Hardware models:

inventing a usable abstraction for Power/ARM

Hardware models:

inventing a usable abstraction for Power/ARM

Disclaimer:

1. ARM MM is analogous to Power MM… all this is your next phone!

2. The model I will present is (as far as we know) accurate for ARM if barriers weaker than DMB are not used.

Power: much more relaxed than x86

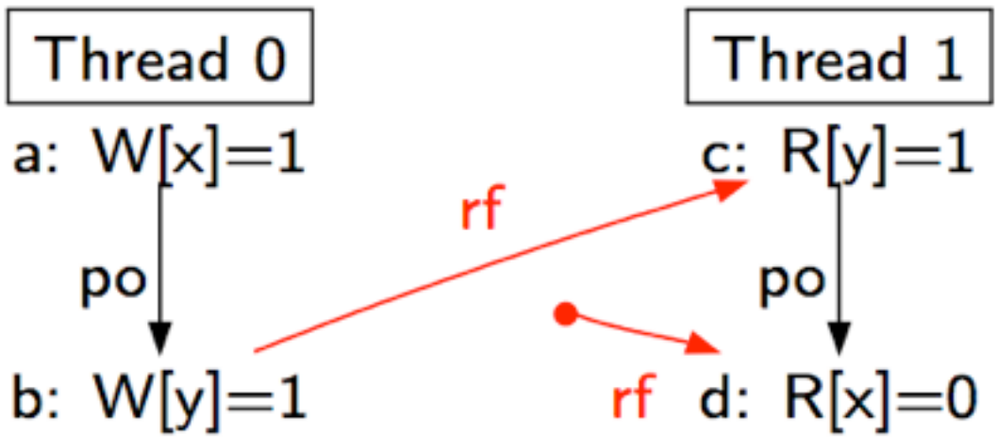
Thread 0	Thread 1
`x = 1`	`while (y==0) {};`
`y = 1`	`r = x`

Observable behaviour: `r = 0`

Power: much more relaxed than x86

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

Power: much more relaxed than x86

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 perfomed in opposite order

   load reorder buffers / speculation

3. propagation of writes ignores order in which they are presented

   interconnects partitioned by address (cache lines)

Power: much more relaxed than x86

Three pos

Power has all three!

1. the two writes are performed in opposite order

   reordering store buffers

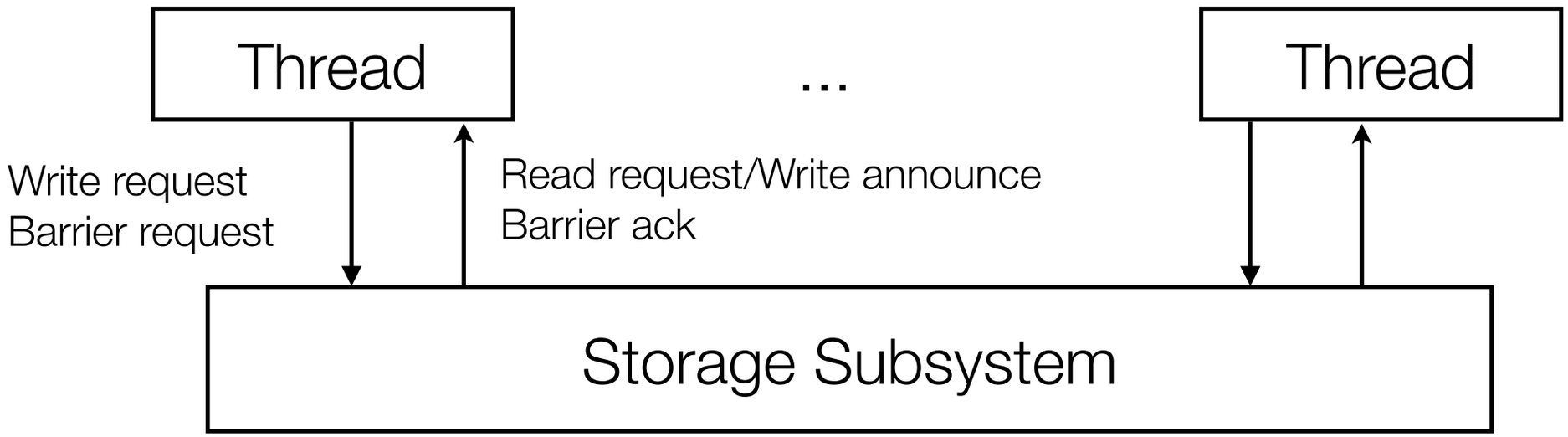
2. the two reads are perfomed in opposite order

   load reorder buffers / speculation

3. propagation of writes ignores order in which they are presented

   interconnects partitioned by address (cache lines)

The model overall structure



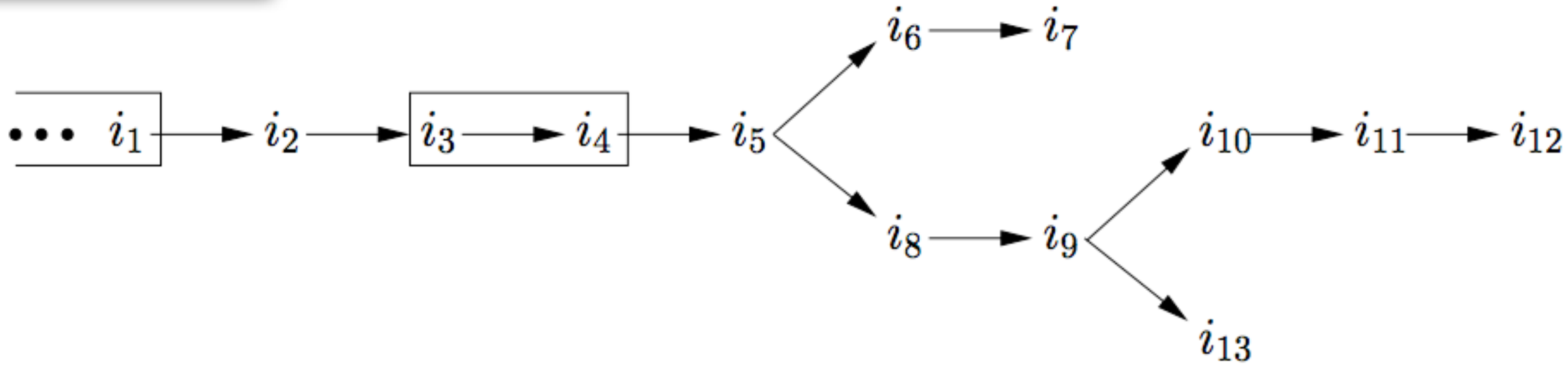
Some aspects are thread-only, some storage-only, some both.

Threads and storage subsystem are abstract state machines.

Speculative execution in Threads; topology-independent Storage.

Much more complicated than x86-TSO.
Are you ready?

Thread



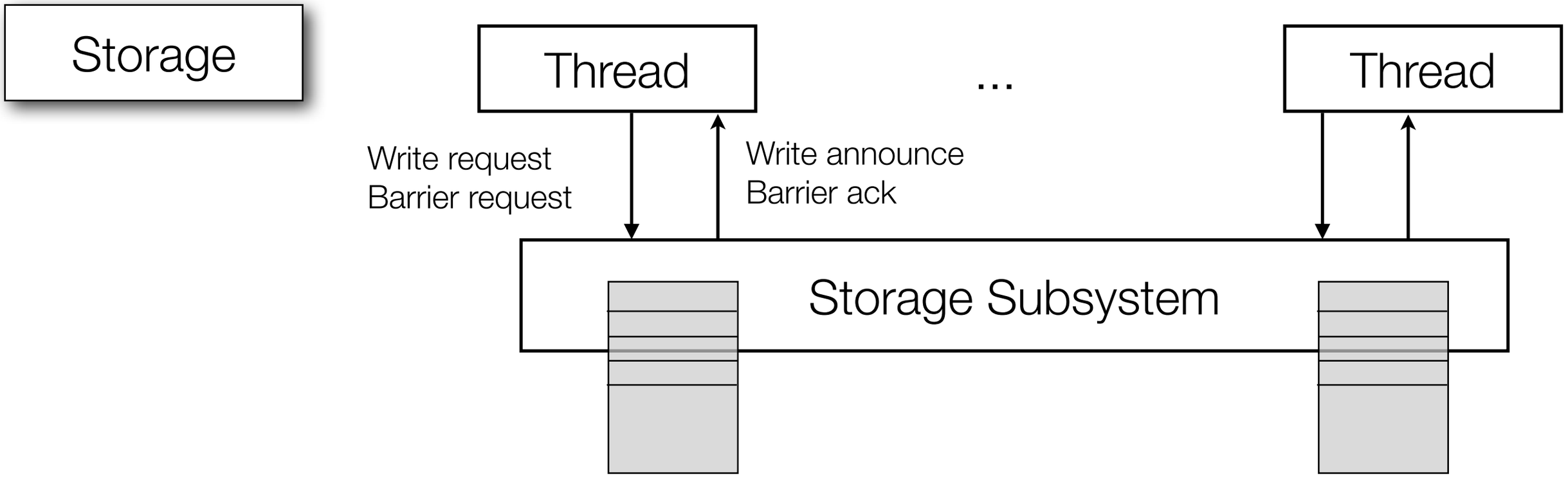
Each thread loads its code, instructions instances are initially marked *in-flight.*

In-flight instructions can be *committed,* not necessarily in program order.

When a branch is committed, the un-taken alternatives are discarded.

Instructions that follow an uncommitted branch cannot be committed.

In-flight instructions can be processed even before being committed (e.g. to speculate reads from memory, perform computation, ...).



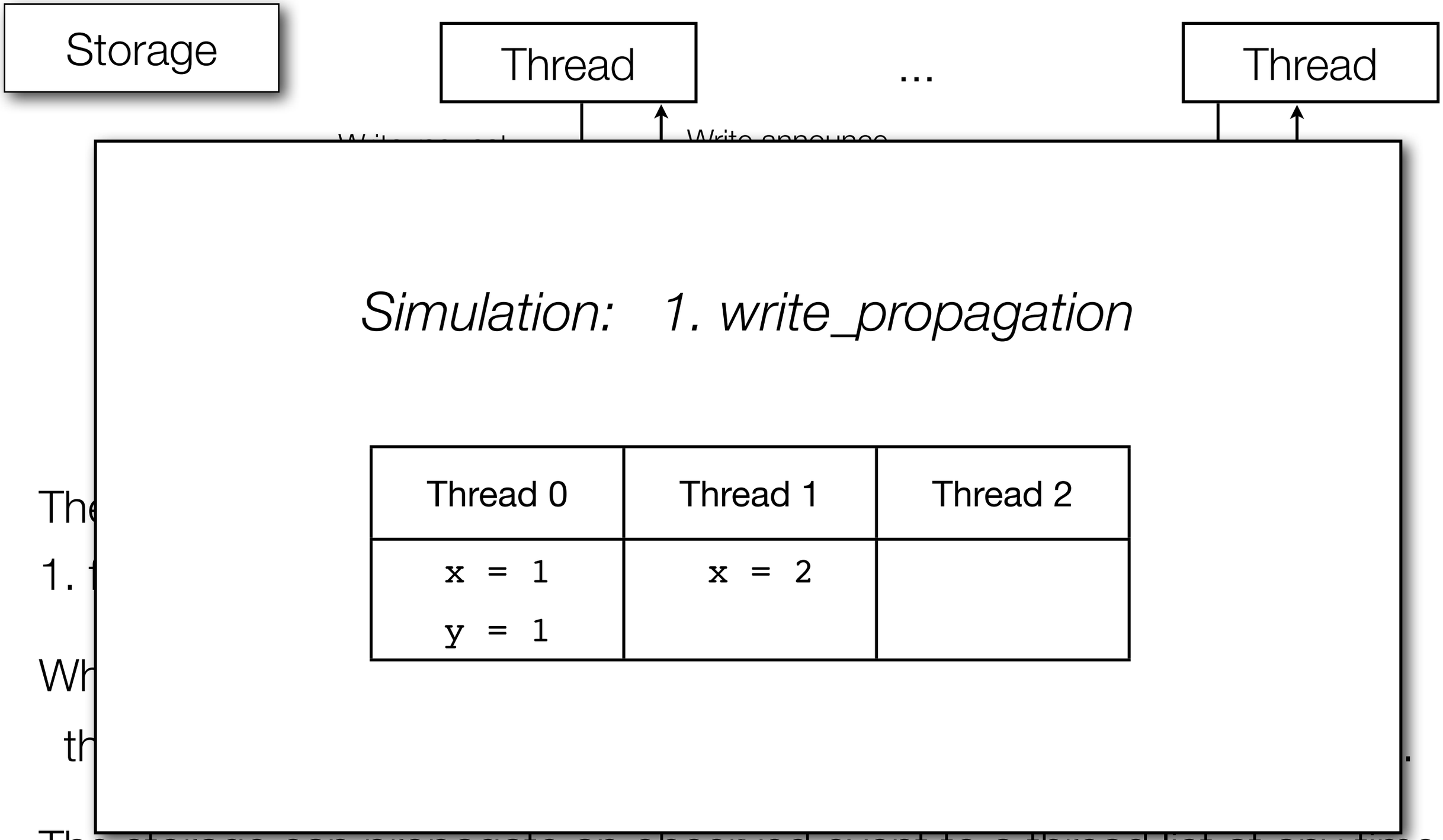
The storage keeps (among other things):

1. for each thread, a list of the events propagated to the thread.

When receiving a write request, the storage adds the write event to the list of the events propagated to the thread who issued the request.

The storage can propagate an observed event to a thread list at any time *(unless barriers / coherence /... conditions)*.

Threads can commit writes at any time *(unless dependency / synch / pending /… conditions).*



The

1. f

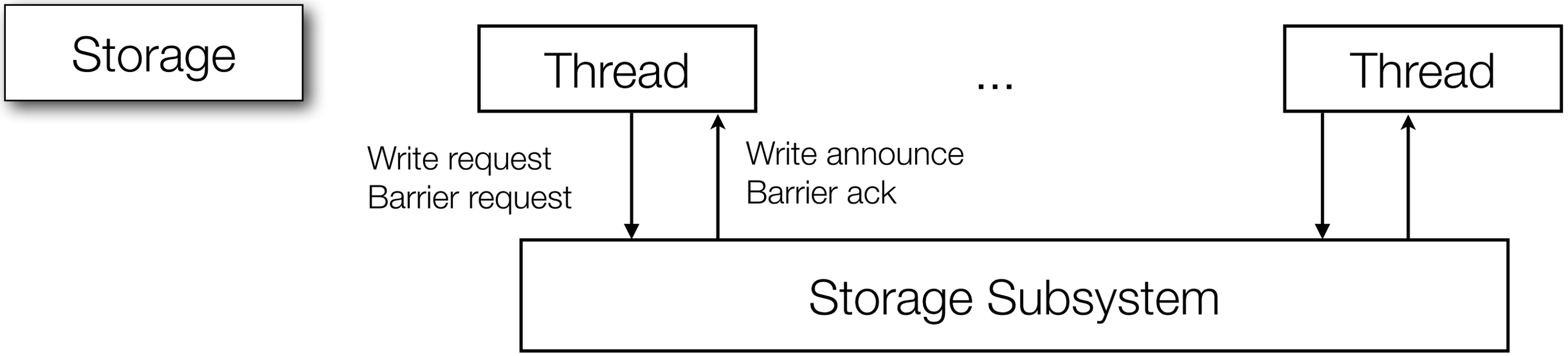
Wh

th

.

The storage can propagate an observed event to a thread list at any time *(unless barriers / coherence /... conditions)*.

Threads can commit writes at any time
(unless dependency / synch / pending /… conditions).



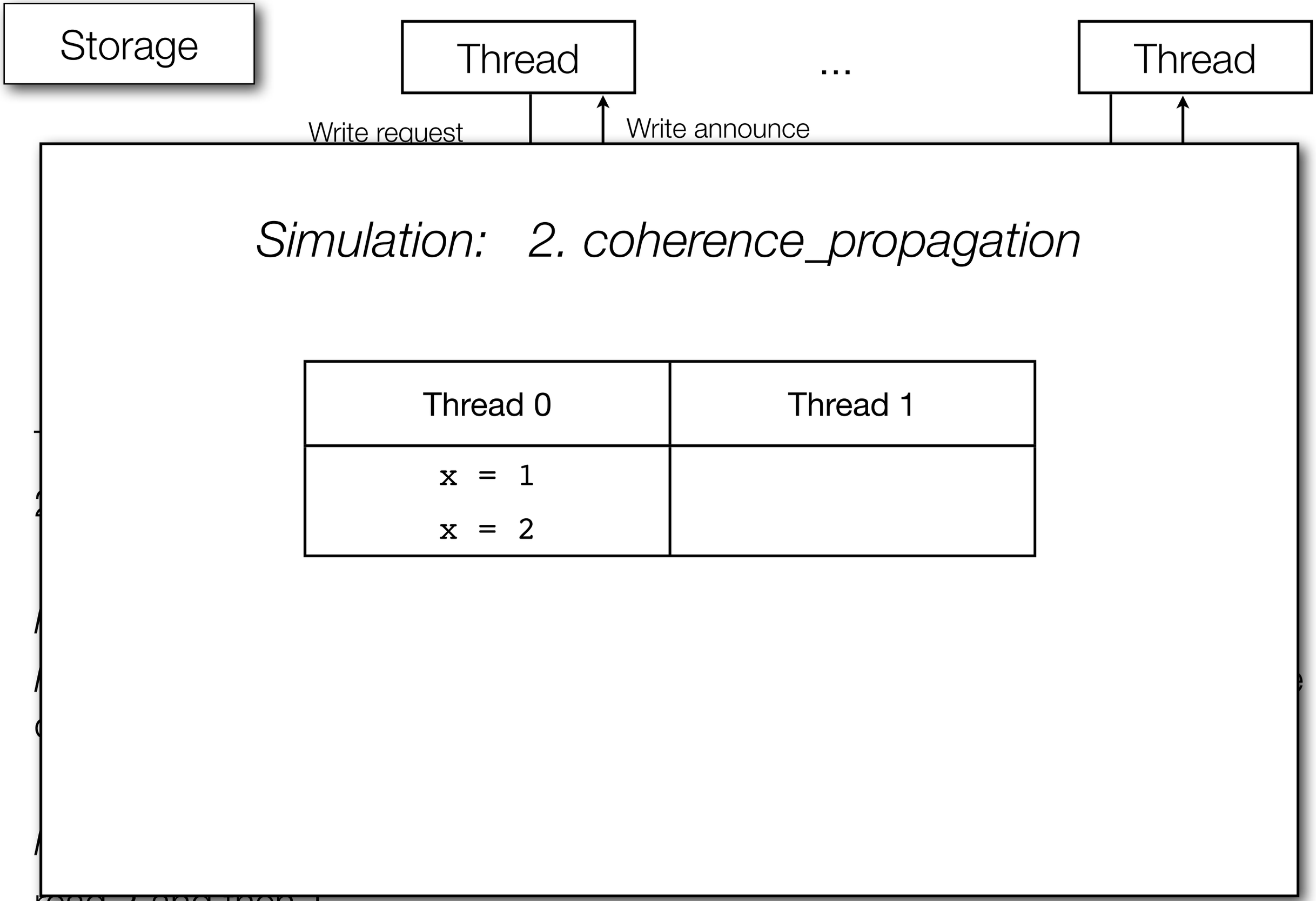
The storage keeps: ...

2. for each location, a partial order of *coherence commitments*

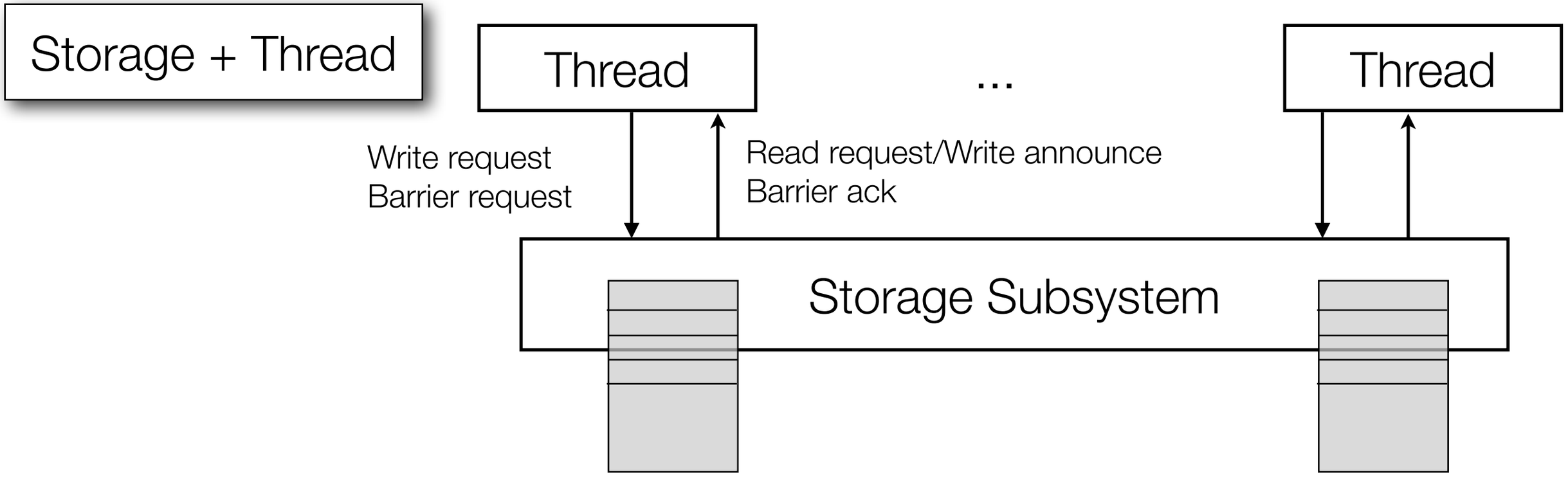
Idea 1: at the end of the execution, writes to each location are totally ordered.

Idea 2: during computation, reads and propagation of writes must respect the coherence order (*reduce non-determism of previous rules).*

Intuition: if a thread executes `x=1` and then `x=2`, another thread cannot first read `2` and then `1`.



read 2 and then 1.



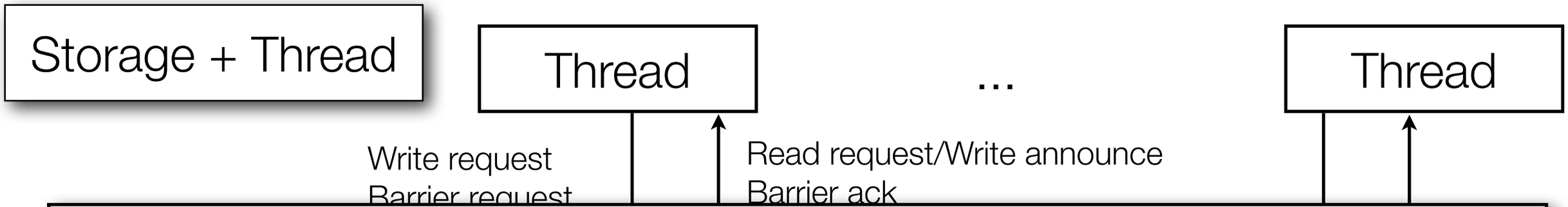
Threads can issue *read-requests* at any time *(unless dependency / synch / ...).*

Issuing a read-request and committing a read are **different actions**.

Storage can accept a read-request by a thread at any time, and reply with the **most recent write** to the same address **that has been propagated** to the thread.

Remark: receiving a write-announce is not enough to commit a read instruction.

Write-announces can be invalidated, and read-requests can be re-issued.



Simulation: 3. read_satisfy

Thread 0	Thread 1
`x = 1` `x = 2`	`r = x`

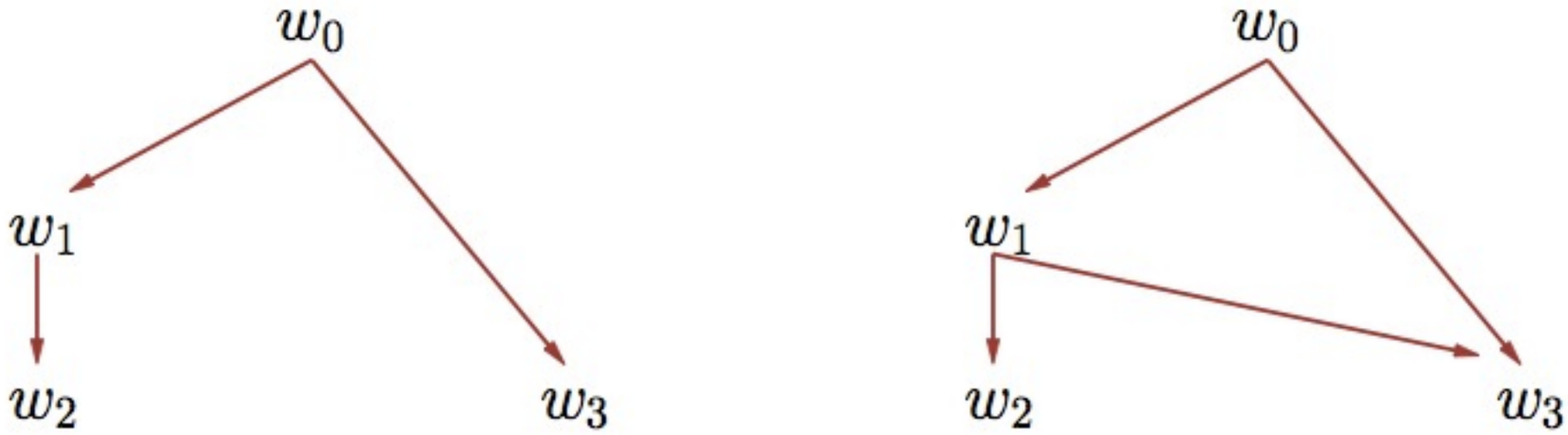
Simulation: 4. invalidate_read

Thread 0	Thread 1
`x = 1`	`r1 = x` `r2 = x`

Remarks: loads can be speculated; difference between read/write transitions

Coherence by Fiat

Suppose the storage subsystem has seen 4 writes to x:



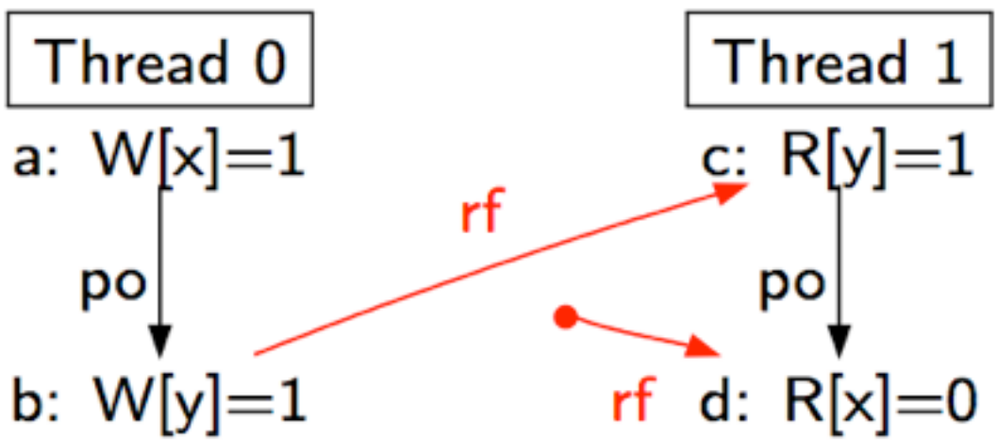
Suppose just [w1] has propagated to tid and then tid reads x.

- it cannot be sent w0, as w0 is coherence-before the w1 write that (because it is in the writes-propagated list) it might have read from;
- it could re-read from w1, leaving the coherence constraint unchanged;
- it could be sent w2, again leaving the coherence constraint unchanged, in which case w2 must be appended to the events propagated to tid; or
- it could be sent w3, again appending this to the events propagated to tid, which moreover entails committing to w3 being coherence-after w1, as in the coherence constraint on the right above. Note that this still leaves the relative order of w2 and w3 unconstrained, so another thread could be sent w2 then w3 or (in a different run) the other way around (or indeed just one, or neither).

Naïve message passing

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

Observable behaviour: r = 0



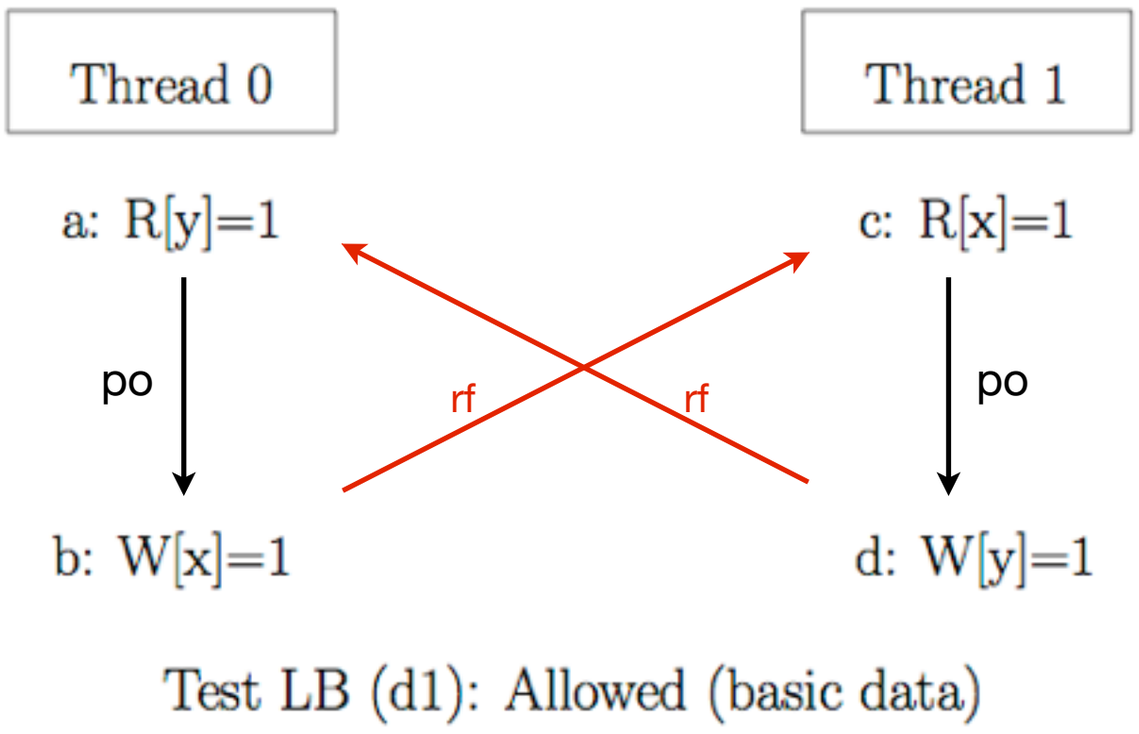
Allowed and **observed** on *Power*

Simulation: 5. message_passing

Load buffering

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

Observable behaviour: `r1 = r2 = 1`



Forbidden on *SC* and *x86-TSO*

Allowed and **observed** on *Power*

Simulation: 6. load_buffering

Power ISA 2.06 and ARM v7

Visible behaviour much weaker and subtle than x86.

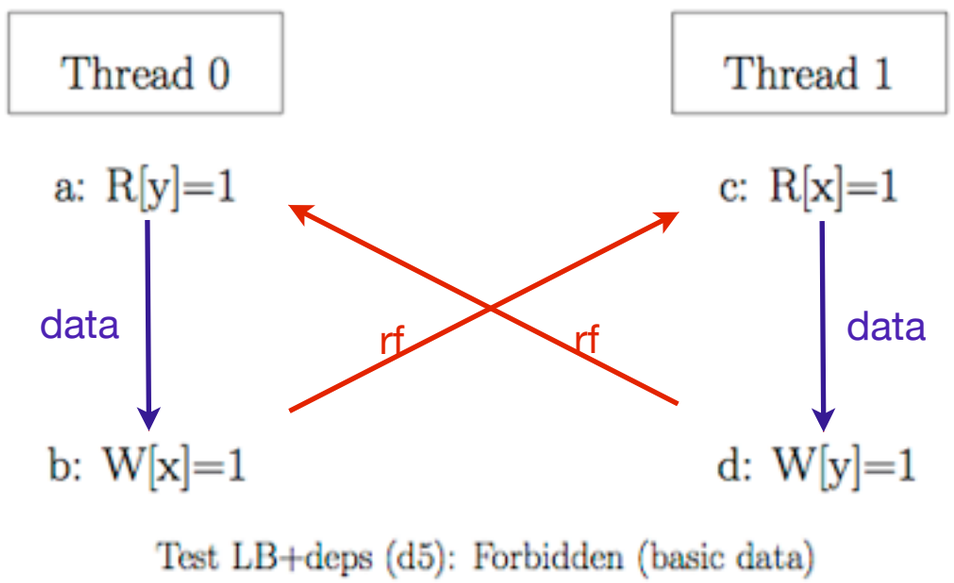
Basically, program order is **not preserved** unless:

- 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).

Load buffering with dependencies

LB+deps ARM

Thread 0	Thread 1
LDR R2, [R5] AND R3, R2, #0 STR R1, [R3,R4]	LDR R2, [R4] AND R3, R2, #0 STR R1, [R3,R5]
Initial state: 0:R1=1 ∧ 0:R4=x ∧ 0:R5=y ∧ 1:R1=1 ∧ 1:R4=x ∧ 1:R5=y	
Forbidden: 0:R2=1 ∧ 1:R2=1	



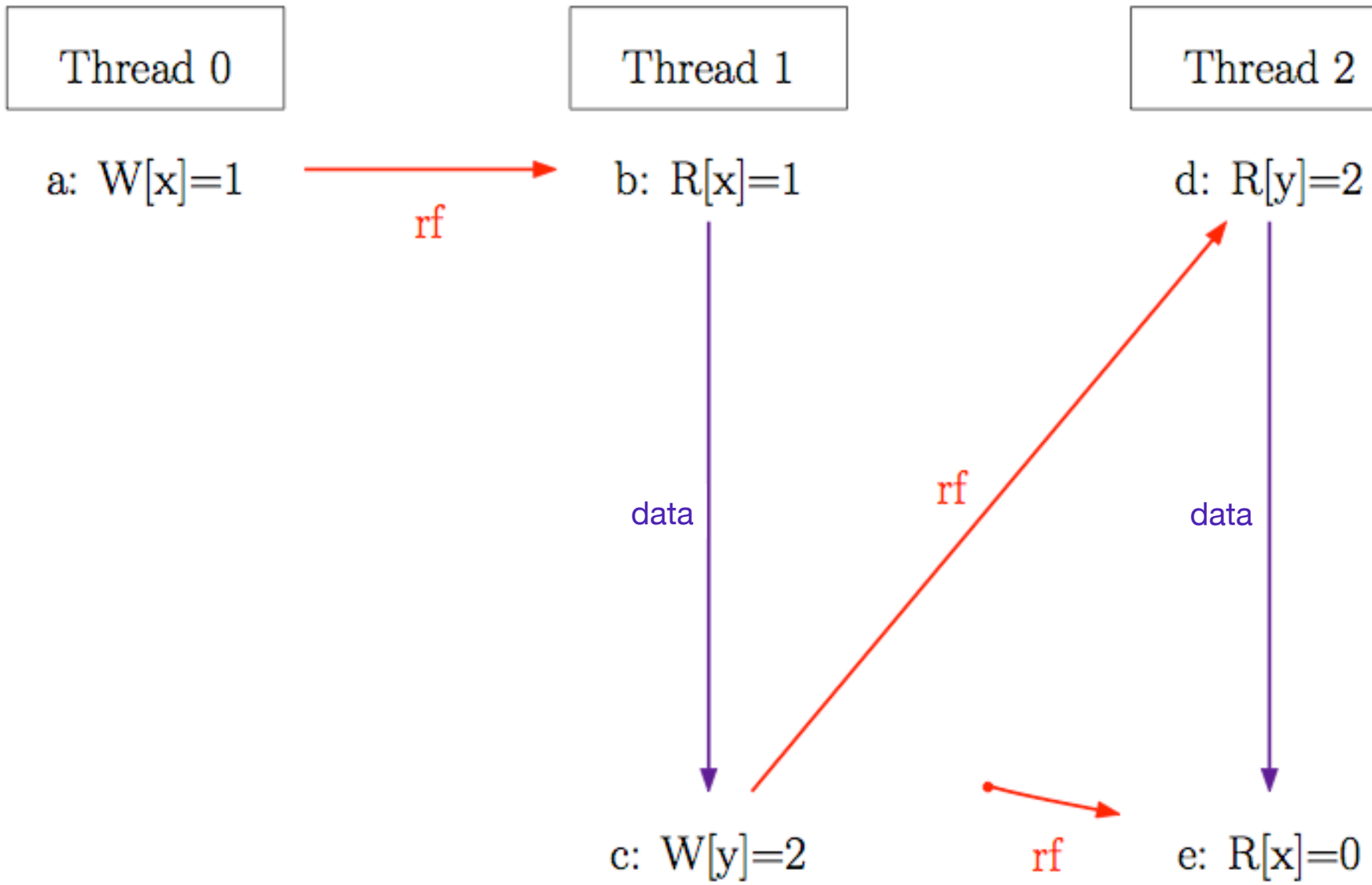
Test LB+deps (d5): Forbidden (basic data)

Simulation: 7. load_buffering_data_deps

Similarly with control dependencies, e.g.:

Play with examples in the LB directory

However dependencies might not be enough



Test WRC+deps (isa1v2): Allowed (basic data)

Exercise: WRC/WRC+addrs

Memory barriers

Power: ptesync, hwsync, lwsync, eieio

ARM: DSB, DMB

For each applicable pair a_i,b_j the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism.

- A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
- B includes all applicable storage accesses by any such processor or mechanism that are performed after a *Load* instruction executed by that processor or mechanism has returned the value stored by a store that is in B.

Memory bar

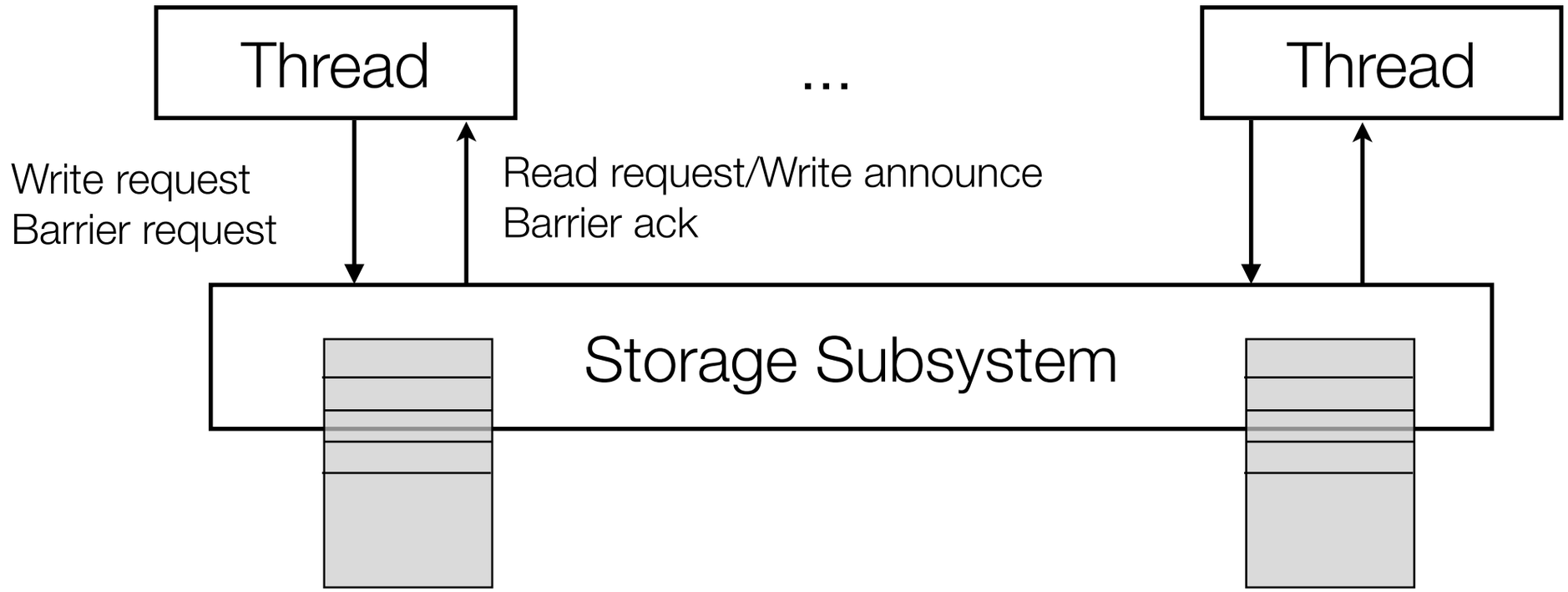
Power: ptesync, hw

ARM: DSB, DMB

For each a will be pe … res that a_i to the exte … echanism, quired att … rence Re- cessor or … that pro-

- A in … such proc … ed with resp … ed.
- B in … any such proc … er a *Load* inst … anism has returned the value stored by a store that is in B.

HWSYNC and LWSYNC



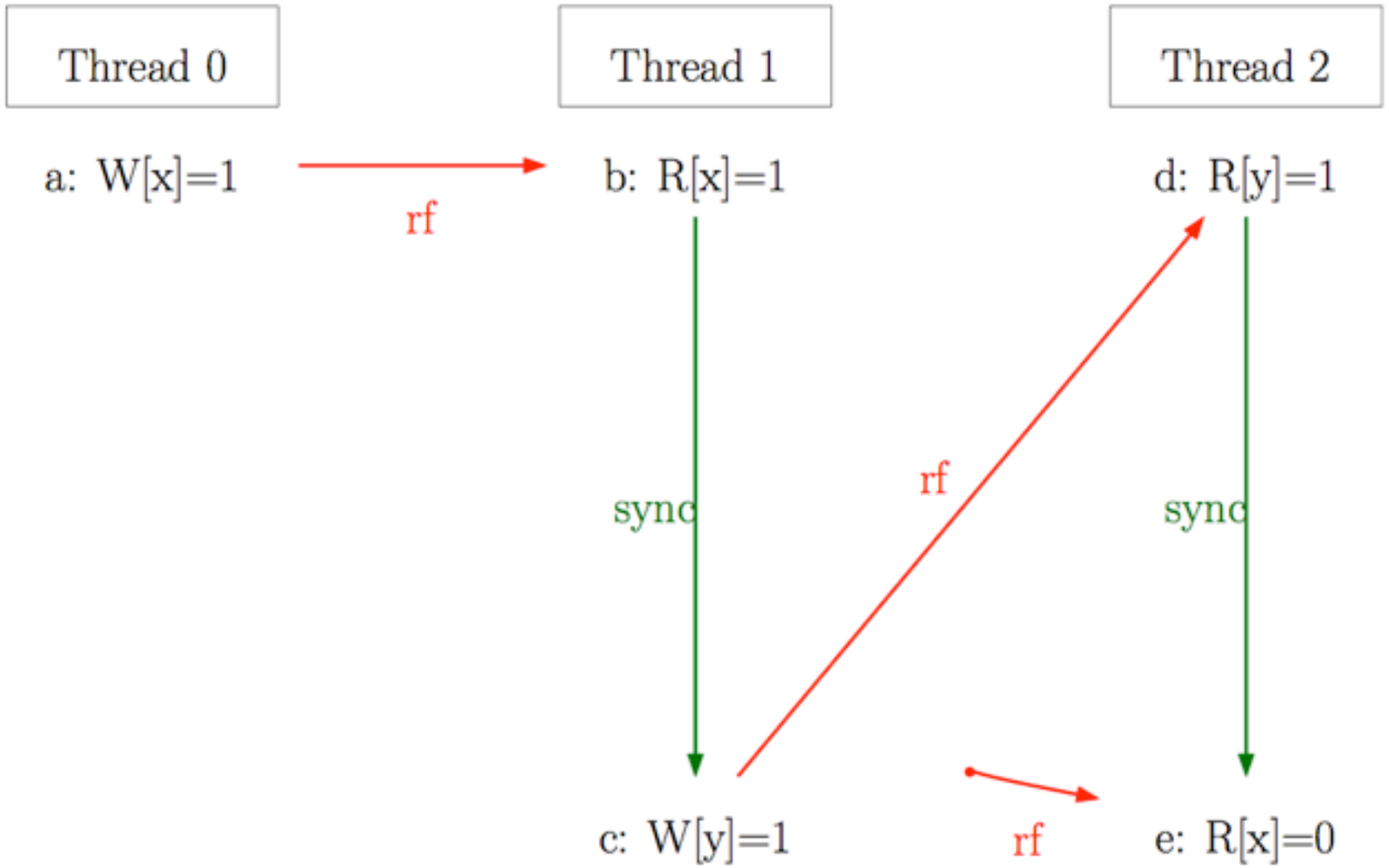
The storage accepts a barrier request (HWSYNC) from a thread.

The barrier request is added to the list of event propagated to that thread.

The thread cannot *execute* instructions following the barrier instructions without first receiving the barrier ack.

The storage sends the barrier ack only once all the preceding events have been propagated to all other threads.

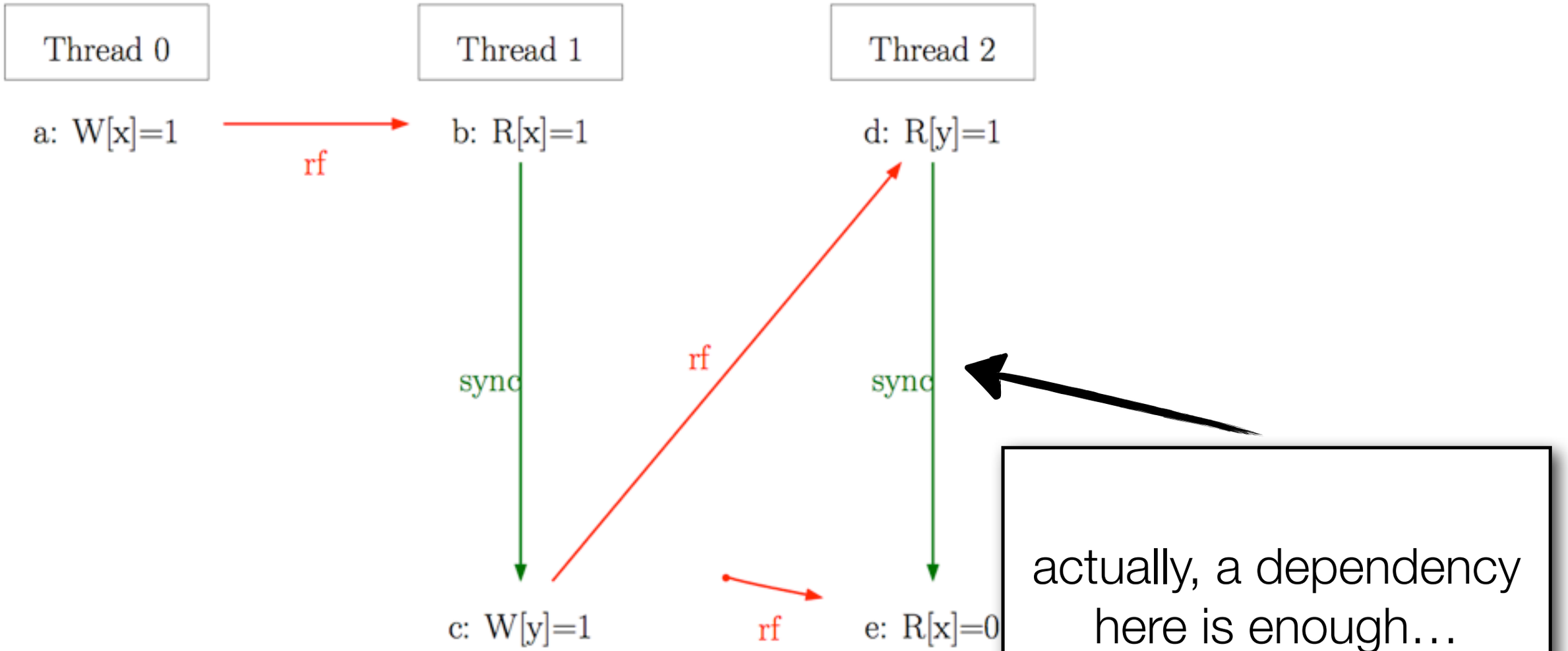
RWC with HWSYNC



Test WRC+syncs (m3s): Forbidden (basic data)

Simulation: WRC/WRC+syncs

RWC with HWSYNC



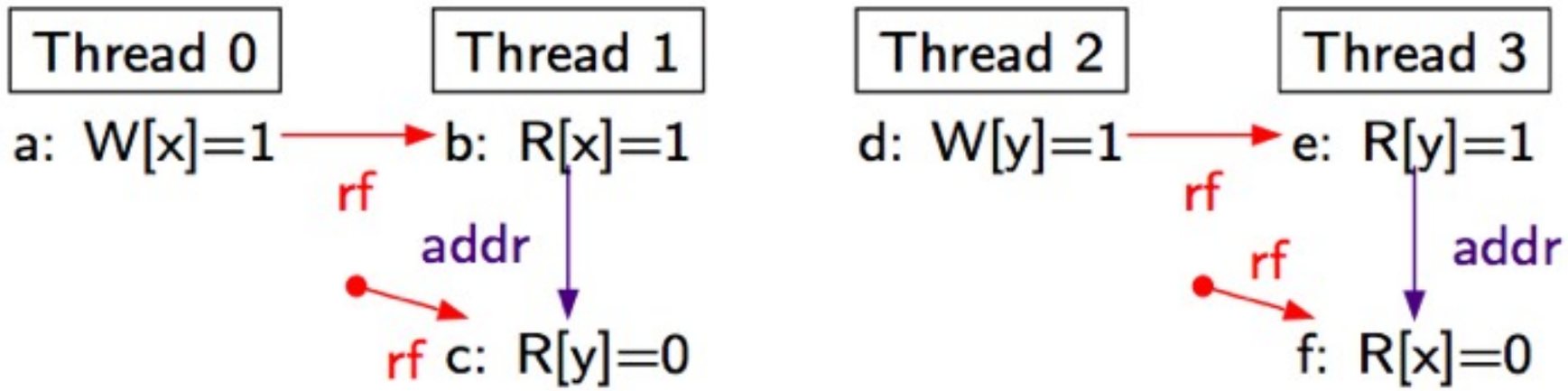
Test WRC+syncs (m3s): Forbidden (basic data)

actually, a dependency here is enough…

WRC/WRC+sync+addr

Simulation: WRC/WRC+syncs

IRIW



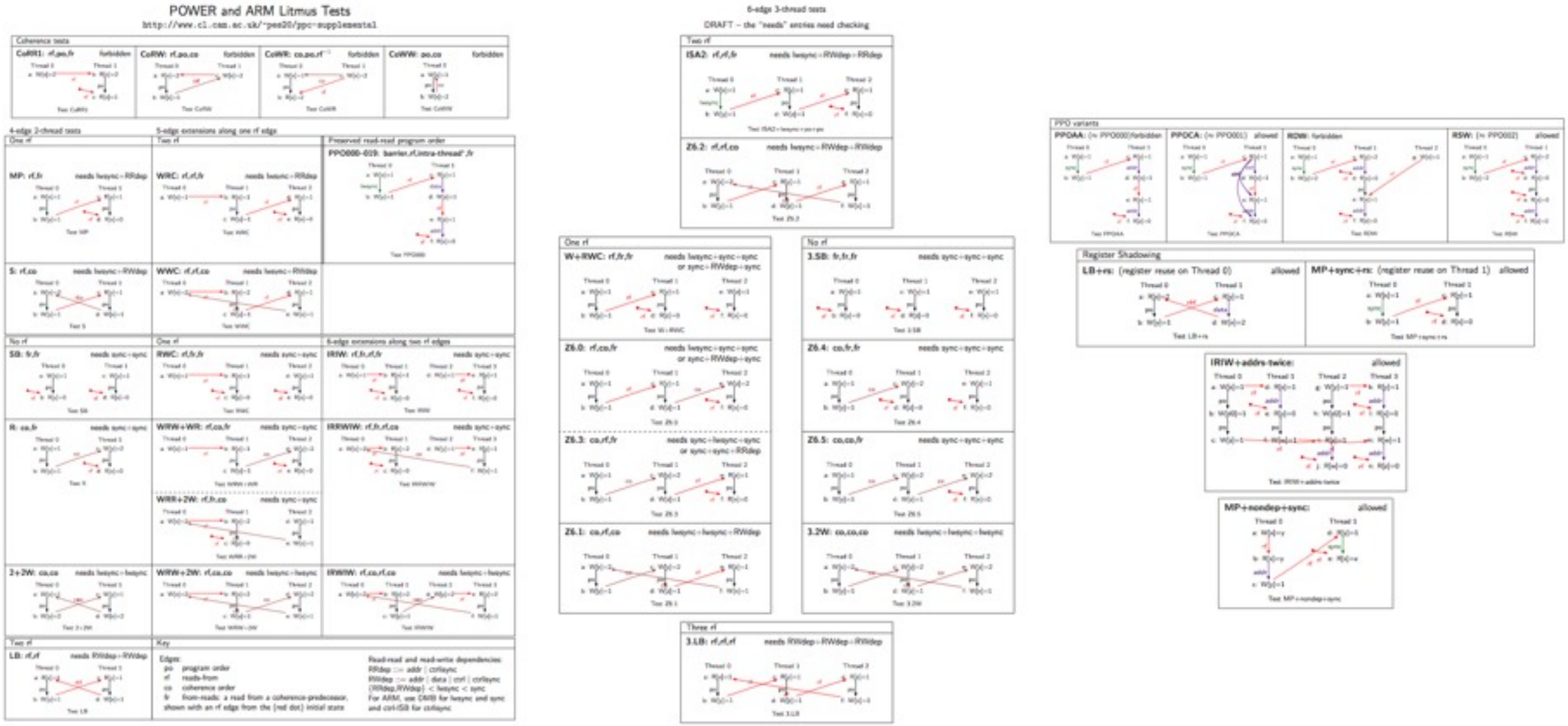
Test IRIW+addrs: Allowed

IRIW+addrs Pseudocode

Thread 0	Thread 1	Thread 2	Thread 3
x=1	r1=x r2=*(&y+r1-r1)	y=1	r3=y r4=*(&x+r3-r3)
Initial state: x=0 ∧ y=0 ∧ z=0			
Allowed: 1:r1=1 ∧ 1:r2=0 ∧ 3:r3=1 ∧ 3:r4=0			

Like SB, this needs two DMBs or syncs (lwsyncs not enough).

Periodic table of behaviour

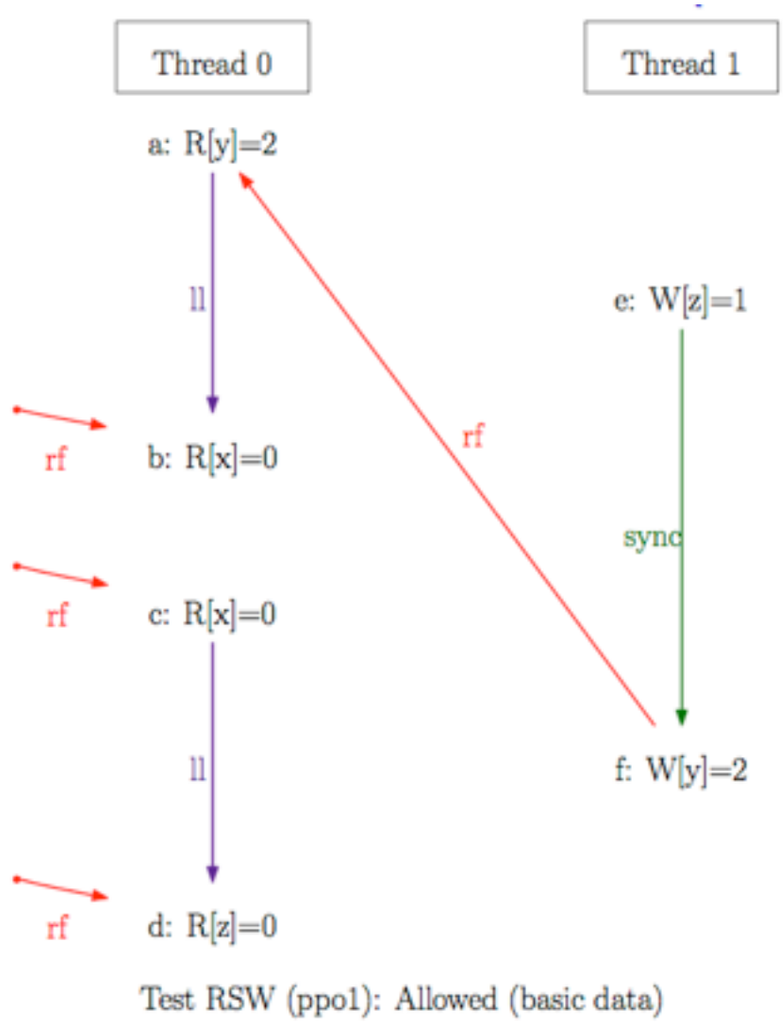


If you want more...

Go to

http://www.cl.cam.ac.uk/~pes20/ppcmem/

For each test, either find a trace that leads to the final state, or convince yourself that such trace does not exists. *Some tests are complicated...*



Tomorrow: The C and C++ memory model