-before is partial.

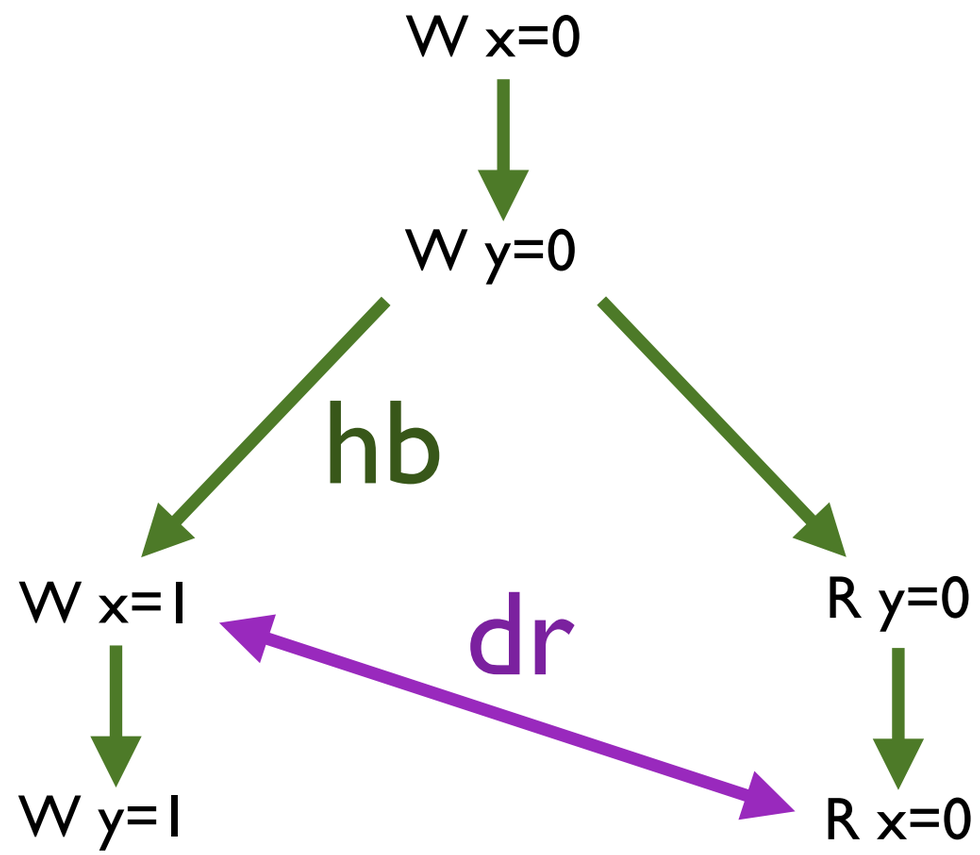
```
int r;
int x=0;
int y=0;
x = 1; || r = y;
y = 1; || r = x;
```



Oh no! A data race

hb unordered reads and writes are a race.

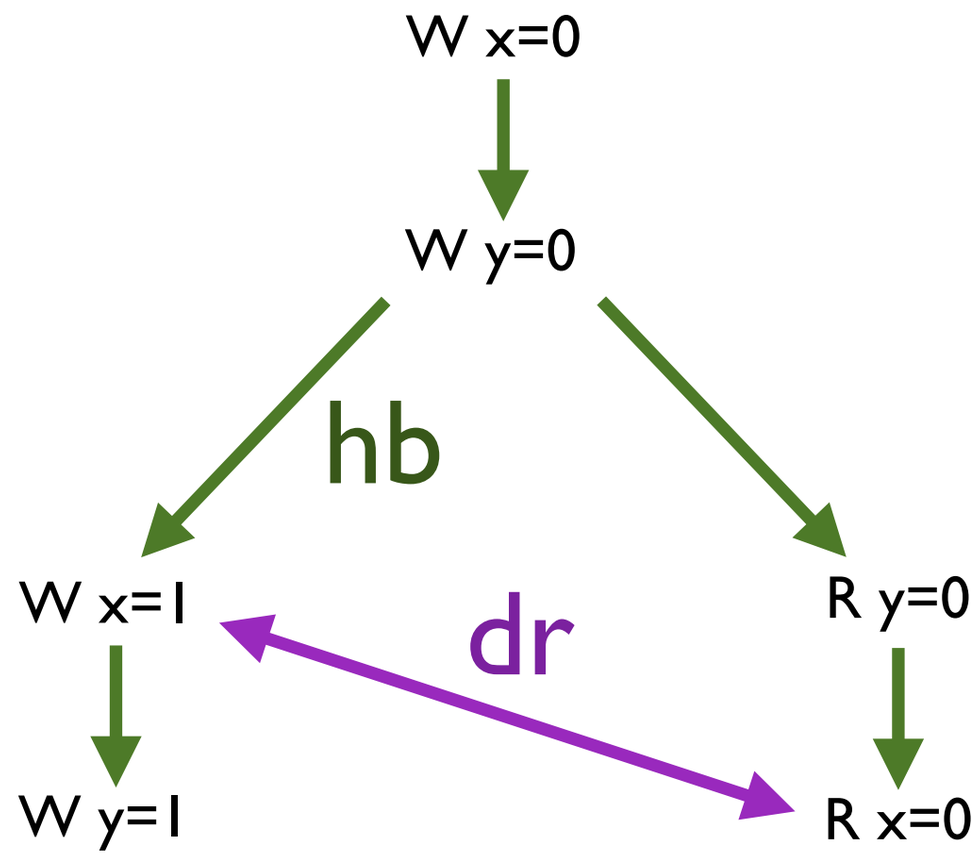
```
int r;
int x=0;
int y=0;
x = 1; || r = y;
y = 1; || r = x;
```



Oh no! A data race

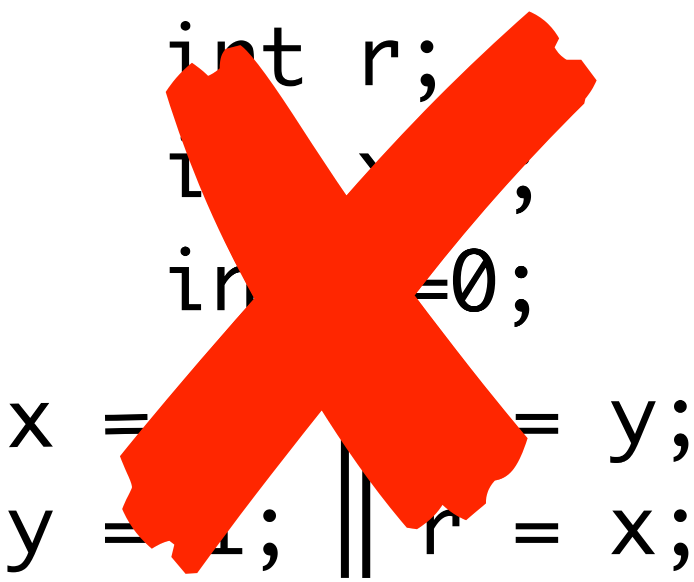
Racy programs have undefined behaviour.

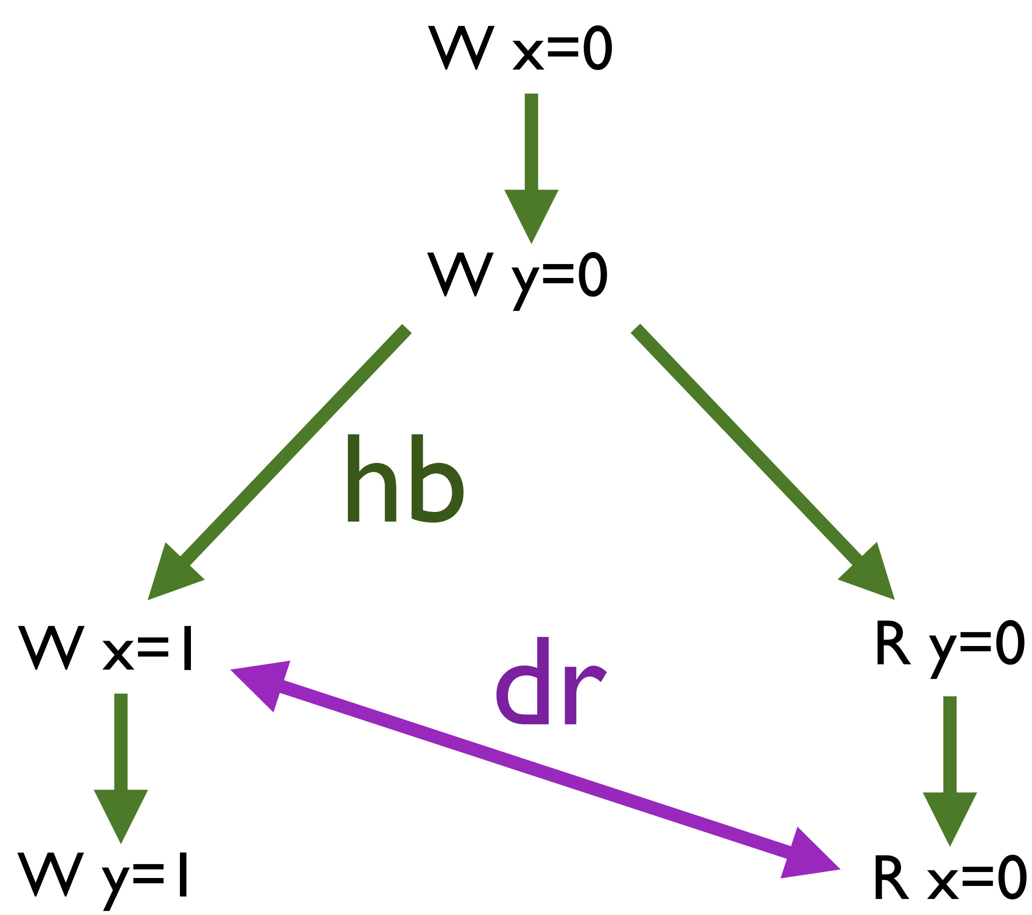
```
int r;
int x=0;
int y=0;
x = 1; || r = y;
y = 1; || r = x;
```



Oh no! A data race

Racy programs have undefined behaviour.

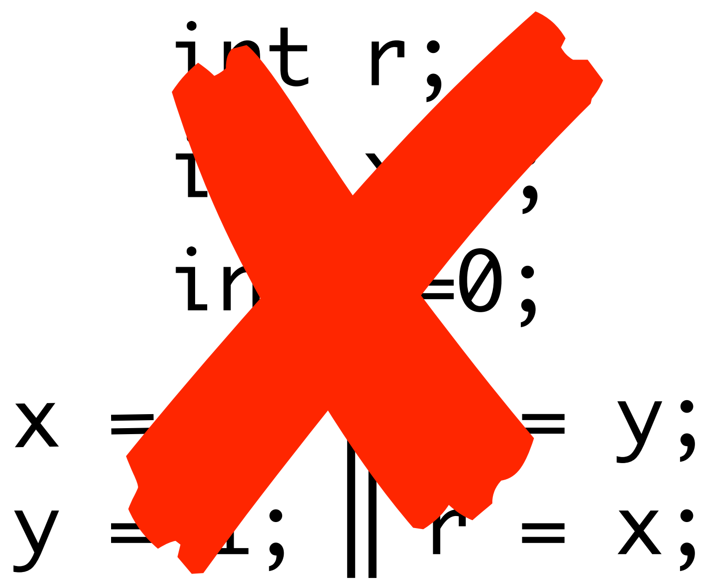


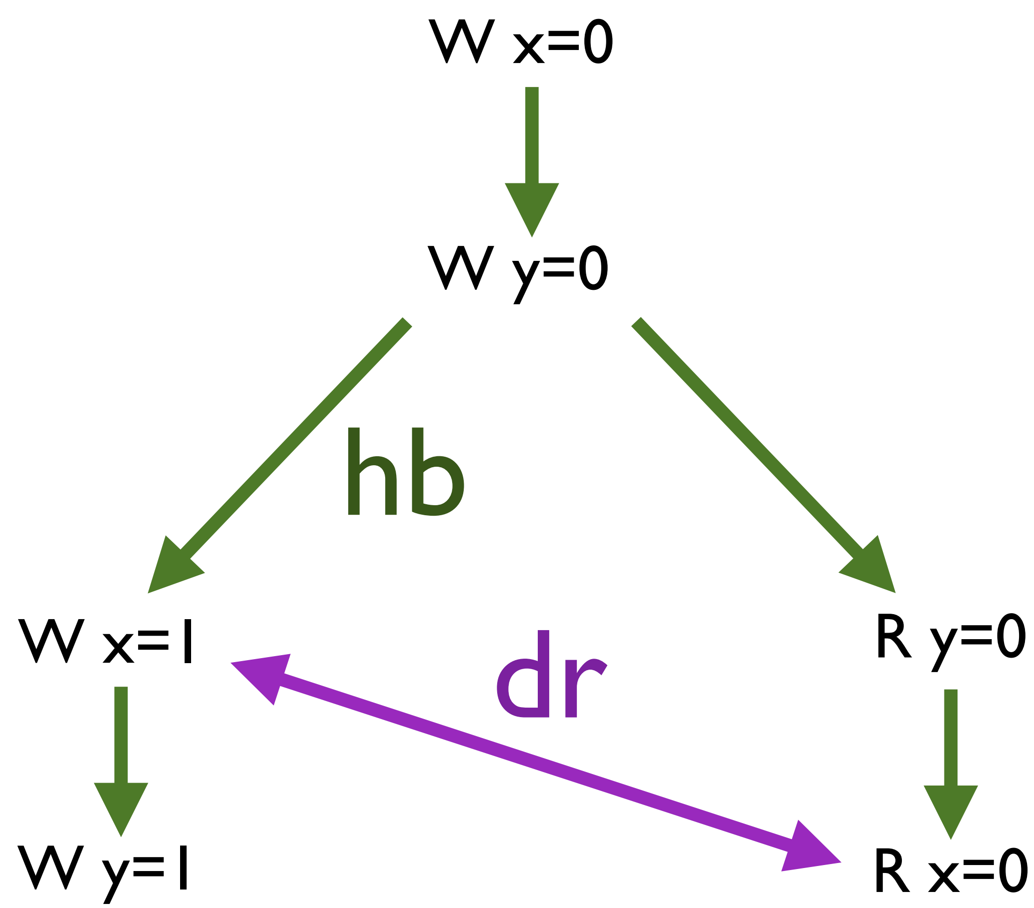


Wednesday, 17 July 13

Oh no! A data race

Racy programs have undefined behaviour.



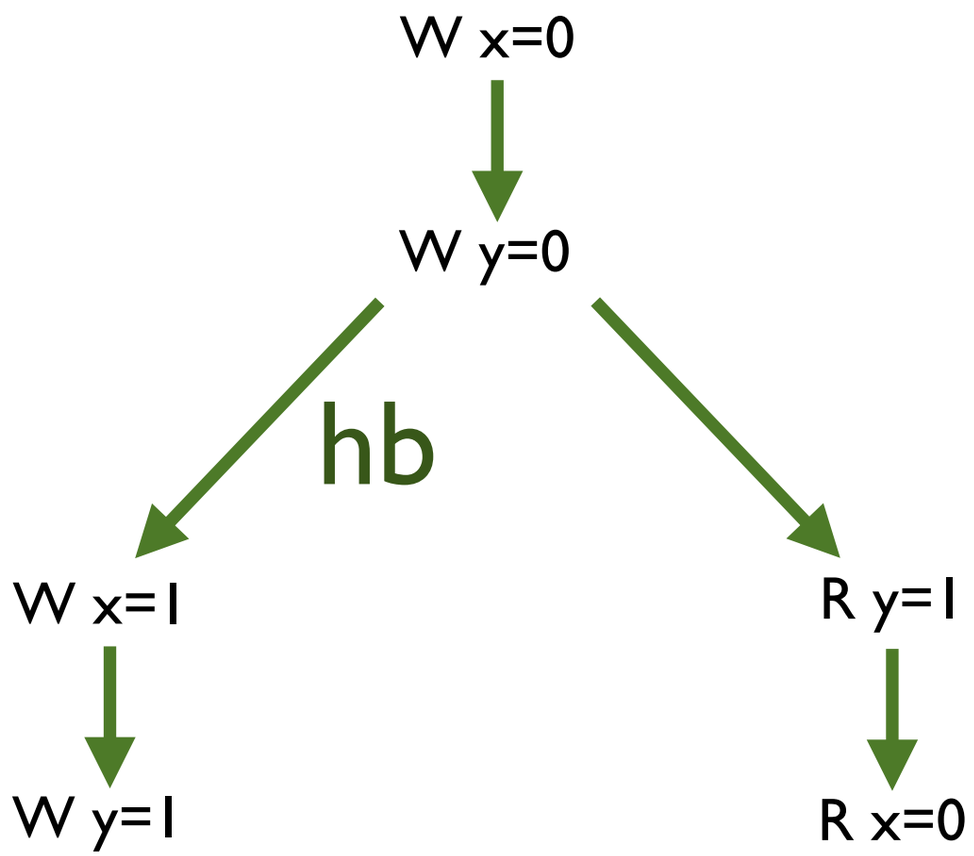


The programmer is required to avoid races

Atomic accesses

Use atomic accesses to write racy code.

```
int r;
atomic int x=0;
atomic int y=0;
storeRLX(&x,1); || r=loadRLX(&y);
storeRLX(&y,1); || r=loadRLX(&x);
```

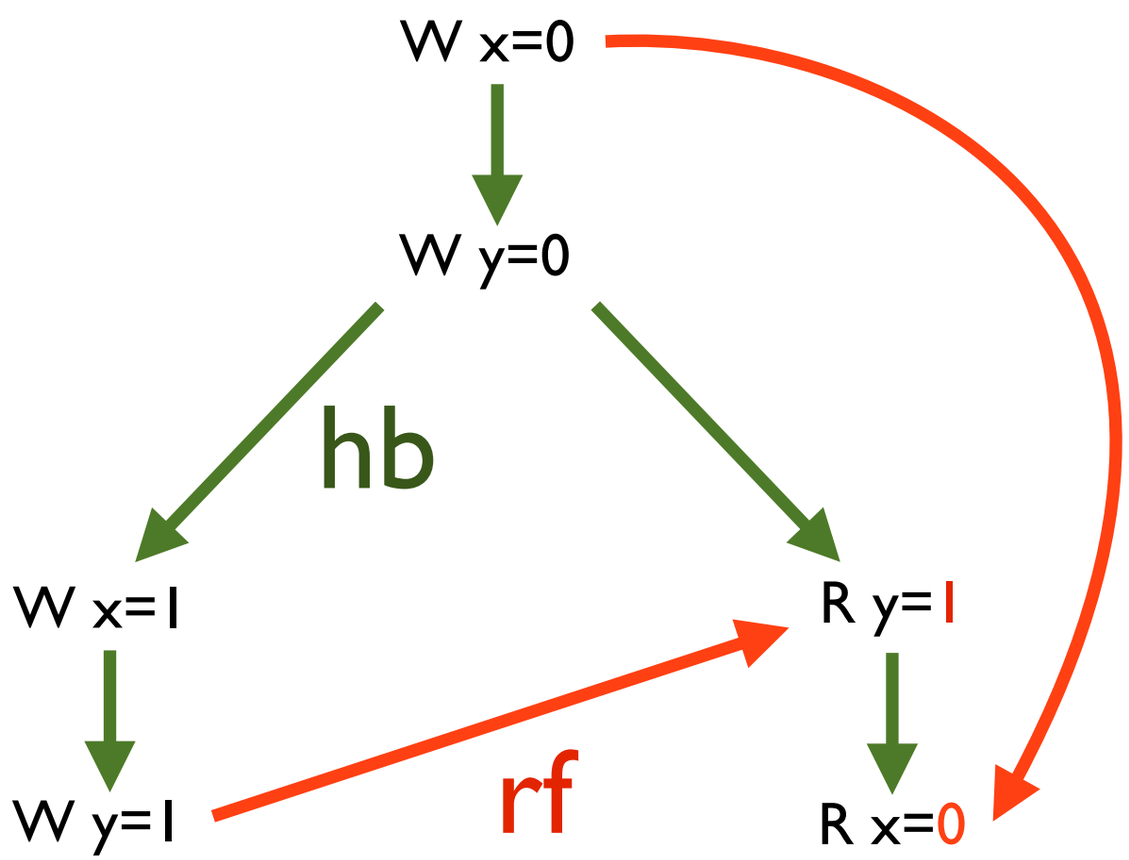


Atomic accesses do not race.

Atomic accesses

Beware! Not sequentially consistent.

```
int r;
atomic int x=0;
atomic int y=0;
storeRLX(&x,1); || r=loadRLX(&y);
storeRLX(&y,1); || r=loadRLX(&x);
```



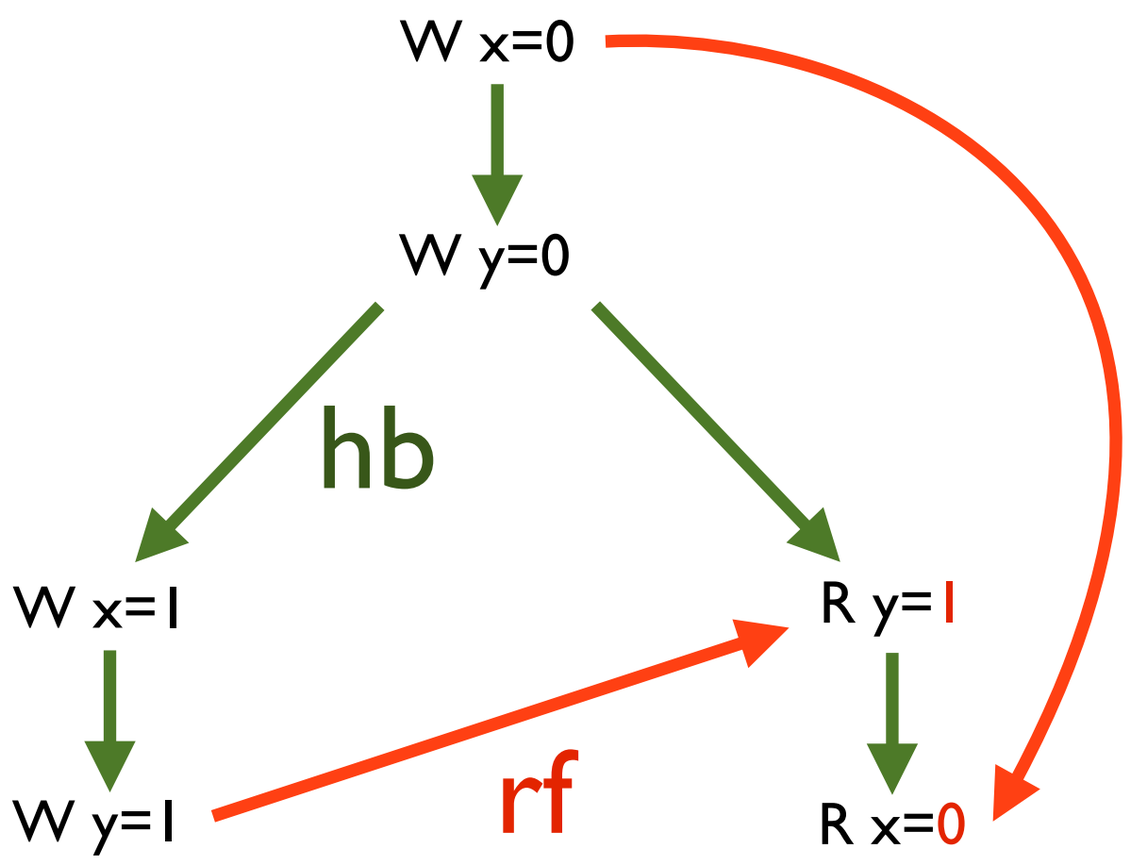
Atomic accesses

Beware! Not sequentially consistent.

```
int r;
atomic int x=0;
atomic int y=0;
```

$store_{RLX}$(&x,1);	r=$load_{RLX}$(&y);
$store_{RLX}$(&y,1);	r=$load_{RLX}$(&x);

Allowed by Power, ARM, or compiler optimisations

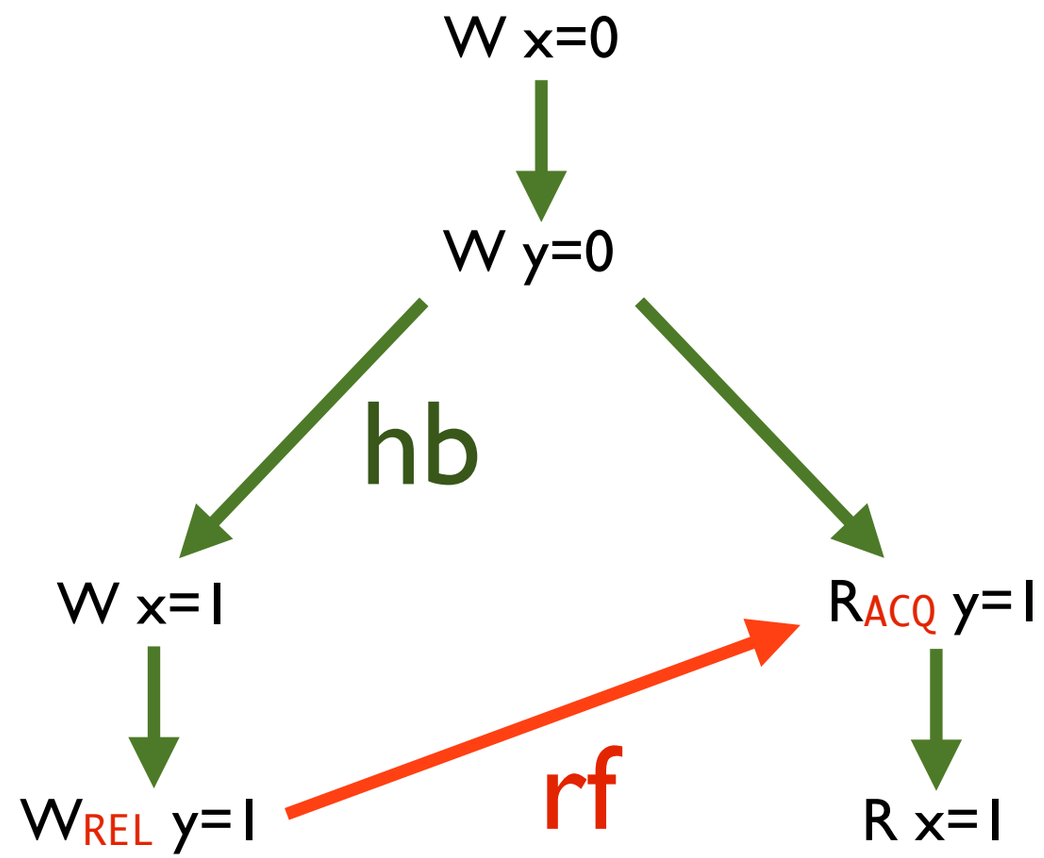


Message passing

Use memory order annotations to synchronise.

```
int r;
atomic int x=0;
atomic int y=0;
```

$store_{RLX}$(&x,1);	r=$load_{ACQ}$(&y);
$store_{REL}$(&y,1);	r=$load_{RLX}$(&x);

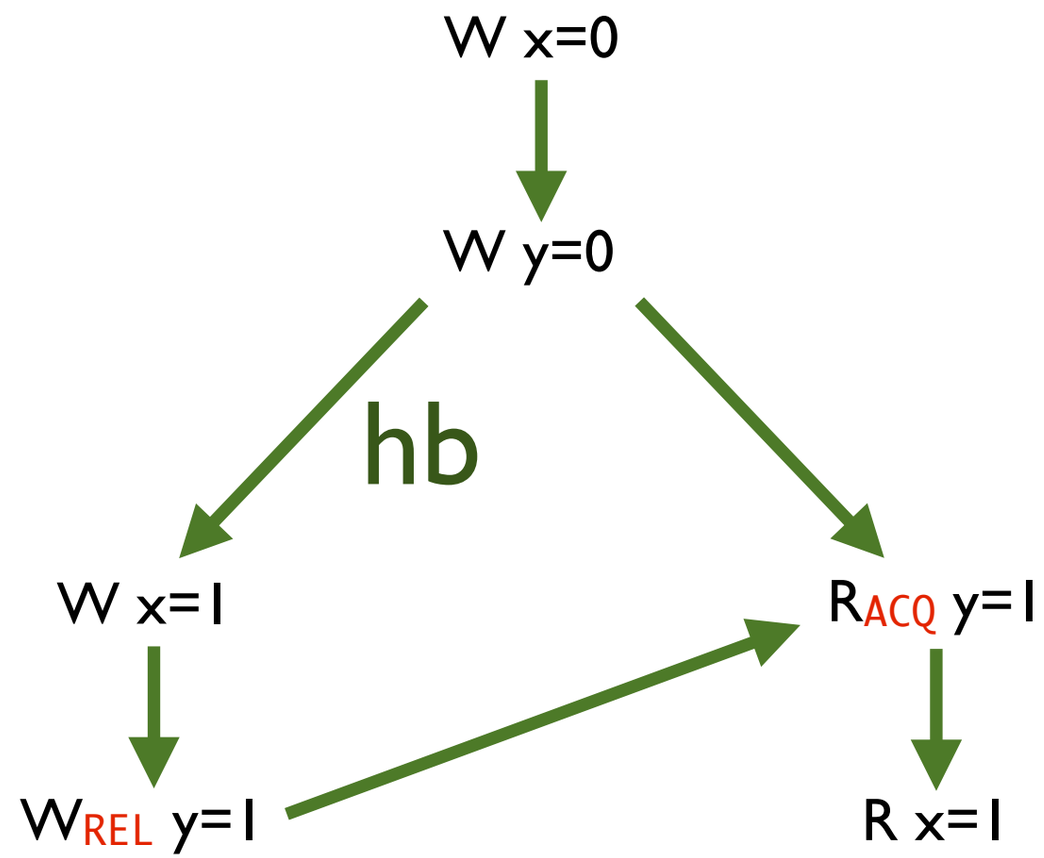


relaxed, release/acquire, seq_cst

Message passing

Use memory order annotations to synchronise.

```
int r;
atomic int x=0;
atomic int y=0;
storeRLX(&x,1);  ||  r=loadACQ(&y);
storeREL(&y,1);  ||  r=loadRLX(&x);
```



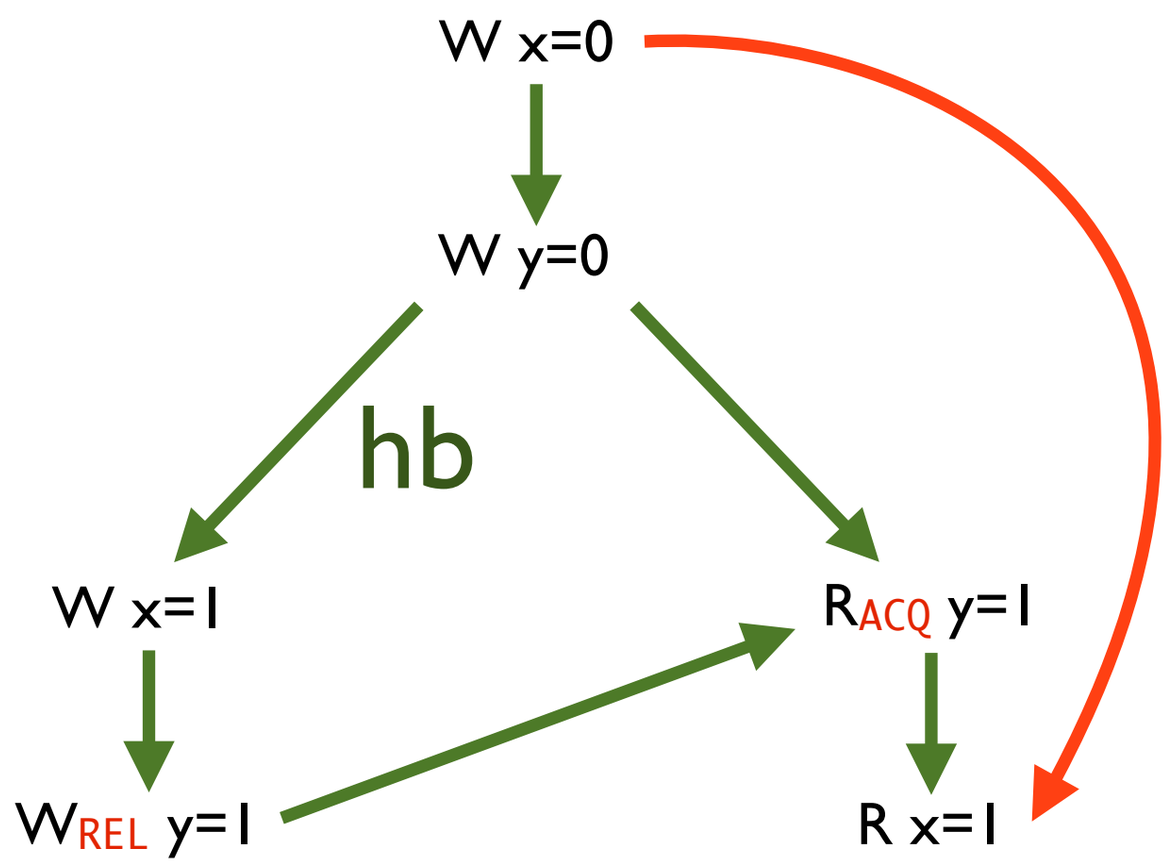
rf from REL to ACQ becomes an hb edge

Message passing

Use memory order annotations to synchronise.

```
int r;
atomic int x=0;
atomic int y=0;
```

```
store_RLX(&x,1);  ||  r=load_ACQ(&y);
store_REL(&y,1);  ||  r=load_RLX(&x);
```



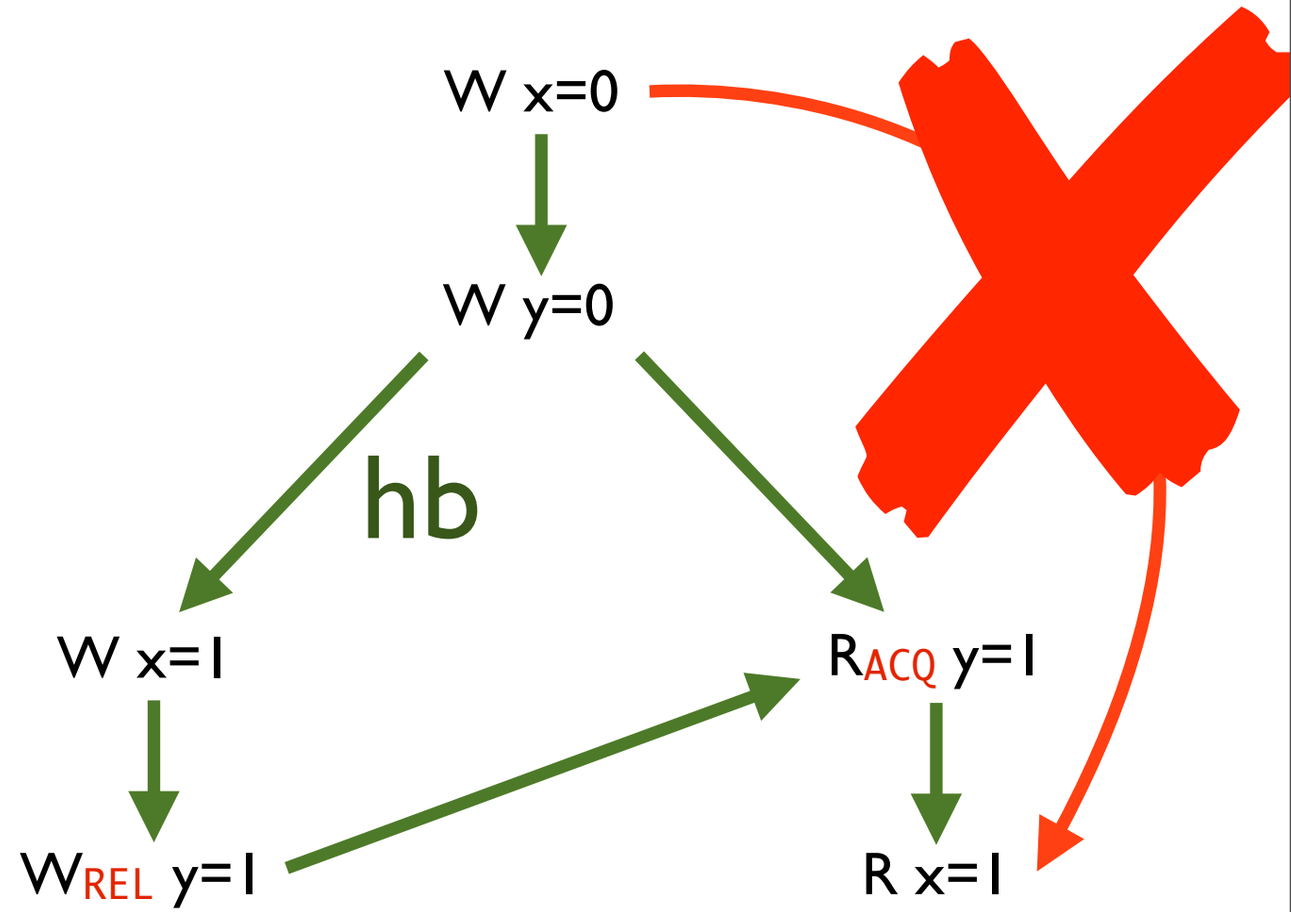
Cannot read a stale write in hb

Message passing

Use memory order annotations to synchronise.

```
int r;
atomic int x=0;
atomic int y=0;
```

```
storeRLX(&x,1);  ||  r=loadACQ(&y);
storeREL(&y,1);  ||  r=loadRLX(&x);
```

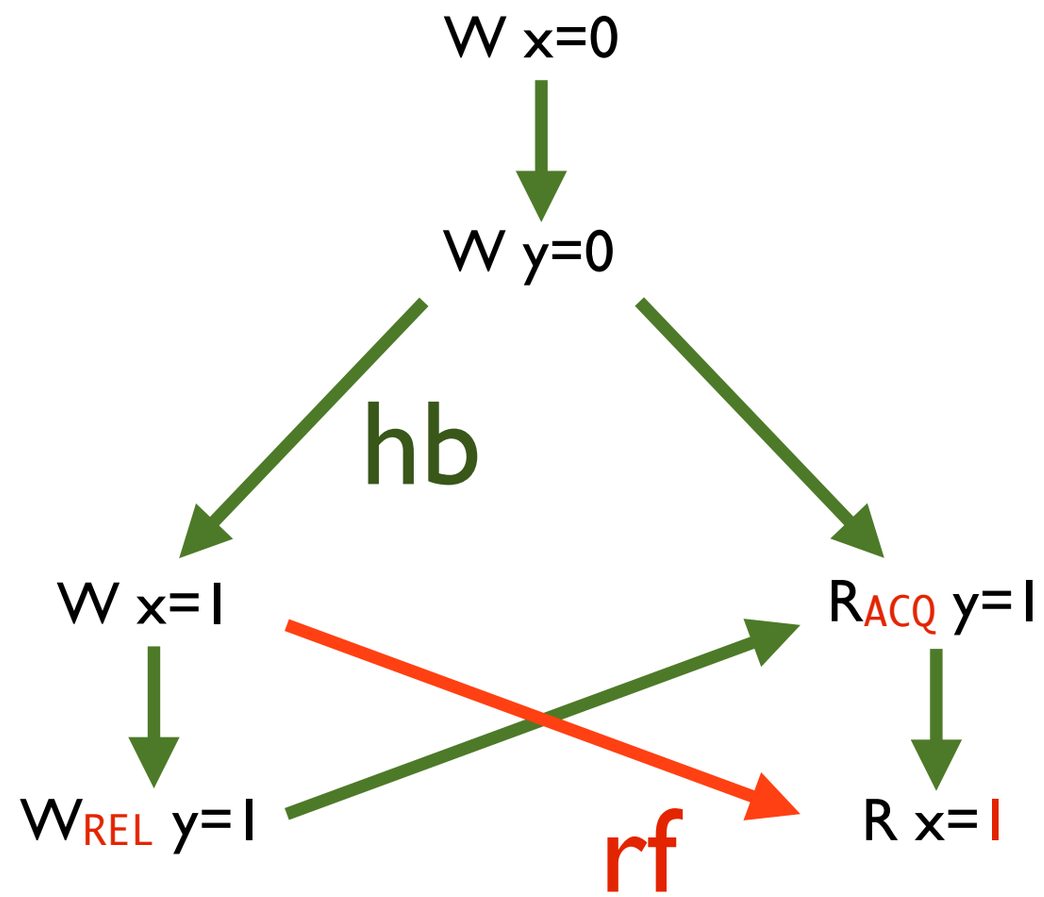


Cannot read a stale write in hb

Message passing

Use memory order annotations to synchronise.

```
int r;
atomic int x=0;
atomic int y=0;
storeRLX(&x,1);  ||  r=loadACQ(&y);
storeREL(&y,1);  ||  r=loadRLX(&x);
```



Coherence

Modification order is a per-location total order over atomic writes

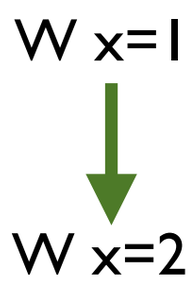
Modification order must agree with happens before

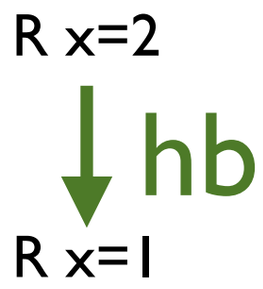
Coherence is defined in terms of this order

CoRR

Thread 0	Thread 1

x=1
x=2
	r1=x //reads 2
	r2=x //reads 1





Coherence

Modification order is a per-location total order over atomic writes

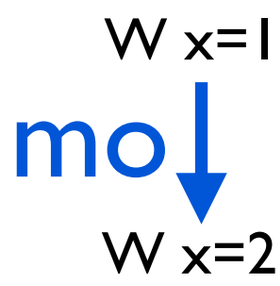
Modification order must agree with happens before

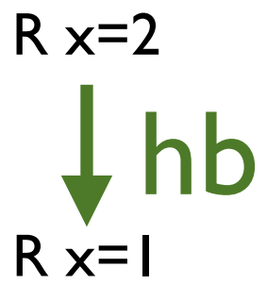
Coherence is defined in terms of this order

CoRR

Thread 0	Thread 1

x=1
x=2
	r1=x //reads 2
	r2=x //reads 1





Coherence

Modification order is a per-location total order over atomic writes

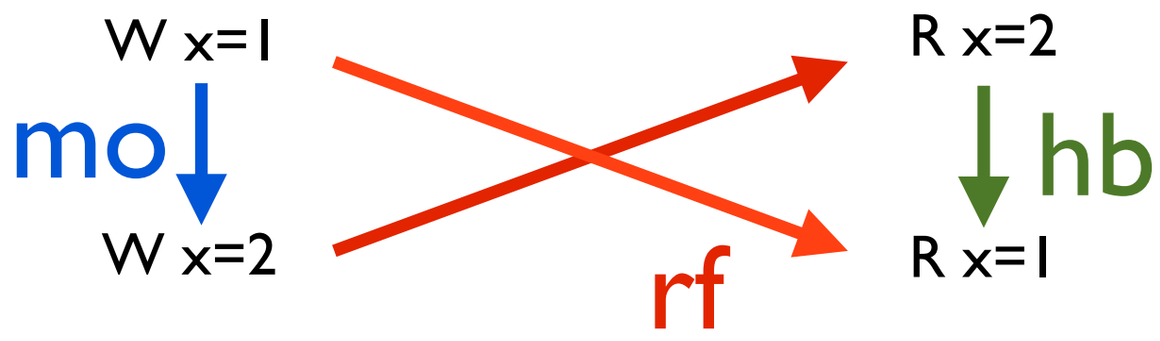
Modification order must agree with happens before

Coherence is defined in terms of this order

CoRR

Thread 0	Thread 1

x=1
x=2
	r1=x //reads 2
	r2=x //reads 1



Coherence

Modification order is a per-location total order over atomic writes

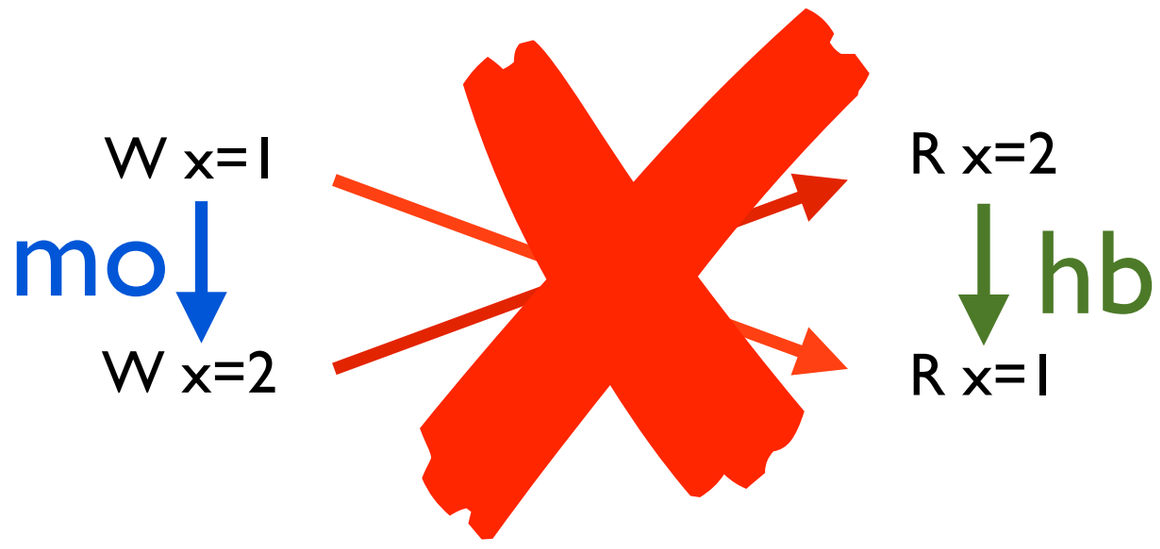
Modification order must agree with happens before

Coherence is defined in terms of this order

CoRR

Thread 0	Thread 1

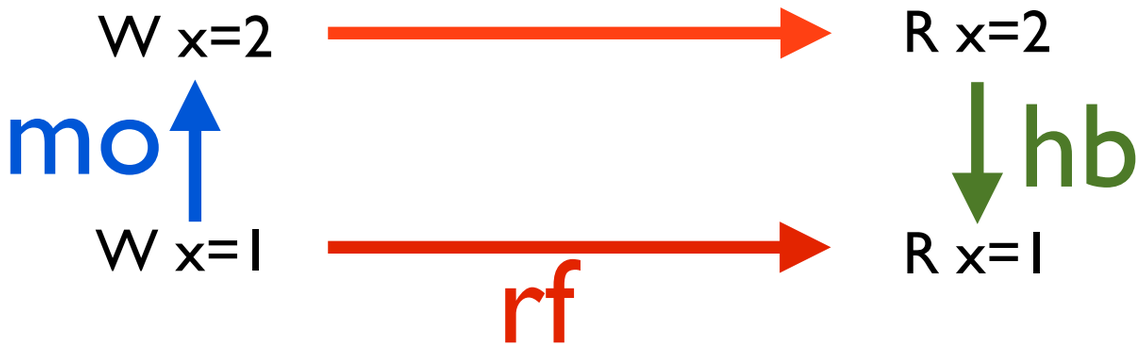
x=1
x=2
	r1=x //reads 2
	r2=x //reads 1



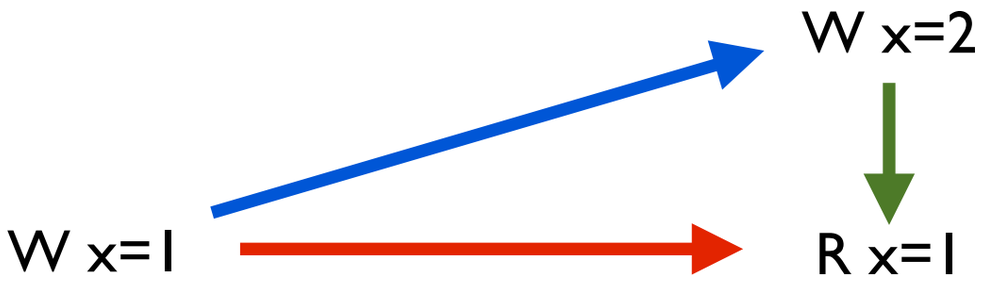
Coherence

All forbidden in C/C++11!

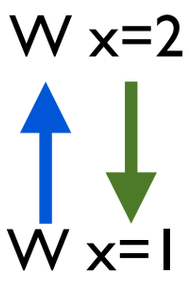
CoRR



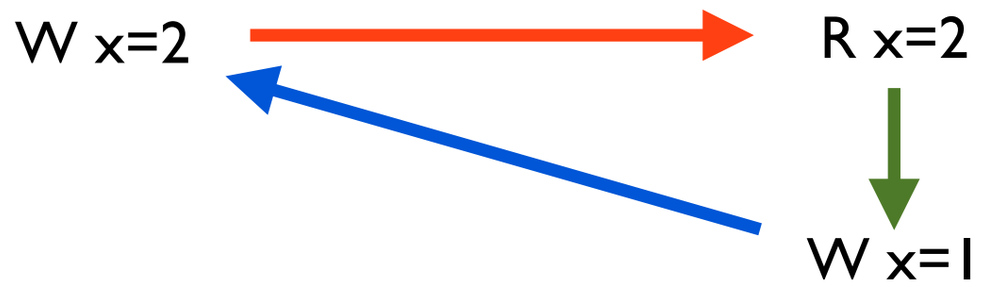
CoWR



CoWW



CoRW



Atomicity

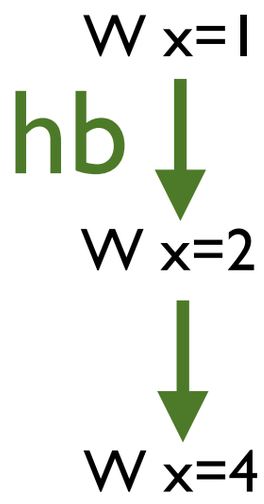
Modification order also defines atomicity of CAS-like features

A successful CAS produces a read-modify-write (RMW) access
(A failed CAS is an atomic read. In Fortran?)

A RMW reads the maximal preceding write in mo

RMW

Thread 0	Thread 1
x=1 x=2 x=4	CAS (&x,2,3)



RMW x=2,3

Atomicity

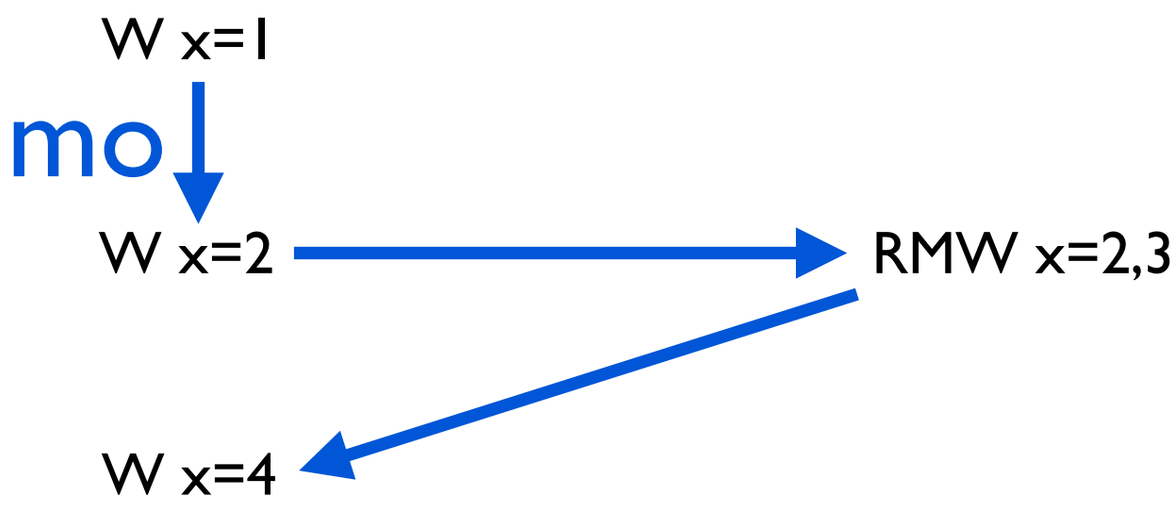
Modification order also defines atomicity of CAS-like features

A successful CAS produces a read-modify-write (RMW) access
(A failed CAS is an atomic read. In Fortran?)

A RMW reads the maximal preceding write in mo

RMW

Thread 0	Thread 1
x=1 x=2 x=4	CAS (&x,2,3)



Atomicity

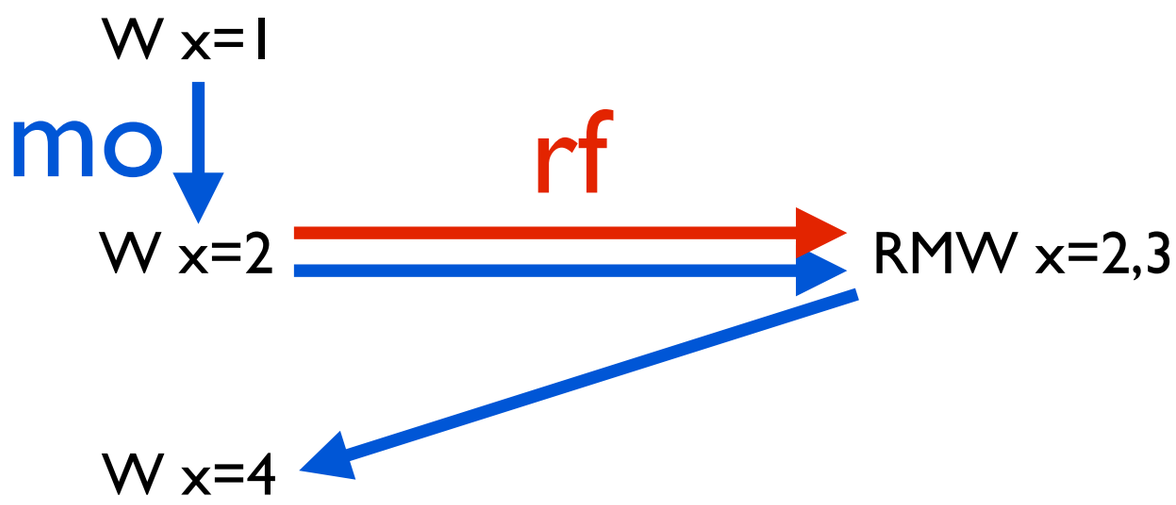
Modification order also defines atomicity of CAS-like features

A successful CAS produces a read-modify-write (RMW) access
(A failed CAS is an atomic read. In Fortran?)

A RMW reads the maximal preceding write in mo

RMW

Thread 0	Thread 1
x=1 x=2 x=4	CAS (&x,2,3)



Operational vs Axiomatic models

Operational models (x86, Power/ARM):

- Can be similar to micro-architectural implementation
- Provide a decent local intuition
- Good for simulation
- Malleable
- Disguise incidental choices
- Awkward, ad-hoc structure

Axiomatic models (C/C++11):

- Mathematically simpler
- Choices are made explicitly
- Simulation is harder
- Intuition is not local, and no machine state
- Out-of-thin-air values an open problem

Operational vs Axiomatic models

Operational models (x86, Power/ARM):

- Can be similar to micro-architectural implementation
- Provide a decent local intuition
- Good for simulation
- Malleable
- Disguise incidental choices
- Awkward, ad-hoc structure

Axiomatic models (C/C++11):

- Mathematically simpler
- Choices are made explicitly
- Simulation is harder
- Intuition is not local, and no machine state
- Out-of-thin-air values an open problem

In C/C++11 out-of-thin-air is ill-defined

C11 abstracts relaxed behaviour introduced by processors and compiler optimisations

Out-of-thin-air values should be forbidden

At the moment this is poorly defined

Example 1: simple thin air

```
                atomic int x=0, y=0;
r1=loadRLX(&x);      || r2=loadRLX(&y);
storeRLX(&y,r1);     || storeRLX(&x,r2);
```

Can r1 and r2 end up with the value 42?

The intent is to forbid this sort of thing.

Example 2: load buffering (LB)

...but Power and ARM allow load buffering,
a similar behaviour without the dependencies,

Power, ARM and C11 allow 1,1 below:
(observable on ARM)

```
                atomic int x=0, y=0;
r1 = load_RLX(&x);     || r2 = load_RLX(&y);
store_RLX(&y,1);       || store_RLX(&x,1);
```

Example 3: satisfaction cycles

```
atomic int x=0, y=0;
if (load_RLX(&x)==1)  ||  if (load_RLX(&y)==1)
  store_RLX(&y,1);    ||    store_RLX(&x,1);
```

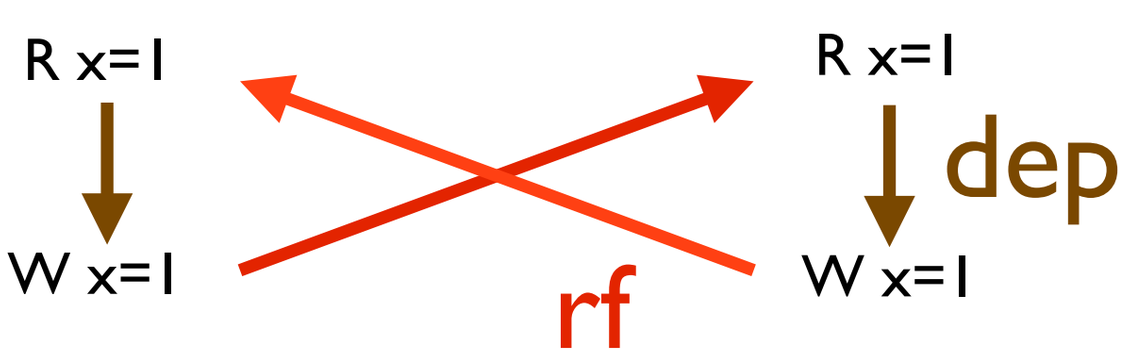
The C11 model ought to forbid this -- it does not

C11:"implementations should not allow such behaviour"

Forbid dependency cycles?

Define a new edge: dependency

Forbid cycles in dependency union rf



But optimisations remove some dependencies

The spec is then choosing which optimisations to allow

Example 4: dependency tracking?

```
                atomic int x=0, y=0;
r1=load_RLX(&x);          ||   r3=load_RLX(&y);
f(&r1,&r2);               ||   f(&r3,&r4);
store_RLX(&y,r2);         ||   store_RLX(&x,r4);
```

Defining and tracking dependencies might be hard!

Optimisations in non-atomic code could break dependencies.

Would separate compilation provide a dependency summary?

Out-of-thin-air restriction: design space

Just allow this behaviour?

Disallow LB with a stronger model?

If neither, then defining dependency is key

Must allow sequential compiler optimisations in C/C++

Out-of-thin-air restriction: design space

Just allow this behaviour?

Disallow LB with a stronger model?

If neither, then defining dependency is key

Must allow sequential compiler optimisations in C/C++

We don't know how to fix this in C/C++11 right now!

Working towards a model for Fortran

What do implementations allow?

Which programming idioms to allow?

So... what do implementations allow?

Which litmus tests can be observed for atomics?

SB

Thread 0	Thread 1
x=1 r1=y //reads 0	y=1 r2=x //reads 0

MP

Thread 0	Thread 1

x=1
y=1
	r1=y //reads 1
	r2=x //reads 0

LB

Thread 0	Thread 1

r1=x //reads 1
y=1
	r2=y //reads 1
	x=1

IRIW

Thread 0	Thread 1	Thread 2	Thread 3

y=1	x=1	r1=x //reads 1
		r2=y //reads 0
			r3=y //reads 1
			r4=x //reads 0

There are many more!

With relaxed CAS in C/C++11

SB, MP and LB look the same, but are allowed:

SB

Thread 0	Thread 1
CAS(&x,1,2) r1= CAS(&y,0,1) //reads 0	CAS(&y,1,2) r2= CAS(&x,0,1) //reads 0

MP

Thread 0	Thread 1

CAS(&x,1,2)
CAS(&y,0,1)
	r1= CAS(&y,1,2) //reads 1
	r2= CAS(&x,0,1) //reads 0

LB

Thread 0	Thread 1

r1= CAS(&x,1,2) //reads 1
CAS(&y,0,1)
	r2= CAS(&y,1,2) //reads 1
	CAS(&x,0,1)

13-269: A model for Fortran?

Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

13-269: A model for Fortran?

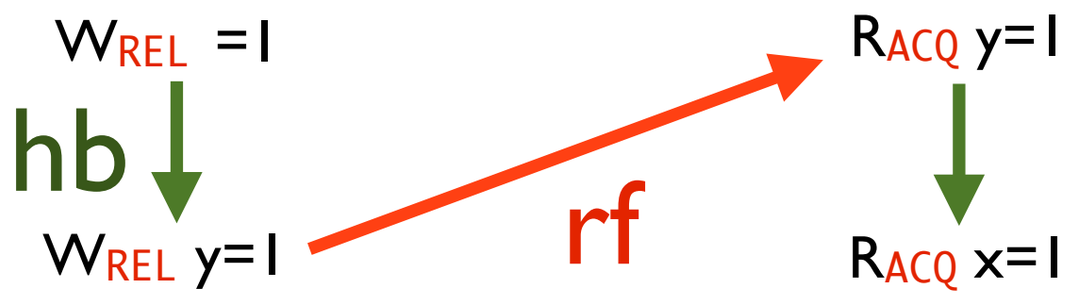
Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

This is "causal consistency"

All writes are releases, all reads acquires.
All reading produces happens-before.



13-269: A model for Fortran?

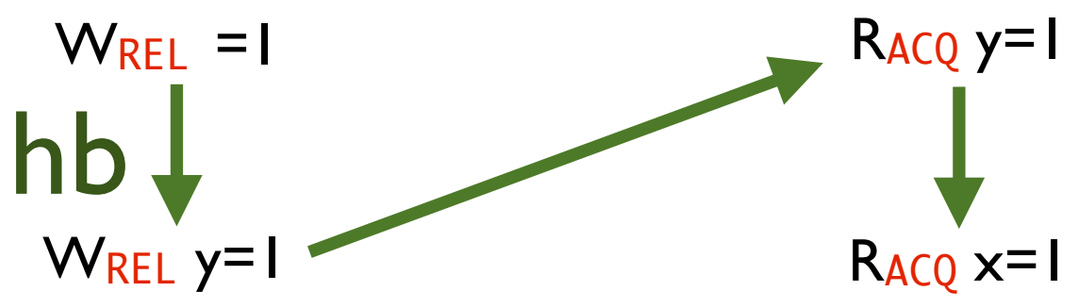
Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

This is "causal consistency"

All writes are releases, all reads acquires.
All reading produces happens-before.



13-269: A model for Fortran?

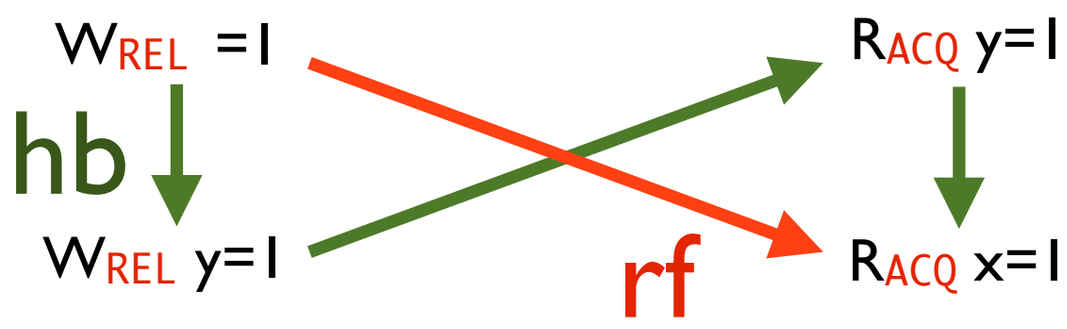
Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

This is "causal consistency"

All writes are releases, all reads acquires.
All reading produces happens-before.



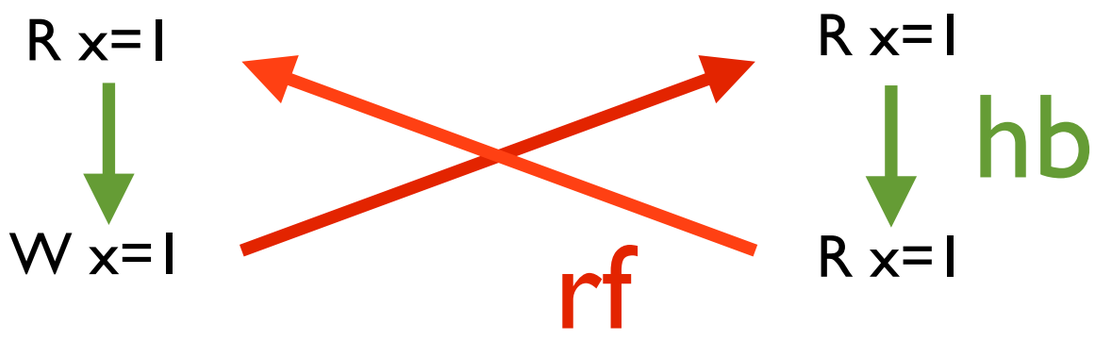
13-269: A model for Fortran?

Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

Cycles are forbidden in happens-before



This gets rid of load buffering

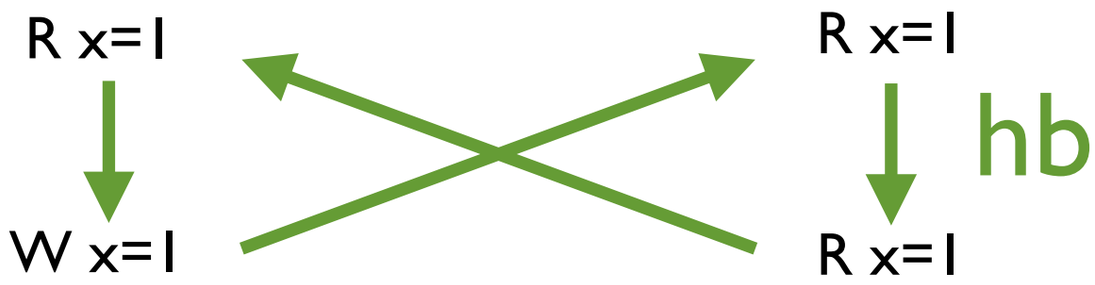
13-269: A model for Fortran?

Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

Cycles are forbidden in happens-before



This gets rid of load buffering

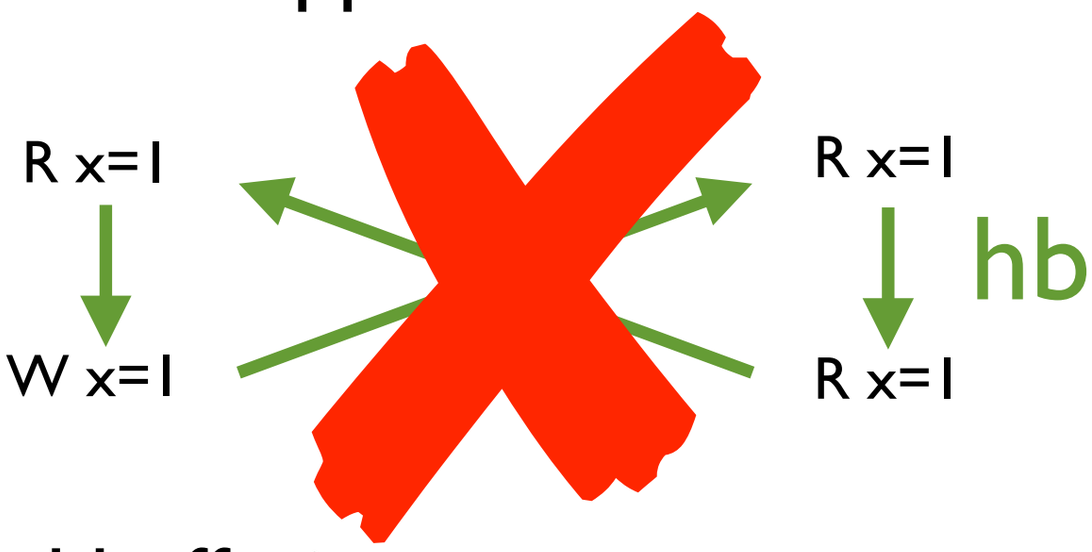
13-269: A model for Fortran?

Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

Cycles are forbidden in happens-before



This gets rid of load buffering

13-269: A model for Fortran?

Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

SB

Thread 0	
CAS(&x,1,2) r1= CAS(&y,0,1) //reads	&y,1,2) AS(&x,0,1) //reads 0

MP

Thread 0	Thread 1

CAS(&x,1,2)
CAS(&y,0,1)
	r1= C 1,2) //reads 1
	r2= x,0,1) //reads 0

MP

Thread 0	read 1

r1= CAS(&x,1,2) //r
CAS(&y,0,1)
	AS(&y,1,2) //reads 1
	0,1)

Strange CAS behaviour goes away

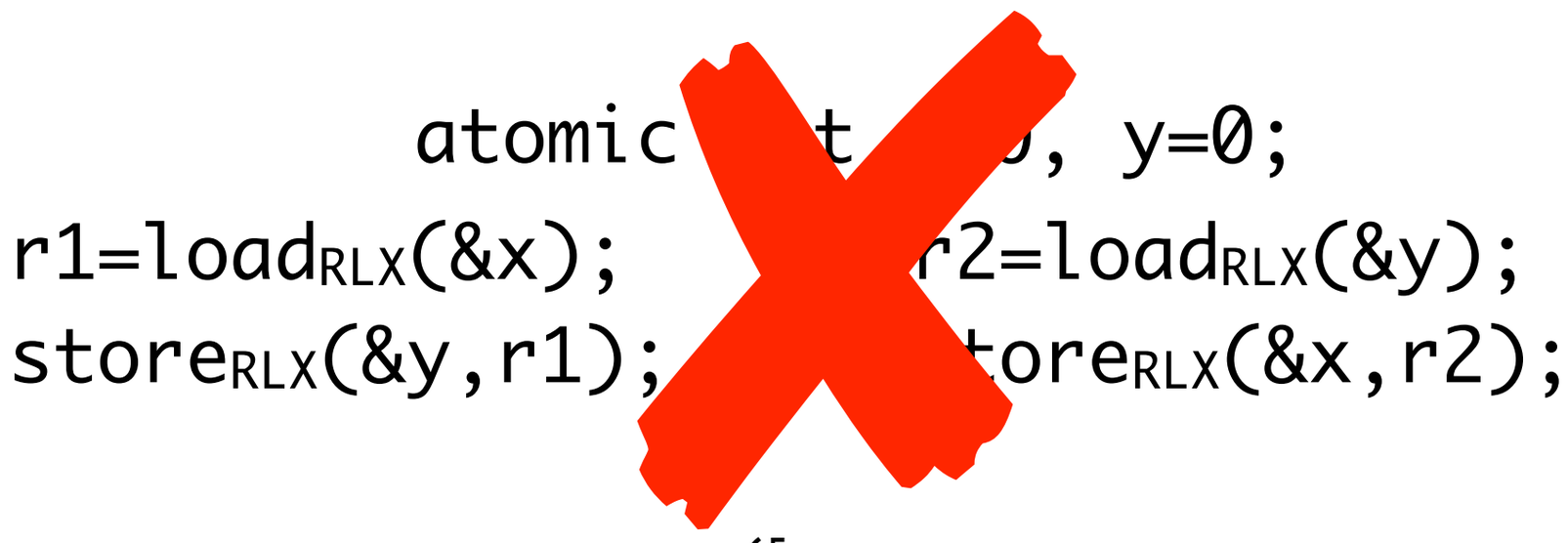
13-269: A model for Fortran?

Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

Thin air behaviour goes away



13-269: A model for Fortran?

Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

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

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

13-269: A model for Fortran?

Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

This comes for free on x86.

On Power/ARM this costs something:

Each atomic write needs an lwsync
Reads need an isync, and false dependencies
RMW's need both

13-269: A model for Fortran?

Underlying targets use message passing

Optimisation very important

Use C/C++ without relaxed?

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

It would forbid breaking-out of synchronisation:

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

Which litmus tests can be observed?

Which programming idioms should be supported?

Can the target machine be abstracted?

Which compiler optimisations should be allowed?

What behaviours do intended implementations have?