Linux Kernel Memory Ordering Model: Concepts, Theories and Tools

Boqun Feng

Linux Technology Center, IBM

Self-Introduction

- 冯博群(Boqun Feng)
- Join IBM LTC at 2014 April, Working on Linux Kernel since then
- Focus on scheduler, locking, RCU and other core kernel areas

When We Talk about Memory Ordering

- What We Talk About?

A: Hey, We need you to work on memory ordering

B: OK! No Problem. How much memory we need to order?

A Real World Ordering Problem

- “内核和应用同时访问一个物理内存，比如大小为8字节,内核需要更新这个8字节内存对应的内容，应用去读这8字节物理内存里面的内容;反过来应 用有时也会去写，内核去读。有时写完后，去读时有时读取为-1，有时可能是其他值。尝试多次重新读取后，正确了。”
- Parallel Program and Shared Memory

Agenda

- Parallel Programming in Linux Kernel: Two Facts and One Question
- Primitives and Semantics
- The role of cycles
- Litmus Tests and Tools

Fact #1

- **More and More Complex but “Efficient” Parallel Programming Algorithms are getting into Linux Kernel.**
  - Filesystem: Pathname lookup rewrite: https://lwn.net/Articles/649115/
  - Network: Lockless TCP Listener: https://lwn.net/Articles/659199/
  - Even Locking Primitives: Qspinlock: https://lwn.net/Articles/590243/
- Why “Efficient”
  - Parallel programs are not definitely efficient
  - Parallel programs are not easily to be correct
- But Incorrect parallel programs are usually efficient.

Fact #2

- **Yet We Still Could Find Parallel Programming Bugs Existed in Linux Kernel For a Long Time**
  - spin_unlock_wait() on PPC and ARM
    - https://marc.info/?l=linux-kernel&m=144731258921696
  - spin_unlock_wait() for qspinlock on x86
    - https://marc.info/?l=linux-kernel&m=146372279722288
  - Race in try_to_wake_up()
    - https://marc.info/?l=linux-kernel&m=147263879404805

One Question

- How Could We Verify Our Parallel Code?
  - To ensure new “efficient” algorithms are correct
  - To find old bugs in kernel

Answer

- Before Year 2014, we had:
  - Documentation/memory-barriers.txt, the *Children-Frightener.*
  - Several people who could review patches for memory ordering issues.
    - Paul Mckenney, Peter Zijlstra, Will Deacon, etc.
- Around Year 2014 and 2015, things are better, because we had more:
  - Me ;-)
  - Several researches and tools for arch-specific memory ordering models:
    - PPCMEM and ARMMEM, and related researches
    - Herd and related researches
- Now and Future, we are going to have:
  - A Formal Ordering Model:
    - LCE 2016: *Linux-Kernel Memory Ordering: Help Arrives At Last!*
    - by Paul E. Mckenney, Jade Alglave, Luc Maranget, Andrea Parri, and Alan Stern and more

When We Talk about Memory Ordering

- What We Talk About?
  - Things that could reorder memory operations
  - Things that could prevent reordering, IOW, could order memory operations.

Things That Could Reorder Memory Ops

- Compiler
  - ”A compiler is within its right to”…
  - *N4455: No Sane Compiler Would Optimize Atomics*
- Hardware
  - Out-of-order execution
  - Store buffer and invalidate queue.

From a viewpoint of a memory ordering model, nothing is ordered, unless the model says yes.

Things That Order Memory Operations

- Barriers
  - Compiler Barrier: barrier()
  - Full barrier: mfence(x86), sync(PPC)
  - Partial Barrier: lwsync
  - Operation with Barrier Semantics: ldaxr, stlxr(AARCH64)
- Dependencies
- Arch-dependent Intrinsic Ordering
  - TSO, SC

Based on which, we build Linux internal primitives on ordering.

Agenda

- Parallel Programming in Linux Kernel: Two Facts and One Question
- **Primitives and Semantics**
- The role of cycles
- Litmus Tests and Tools

Relationship Between Memory Ops

- Ordering is one kind of relationship on the set of memory operations.
- And ordering primitives and concepts will make some relationships special
  - i.e. Make a relationship provide ordering (in some cases).
- Let’s go through some kernel primitives relationship-wisely.

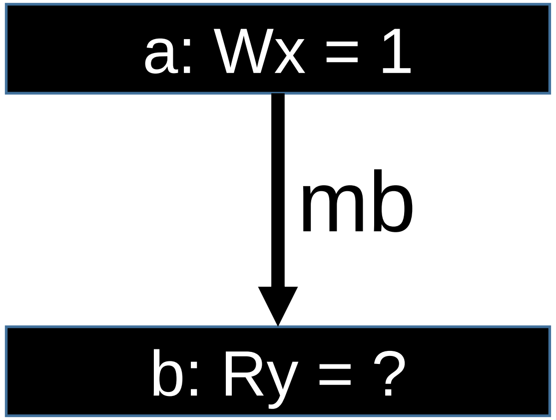
Linux Primitives for Memory Ordering

- smp_mb()
- smp_wmb() and smp_rmb()
- smp_load_acquire() and smp_store_release()
- lockless_dereference(), rcu_dereference() and rcu_assign_pointer()

smp_mb()

- A full barrier
  - Orders READ->READ, READ->WRITE, WRITE->READ, WRITE->WRITE
  - Provides Transitivity
  - Implemented as mfence on x86_64, sync on PPC

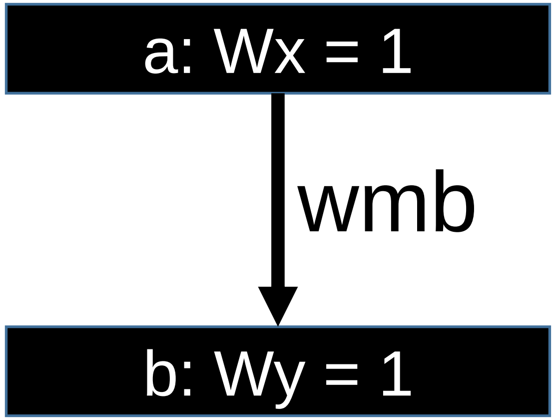
```
WRITE_ONCE(*x, 1);

smp_mb();

r1 = READ_ONCE(*y, 1);
```



smp_wmb() and smp_rmb()

- Pairtial barriers:
  - smp_wmb() orders WRITE->WRITE
  - smp_rmb() orders READ->READ
  - Implemented as lwsync on PPC

```
WRITE_ONCE(*x, 1);

smp_wmb();

WRITE_ONCE(*y, 1);
```

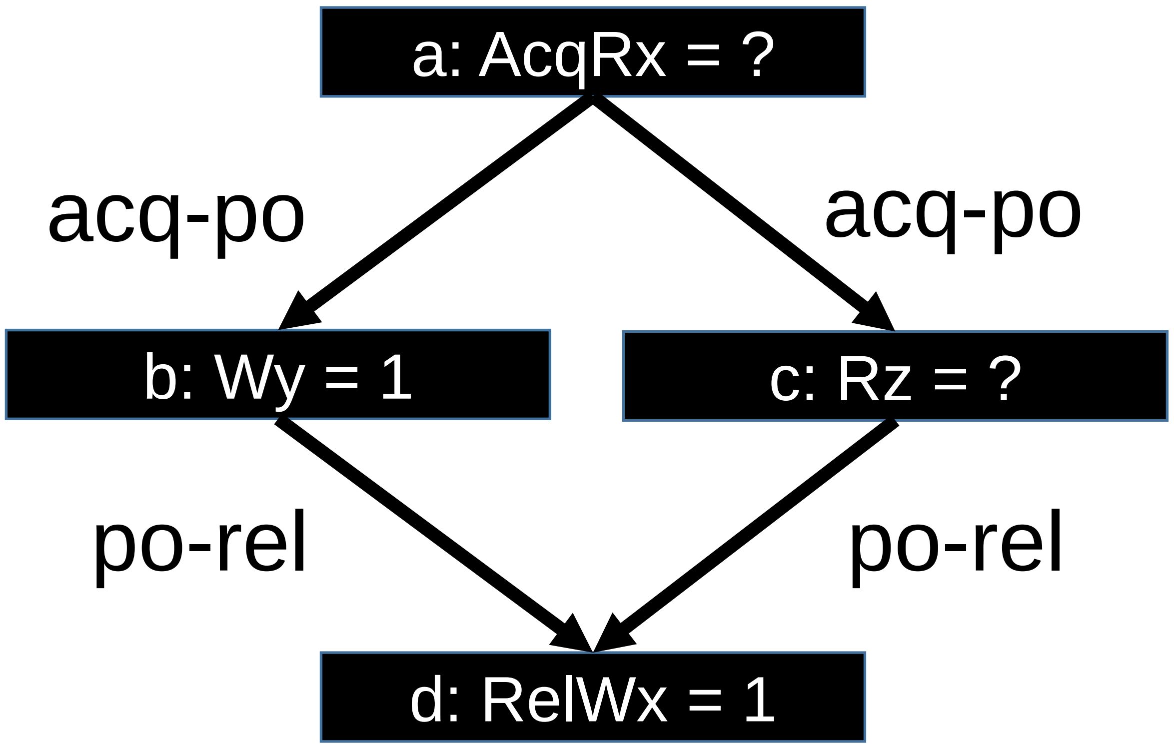


smp_load_acquire() and smp_store_release()

- smp_load_acquire()
  - A load
  - Every operation (program-order) after smp_load_acquire() will not happen before load.
  - One-way barrier
- smp_store_release()
  - A store
  - Every operation (program-order) before smp_store_release() will not happen after the store
  - One-way barrier

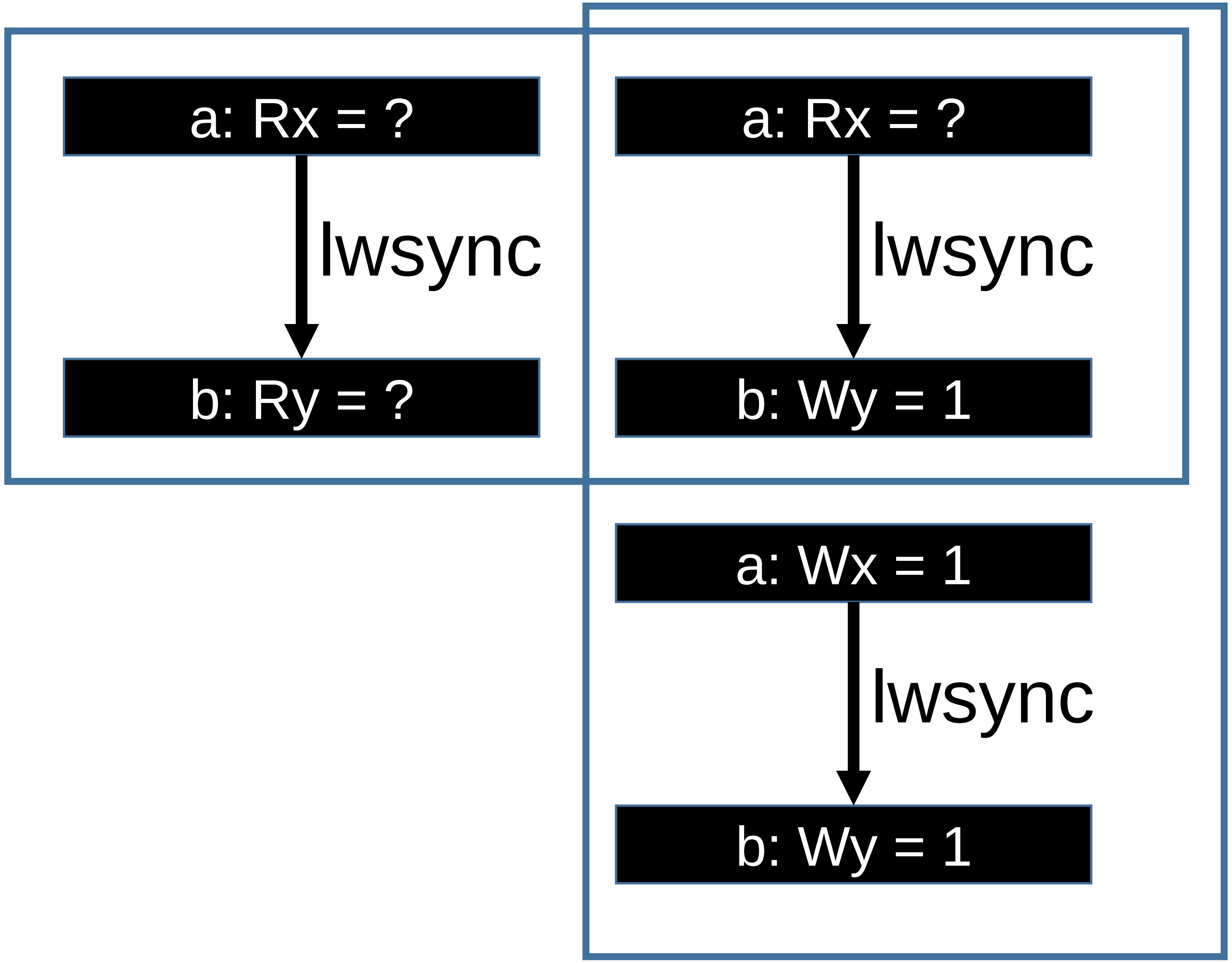
smp_load_acquire() and smp_store_release()

```
WRITE_ONCE(*a, 1)

r1 = smp_load_acquire(x);

WRITE_ONCE(*y, 1);

r2 = READ_ONCE(*z);

smp_store_release(x, 1);

r3 = READ_ONCE(*a);
```



ACQUIRE and RELEASE: Implementation

- lwsync on PPC
  - Orders READ->READ
  - Orders READ->WRITE
  - Orders WRITE->WRITE
- ACQUIRE: load+lwsync
- RELEASE: lwsync+store



*_dereference()

- lockless_dereference() and rcu_dereference()
  - Provides ordering between memory operations having data/address dependencies.
- rcu_assign_pointer()
  - RELEASE?

Dependencies

- Data/Address
  - READ->{READ, WRITE}
- Control
  - READ->WRITE

```
<Data/Address Dependencies>
struct T *ptr = READ_ONCE(*p);


tmp = ptr->a; // or ptr->a = 1;
```

$$\begin{array}{c} \text{a: Rp = ?} \\ \Big\downarrow \text{addr} \\ \text{b: Ra = ?} \end{array}$$

```
<Control Dependencies>
struct T *ptr = READ_ONCE(*p);

if (ptr) {
        WRITE_ONCE(x, 1);
        …
}
```

$$\begin{array}{c} \text{a: Rp = ?} \\ \Big\downarrow \text{ctrl} \\ \text{b: Wx = 1} \end{array}$$

Orders Provided by Linux Kernel Primitives

- Control Dependencies if no compiler reordering involved
- Data Dependencies if leaded by a *_dereference() primitives
- ACQUIRE and RELEASE ordering by their definitions, i.e. one way barrier
- Memory Operations before and after smp_mb()
- Reads before and after smp_rmb()
- Writes before and after smp_wmb()

More: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0124r2.html

Agenda

- Parallel Programming in Linux Kernel: Two Facts and One Question
- Primitives and Semantics
- **The role of cycles**
- Litmus Tests and Tools

Communication

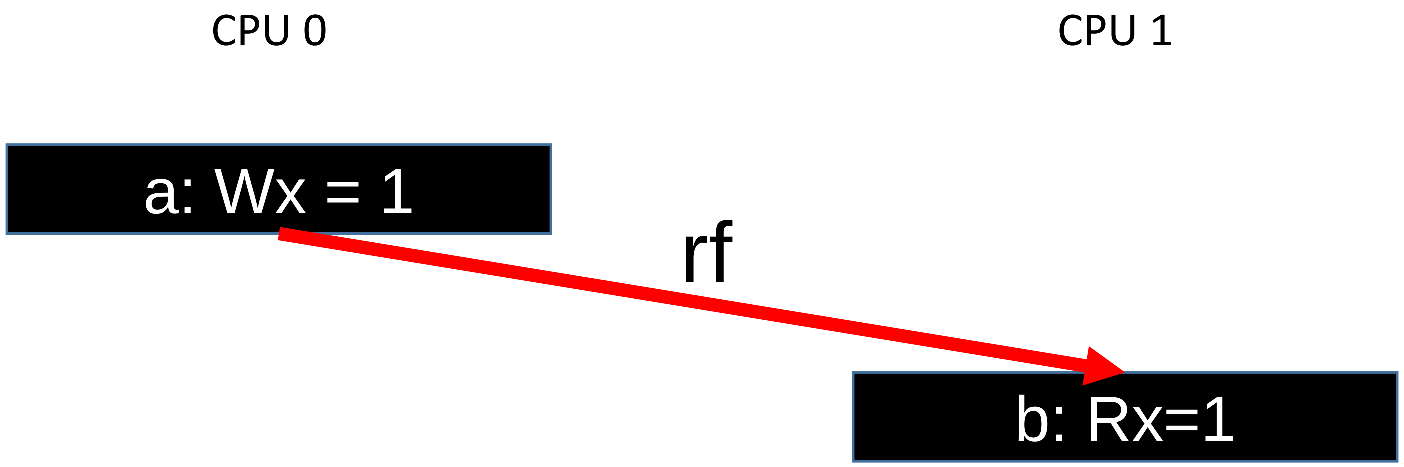
- Orders provided on one CPU are useless unless in the context of communication.
- We usually use the word ”Pairing” or “Synchronize” to describe an communication.

Ways of Communication

- READ FROM
- COHERENCE
- FROM READ

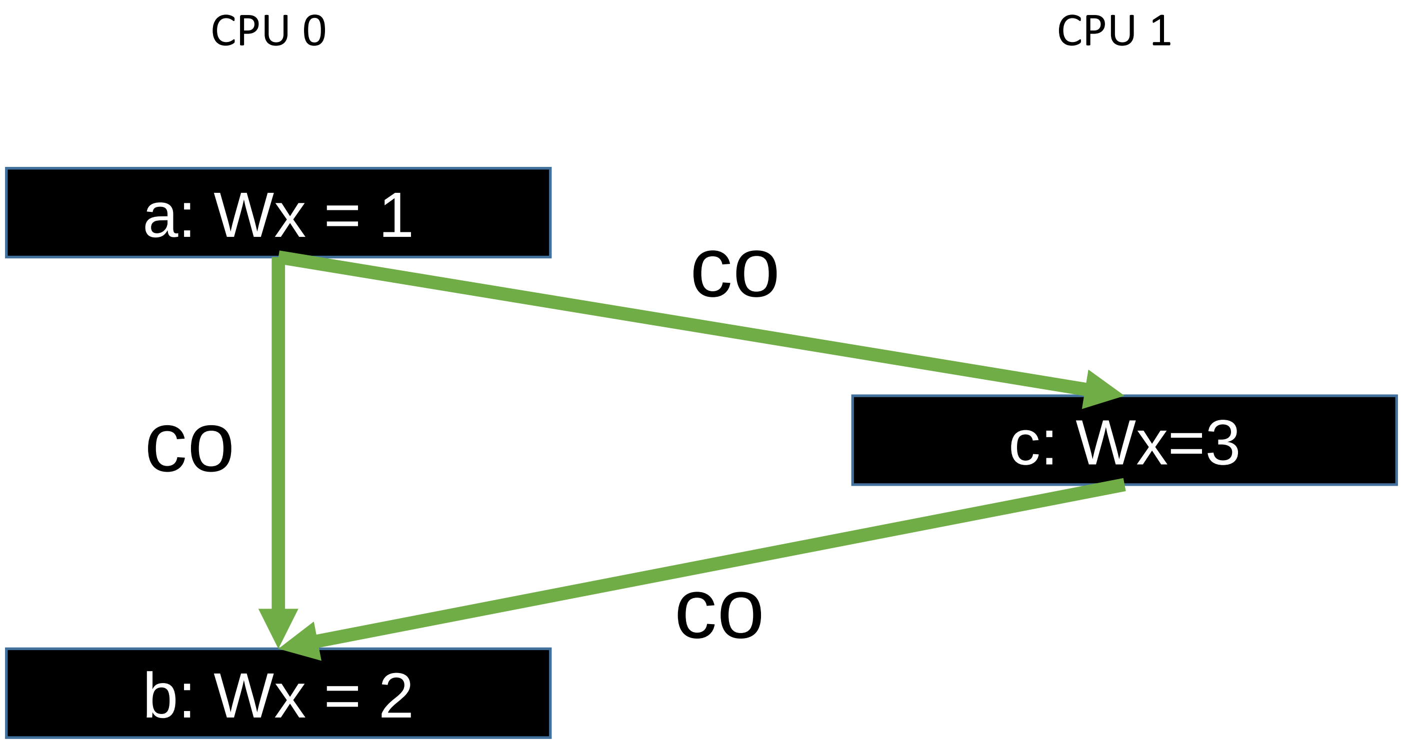
READ FROM(rf)

- A READ observes the value from a WRITE



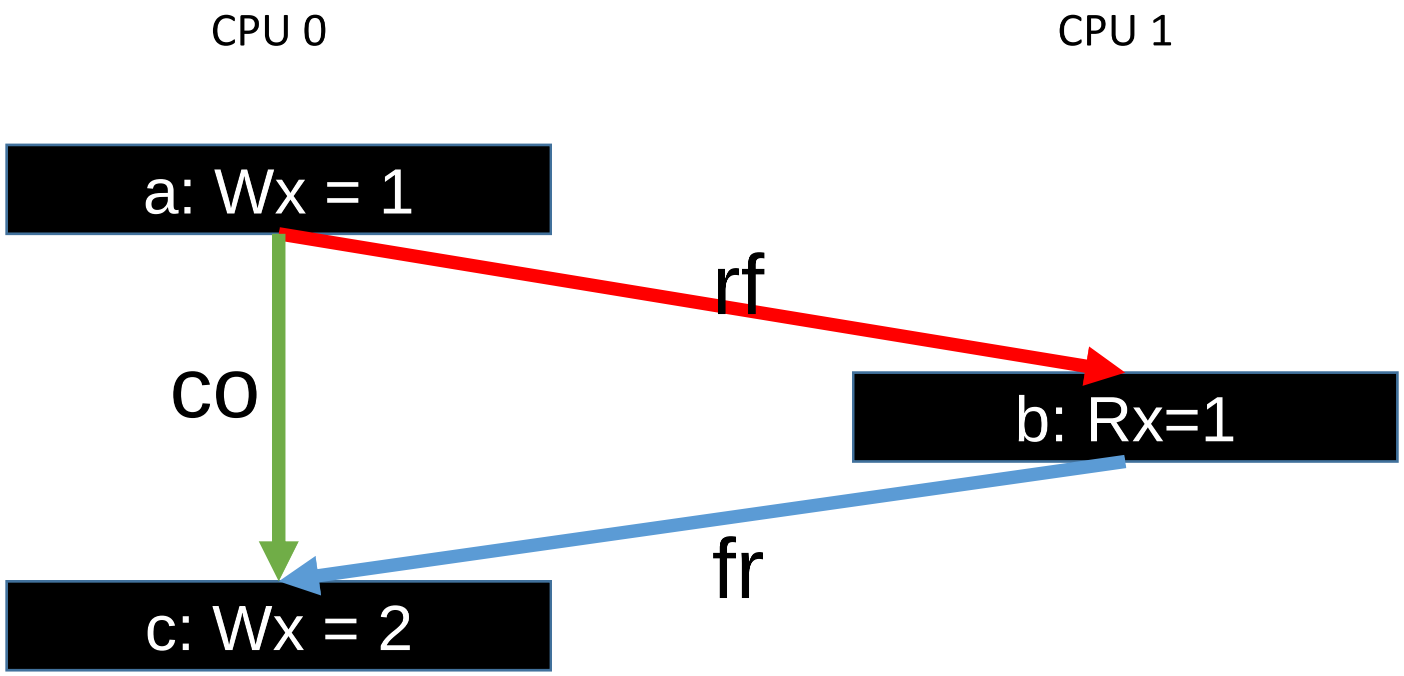
COHERENCE (co)

- A WRITE observes another WRITE via overwriting.
- A Total Order Relationship on WRITEs to a same variable.



FROM READ(fr)

- A WRITE observes a READ via failing to be observed by the READ



Cycles: Questions

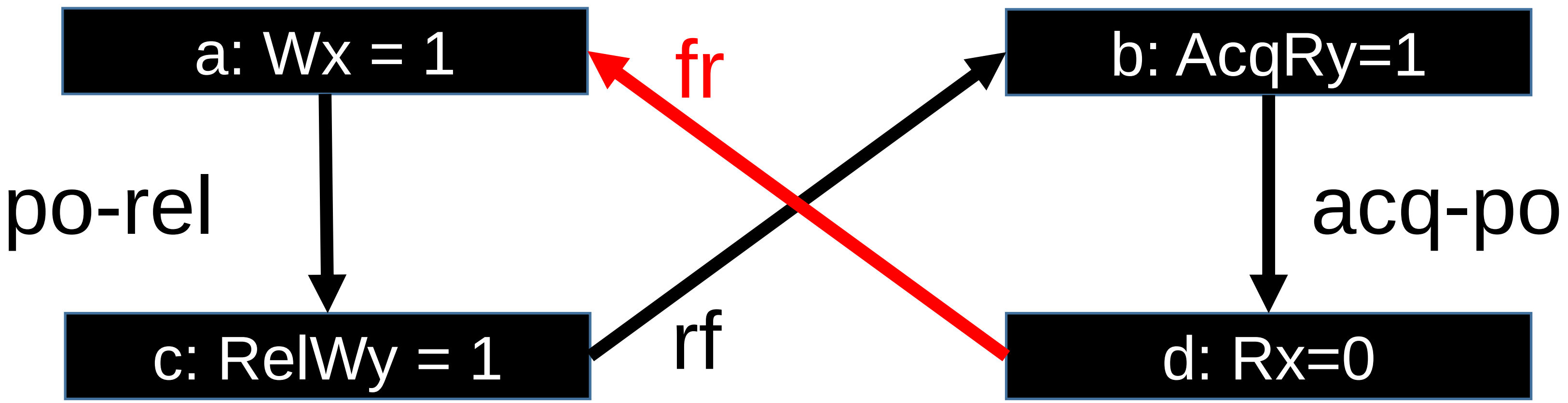
- x and y are zeros initially
- P0 executes on CPU 0
- P1 executes on CPU 1
- if r1 == 1 will r2 == 0?

```
P0(int *x, int *y)
{
        WRITE_ONCE(*x, 1);
        smp_store_release(y, 1);
}

P1(int *x, int *y)
{
        int r1, r2;

        r1 = smp_load_acquire(y);
        r2 = READ_ONCE(*x);
}
```

Cycles

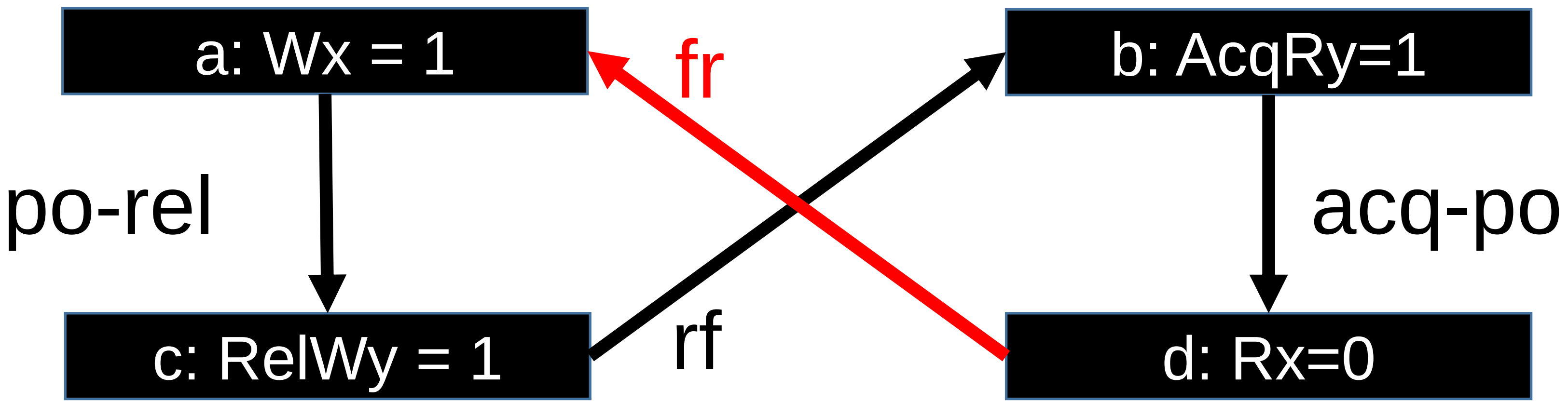
- if r1 == 1 will r2 == 0?

```
P0(int *x, int *y)
{
        WRITE_ONCE(*x, 1);
        smp_store_release(y, 1);
}

P1(int *x, int *y)
{
        int r1, r2;

        r1 = smp_load_acquire(y);
        r2 = READ_ONCE(*x);
}
```



Cycles

- Cycles of relationships may mean cycles of timings
- Do we allow the following cycle?
- What does it mean if we disallow this cycle?



Cycles for modeling

- Memory ordering model could be defined as what kind of cycles of relationship we should prohibit.
- And cycles are easily to detect by tools

Cycles Prohibits in Linux Kernel(In Informal Words)

- Cycles built by communication relationship(com, which consists of **rf**, **fr** and **co**).
- Cycles built by **po-loc**(program order for a memory location, relationship between memory operations on the some variable with program order) or **po**
- Cycles built by either **com** or **po-loc**, this is actually cache coherence.
- Cycles built by at most one **fr** or **rf** or **acq-po** or **po-rel** or other relationship provided by kernel primitives, whose corresponding program pattern fulfills maybe 80% daily use.
- For more? See the formal model.

Agenda

- Parallel Programming in Linux Kernel: Two Facts and One Question
- Primitives and Semantics
- The role of cycles
- **Litmus Tests and Tools**

Litmus Tests

- Pseudo parallel code fragments
- A baby step to verify a whole project
  - Useful for discussions
  - Easy to name and check

```
{ x = 0; y = 0}
P0(int *x, int *y)
{
	WRITE_ONCE(*x, 1);
	smp_store_release(y, 1);
}
P1(int *x, int *y)
{
	int r1, r2;

	r1 = smp_load_acquire(y);
	r2 = READ_ONCE(*x);
}
exists
(1:r1 = 1 /\ 1:r2 = 0)
```

Three Parts of Litmus Tests

- Initial Value
- Code
- Exist-clause: An Assertion

```
{ x = 0; y = 0}

P0(int *x, int *y)
{
        WRITE_ONCE(*x, 1);
        smp_store_release(y, 1);
}
P1(int *x, int *y)
{
        int r1, r2;

        r1 = smp_load_acquire(y);
        r2 = READ_ONCE(*x);
}

exists
(1:r1 = 1 /\ 1:r2 = 0)
```

Litmus Tests Could be Code or Assembly

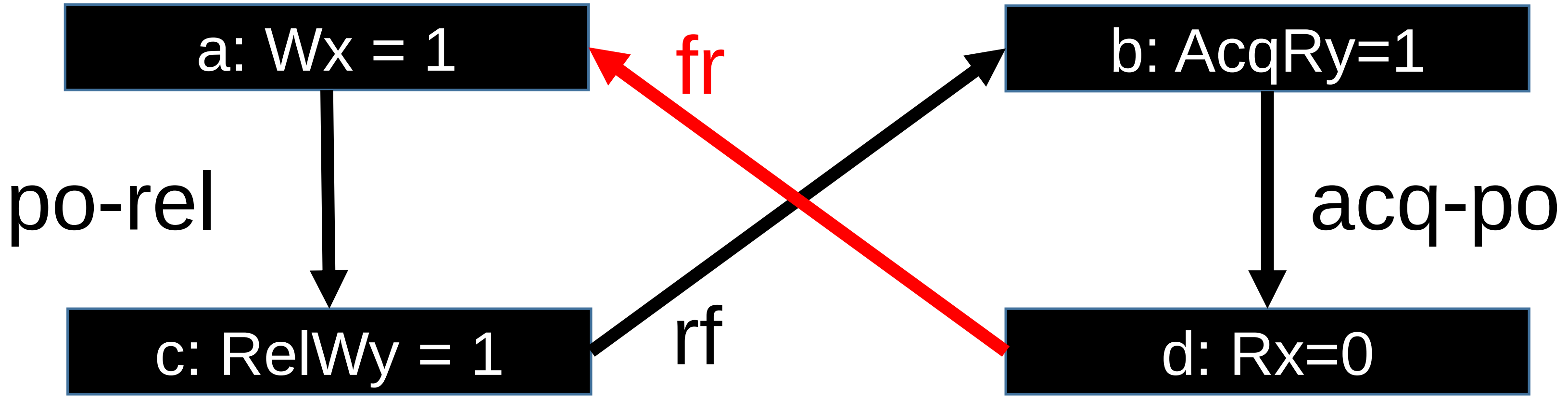
```
{ x = 0; y = 0}
P0(int *x, int *y)
{
	WRITE_ONCE(*x, 1);
	smp_store_release(y, 1);
}
P1(int *x, int *y)
{
	int r1, r2;

	r1 = smp_load_acquire(y);
	r2 = READ_ONCE(*x);
}
exists
(1:r1 = 1 /\ 1:r2 = 0)
```

```
{
0:r10=x;0:r11=y;0:r3=1
1:r10=x;1:r11=y;
x=0;y=0;
}

P0               | P1              ;
stw r3, 0(r10)  | lwz r1, 0(r11) ;
lwsync          | lwsync         ;
stw r3, 0(r11)  | lwz r2, 0(r10) ;

exists
(1:r1 = 1 /\ 1:r2 = 0)
```

Litmus Tests Could be Checked by Tools

Never

```
States 3
1:r1=0; 1:r2=0;
1:r1=0; 1:r2=1;
1:r1=1; 1:r2=1;
No
Witnesses
Positive: 0 Negative: 3
Condition exists (1:r1=1 /\ 1:r2=0)
Observation rel-acq-mp Never 0 3
```



Tools -- PPCMEM

- PPCMEM
  - An Emulator that could emulate modeled memory events of PPC architectures.
  - Input: a small code snippet called litmus
  - Output: result of all possible executions of the given litmus test, and whether a designed assertion fails or not.
  - Pros: Having an interactive interface for understanding the hardware behavior
  - Cons: Slow and not "portable" for other models.

Tools -- Herd

- Herd
  - Generate all possible execution candidates, and examine each candidate with a given model, leave the ones survive after the examination as the possible result of the model.
  - Input: a litmus test and a memory ordering model(having default models for x86, AARCH, PPC, etc.)
  - Output: result of all possible executions of the given litmus test, and whether a designed assertion fails or not.
  - Pros: Fast, and allow self-defined models
  - Cons: Don‘t have a interactive interface.

Conclusion

- We are going to have more and more “fun” in parallel program in kernel
- Luckily after 25 years of development, we have more docs and tools to help us.
- In the future, we will have a more formal model and hopefully, it could solve all the problems in kernel parallel programming ;-)

Q & A

A singleton

Work on a single thread

```
struct T {
	int a;
};
…
static struct T *instance;

struct T *get_instance()
{
	if (instance == NULL) {
		instance = malloc(sizeof(struct T));
		instance->a = some_value;
	}
	return instance;
}
```

A multi-thread-safe singleton: first try

```
static struct T *instance;

struct T *get_instance()
{
        if (instance == NULL) {
                instance = malloc(sizeof(struct T));
                instance->a = some_value;
        }
        return instance;
}
```

won't work:

race when get_instance() is called by two thread in the same time.

malloc twice.

A multi-thread-safe singleton

```
static struct T *instance;
static spinlock_t instance_lock;

struct T *get_instance()
{
	if (instance == NULL) {
		spin_lock(&instance_lock);
		if (instance == NULL) {
			instance = malloc(sizeof(struct T));
			instance->a = some_value;
		}
		spin_unlock(&instance_lock);
	}
	return instance;
}
// won't work, may observe an uninitialized value of ->a
```

A multi-thread-safe singleton: simple solution

```
static struct T *instance;
static spinlock_t instance_lock;

struct T *get_instance()
{
        spin_lock(&instance_lock);
        if (instance == NULL) {
                instance = malloc(sizeof(struct T));
                instance->a = some_value;
        }
        spin_unlock(&instance_lock);
        return instance;
}
```

Simple solution:
Adding a lock.

Work, but need to acquire the lock every time

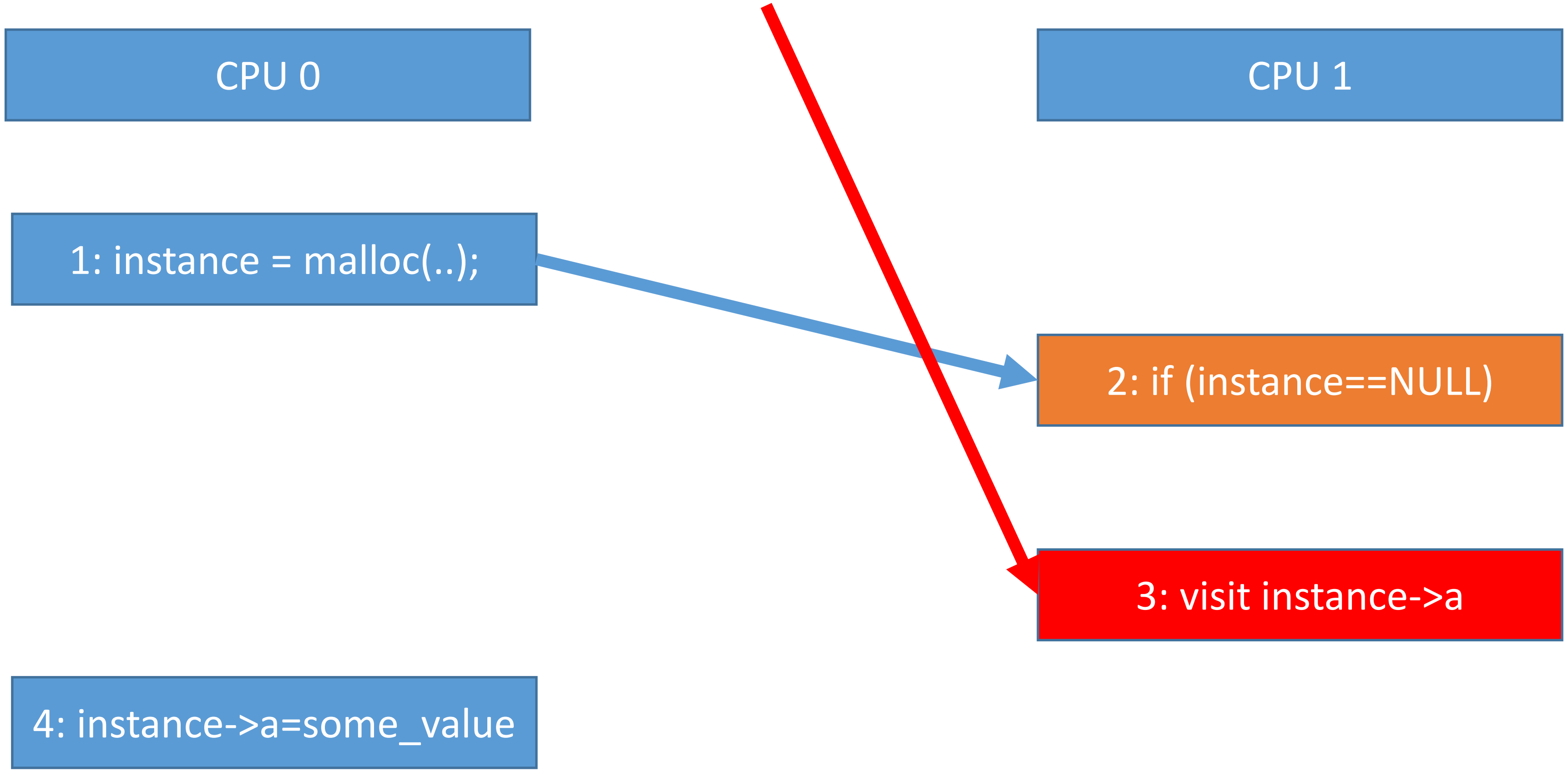
A multi-thread-safe singleton: try to optimize

```
static struct T *instance;
static spinlock_t instance_lock;

struct T *get_instance()
{
        if (instance == NULL) {
                spin_lock(&instance_lock);
                if (instance == NULL) {
                        instance = malloc(sizeof(struct T));
                        instance->a = some_value;
                }
                spin_unlock(&instance_lock);
        }
        return instance;
}
```

Check first?

Won’t work, Why?

A multi-thread-safe singleton



CPU 1 reads the uninitialized value of ->a

A multi-thread-safe singleton: Use barrier()

```
static struct T *instance;
static spinlock_t instance_lock;

struct T *get_instance()
{
        struct T *tmp;
        if (instance == NULL) {
                spin_lock(&instance_lock);
                if (instance == NULL) {
                        tmp = malloc(sizeof(struct T));
                        tmp->a = some_value;

                        barrier();

                        instance = tmp;
                }
                spin_unlock(&instance_lock);
        }
}
```

Initialization before publication!

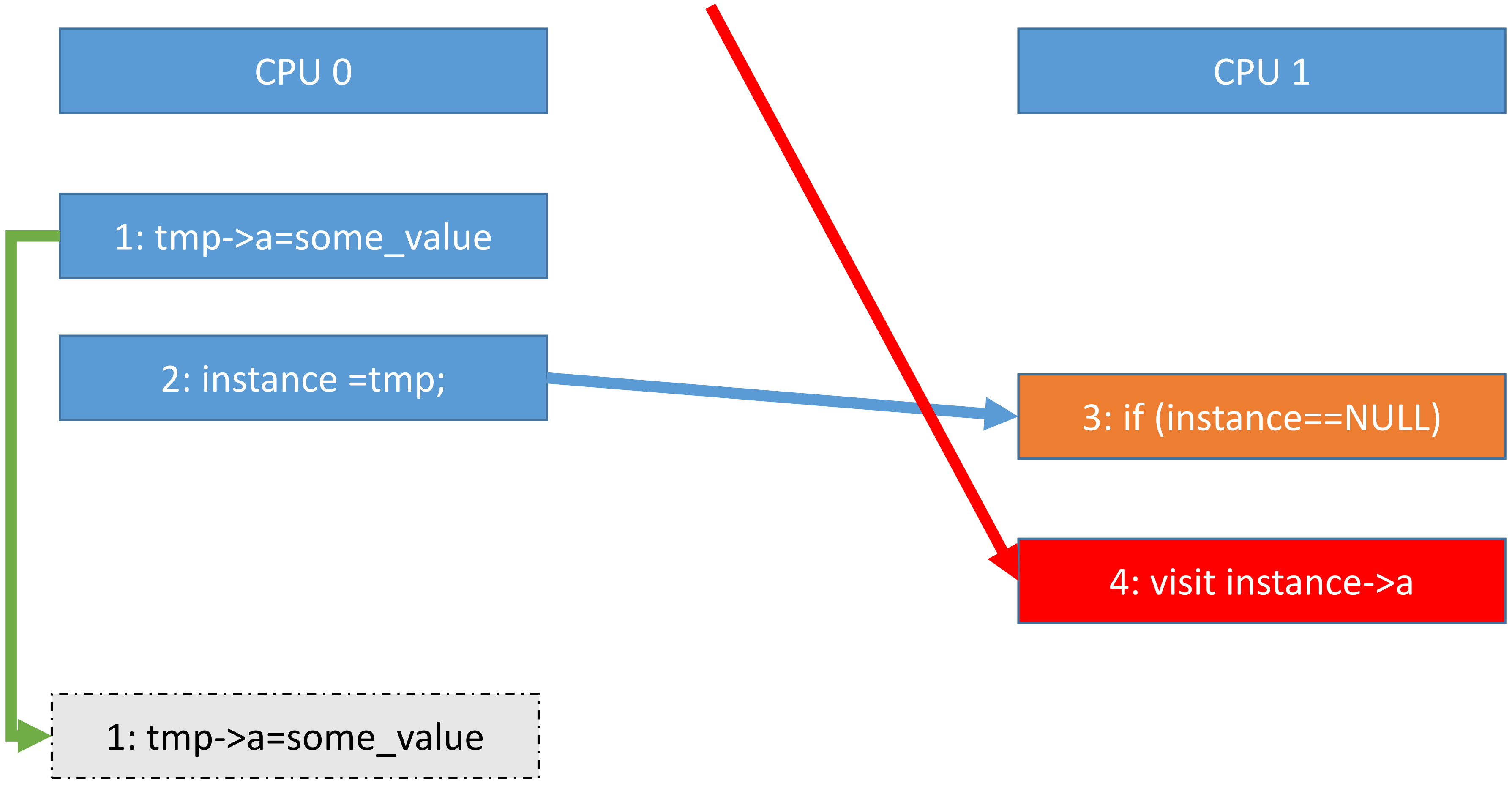
Why barrier() here?

barrier():

asm volatile(“” : : : “memory”);

Are we done?

A multi-thread-safe singleton: HW Reordering



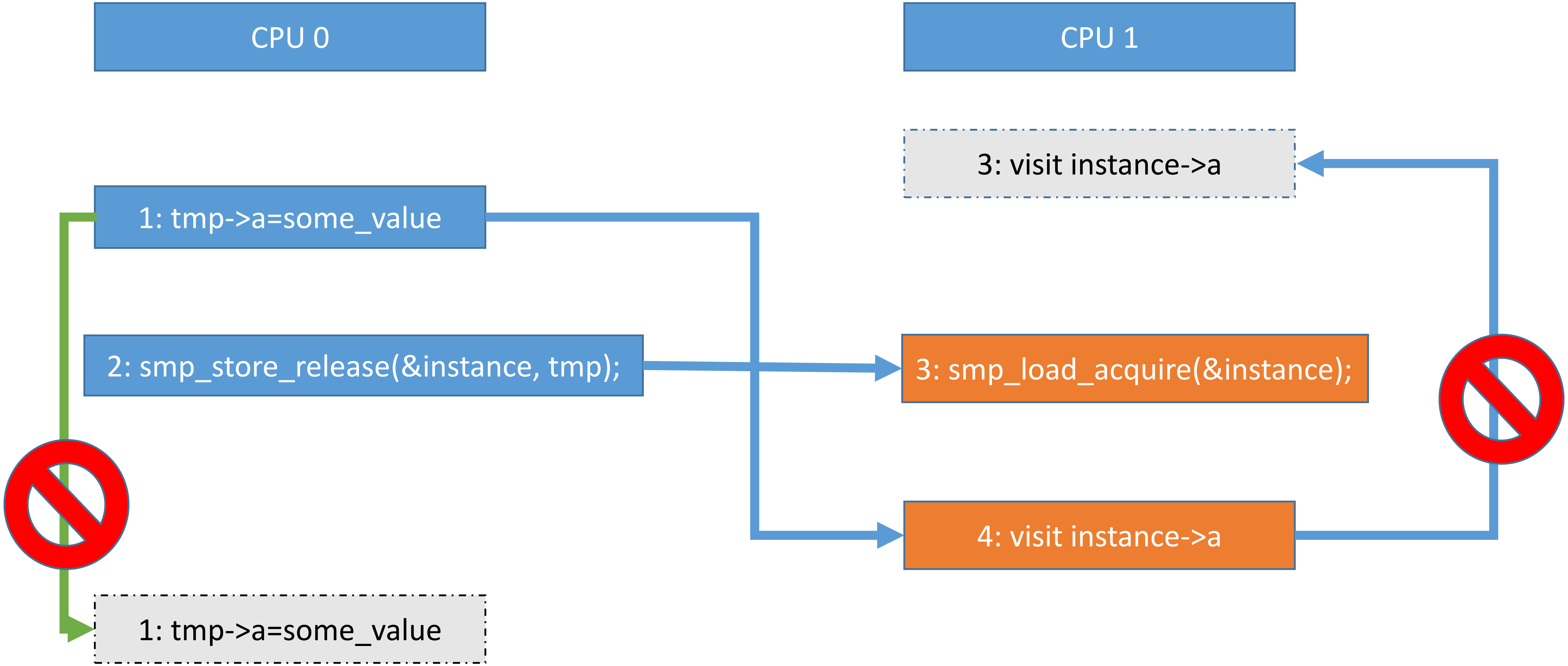
CPU 1 still reads the uninitialized value of ->a

A multi-thread-safe singleton: Final Solution

Use acquire and release.

```
static struct T *instance;
static spinlock_t instance_lock;

struct T *get_instance()
{
        struct T *tmp = smp_load_acquire(&instance);
        if (tmp == NULL) {
                spin_lock(&instance_lock);
                if (instance == NULL) {
                        tmp = malloc(sizeof(struct T));
                        tmp->a = some_value;
                        smp_store_release(&instance, tmp);
                }
                spin_unlock(&instance_lock);
        }
        return tmp;
}
```

Reordering is prohibited.



A multi-thread-safe singleton: Final Solution

Good enough?

```
static struct T *instance;
static spinlock_t instance_lock;

struct T *get_instance()
{
	struct T *tmp = smp_load_acquire(&instance);
	if (tmp == NULL) {
		spin_lock(&instance_lock);
		if (instance == NULL) {
			tmp = malloc(sizeof(struct T));
			tmp->a = some_value;
			smp_store_release(&instance, tmp);
		}
		spin_unlock(&instance_lock);
	}
	return tmp;
}
```