

MariaDB<sup>®</sup>  
FOUNDATION

## Memory barriers in C

Sergey Vojtovich  
Software Engineer @ MariaDB Foundation



# Agenda

- Normal: overview, problem, Relaxed
- Advanced: Acquire, Release
- Nightmare: Acquire\_release, Consume
- Hell: Sequentially consistent
- Summoning Cthulhu: Atomic thread fence



# Abbreviations

```
#define RELAXED MY_MEMORY_ORDER_RELAXED
#define CONSUME MY_MEMORY_ORDER_CONSUME
#define ACQUIRE MY_MEMORY_ORDER_ACQUIRE
#define RELEASE MY_MEMORY_ORDER_RELEASE
#define ACQ_REL MY_MEMORY_ORDER_ACQ_REL
#define SEQ_CST MY_MEMORY_ORDER_SEQ_CST

#define load my_atomic_load32_explicit
#define store my_atomic_store32_explicit
#define fas my_atomic_fas32_explicit
#define add my_atomic_add32_explicit
#define cas my_atomic_cas32_strong_explicit

#define fence std::atomic_thread_fence

/* Global variables */
uint32_t a= 0, b= 0, c= 0, d= 0, result= 0, ready= 0, stage= 0;
char *str= NULL;

/* Thread variables */
uint32_t v1, v2, o;
```



# The problem

## Code

```
a= 1;  
v1= b;  
c= 2;  
v2= d;
```



# The problem

## Code

```
a= 1;  
v1= b;  
c= 2;  
v2= d;
```

## Compiler

```
v2= d;  
v1= b;  
a= 1;  
c= 2;
```



# The problem

## Code

```
a= 1;  
v1= b;  
c= 2;  
v2= d;
```

## Compiler

```
v2= d;  
v1= b;  
a= 1;  
c= 2;
```

## CPU

```
v2= d;  
c= 2;  
a= 1;  
v1= b;
```



# The Problem

Thread 1

```
result= 42;  
ready= 1;
```

Thread 2

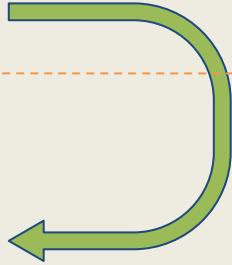
```
while (ready != 1);  
assert(result == 42);
```



# The Problem

## Thread 1

```
result= 42;  
ready= 1;
```



Re-ordered by  
compiler or CPU

## Thread 2

```
while (ready != 1);  
assert(result == 42);
```





# The Problem

## Thread 1

```
ready= 1;
```

```
result= 42;
```

## Thread 2

```
while (ready != 1);  
assert(result == 42);
```



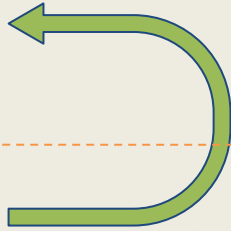
# The Problem

## Thread 1

```
result= 42;  
ready= 1;
```

## Thread 2

```
while (ready != 1);  
assert(result == 42);
```



Re-ordered by  
compiler or CPU



# The Problem

## Thread 1

```
result= 42;  
ready= 1;
```

## Thread 2

```
assert(result == 42);
```

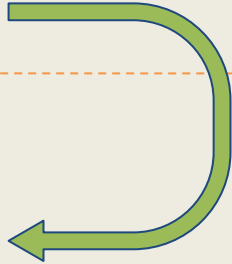
```
while (ready != 1);
```



# The Problem

## Thread 1

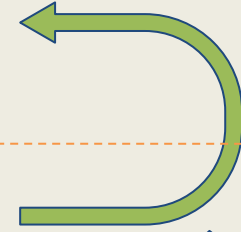
```
result= 42;  
ready= 1;
```



Re-ordered by  
compiler or CPU

## Thread 2

```
while (ready != 1);  
assert(result == 42);
```



Re-ordered by  
compiler or CPU



# The Problem

## Thread 1

```
ready= 1;  
-----  
result= 42;
```

## Thread 2

```
assert(result == 42);  
-----  
while (ready != 1);
```



# Rationale

Memory barriers (jointly with atomic operations) are intended to make data changes visible in concurrent threads.



# API

Memory barrier can be issued along with atomic op

```
my_atomic_store32_explicit(&a, 0, MY_MEMORY_ORDER_RELAXED);
```

or on its own (not available in MariaDB API)

```
std::atomic_thread_fence(std::memory_order_relaxed);
```

Note: thread fence is not supposed to be used alone, it must be accompanied by appropriate atomic operation.



# Memory barriers

- relaxed
- consume
- acquire
- release
- acquire\_release
- sequentially consistent (default)





# Default memory order

```
#define my_atomic_load32(a)
    my_atomic_load32_explicit(a, MY_MEMORY_ORDER_SEQ_CST)

#define my_atomic_store32(a, b)
    my_atomic_store32_explicit(a, b, MY_MEMORY_ORDER_SEQ_CST)

#define my_atomic_fas32(a, b)
    my_atomic_fas32_explicit(a, b, MY_MEMORY_ORDER_SEQ_CST)

#define my_atomic_add32(a, b)
    my_atomic_add32_explicit(a, b, MY_MEMORY_ORDER_SEQ_CST)

#define my_atomic_cas32(a, b, c)
    my_atomic_cas32_strong_explicit(a, b, c, MY_MEMORY_ORDER_SEQ_CST,
                                     MY_MEMORY_ORDER_SEQ_CST)
```



# Memory barriers by strength

1. sequentially consistent

2. acquire\_release

3.1 acquire

3. release

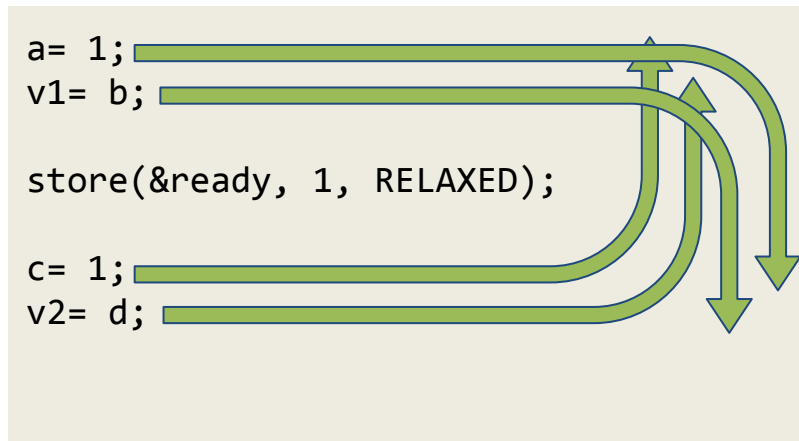
3.2 consume

4. relaxed



# Relaxed memory order

## Relaxed barrier



Atomic operation with **Relaxed** memory barrier guarantees atomicity, but doesn't impose any synchronization or ordering constraints on other loads or stores.



# Relaxed memory order

Valid with any atomic operation

```
b= load(&a, RELAXED);  
store(&a, 1, RELAXED);  
b= fas(&a, 1, RELAXED);  
b= add(&a, 1, RELAXED);  
b= cas(&a, &o, 1, RELAXED, RELAXED);  
  
fence(RELAXED); // no-op
```



# Relaxed memory order

## Example

```
thd->query_id= my_atomic_add64_explicit(&global_query_id, 1,  
                                         MY_MEMORY_ORDER_RELAXED);
```

## Example

```
while (load(&a, RELAXED) != 1);  
fence(ACQUIRE);
```

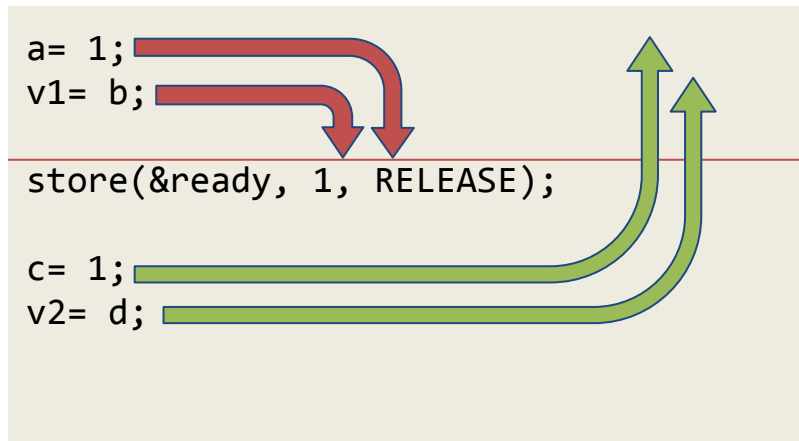
## Example

```
cas(&a, &o, 1, ACQUIRE, RELAXED);
```



# Release memory order

## Release barrier



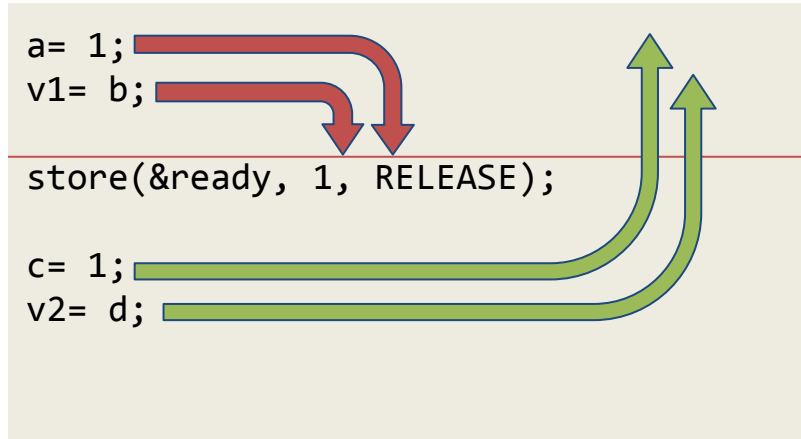
Loads and stores before **Release** can not be reordered after **Release**.

Loads and stores after **Release** can be reordered before **Release**.

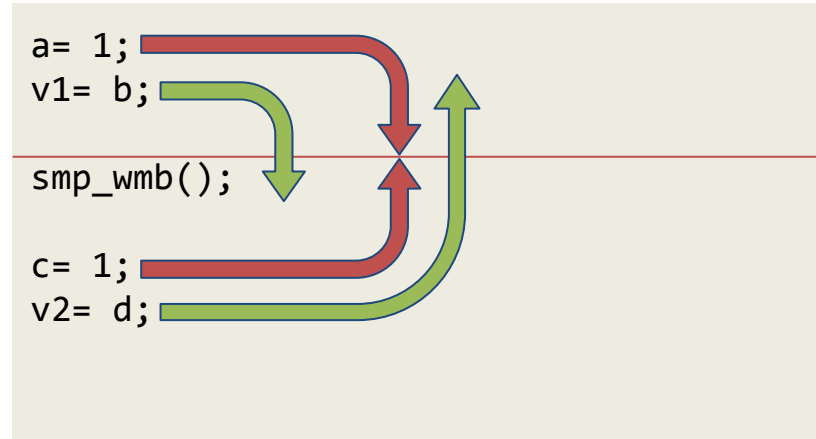


# Release memory order

## Release barrier



## Write barrier



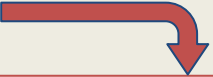
**Not same as write barrier!**



# Release memory order

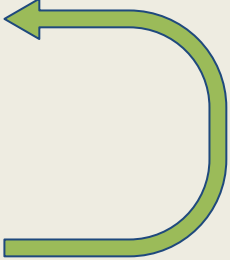
Thread 1

```
result= 42;  
-----  
store(&ready, 1, RELEASE);
```



Thread 2

```
while (ready != 1);  
assert(result == 42);
```



**Meaningless alone!**

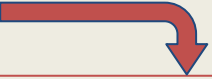




# Release memory order

Thread 1

```
result= 42;  
-----  
store(&ready, 1, RELEASE);
```



Thread 2

```
assert(result == 42);  
  
while (ready != 1);
```

**Meaningless alone!**



# Release memory order

Valid with atomic store or atomic read-modify-write

```
store(&a, 1, RELEASE);  
b= fas(&a, 1, RELEASE);  
b= add(&a, 1, RELEASE);  
b= cas(&a, &o, 1, RELEASE, RELEASE);
```

```
fence(RELEASE); // must be followed by RELAXED atomic store or RMW
```

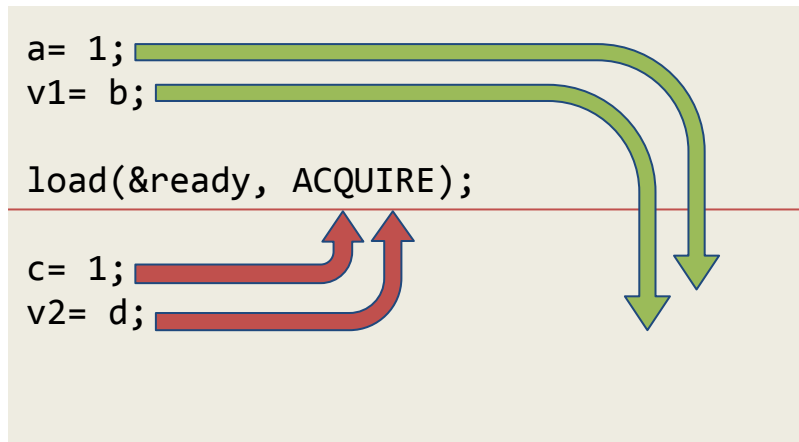
Not valid with atomic load

```
b= load(&a, RELEASE); // undefined, may become RELAXED
```



# Acquire memory order

## Acquire barrier



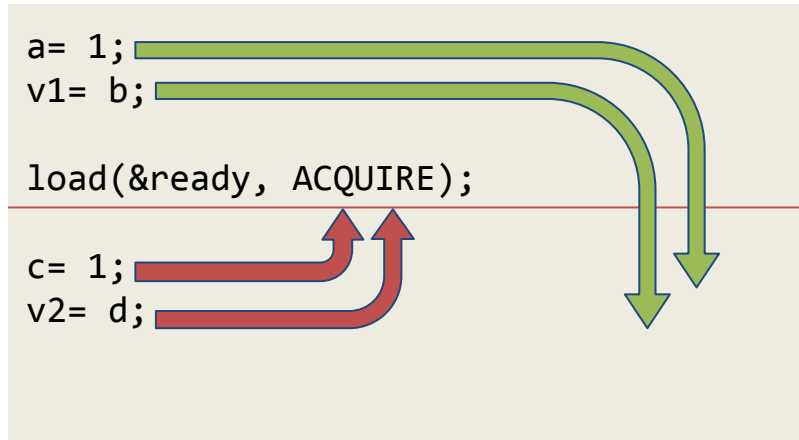
Loads and stores after **Acquire** can not be reordered before **Acquire**.

Loads and stores before **Acquire** can be reordered after **Acquire**.

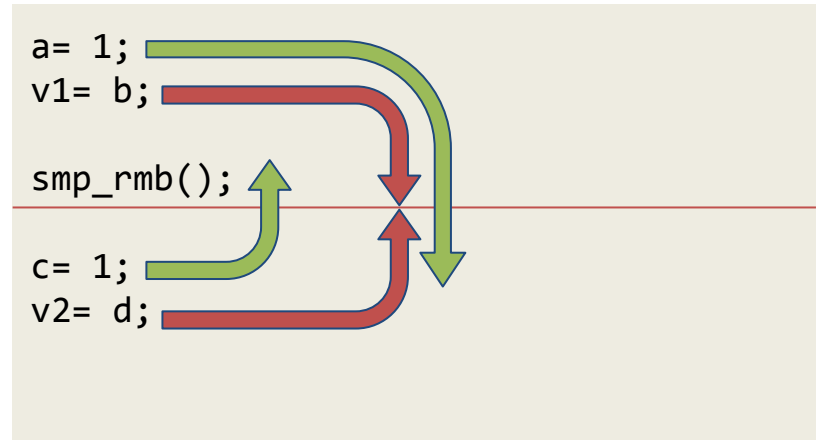


# Acquire memory order

## Acquire barrier



## Read barrier



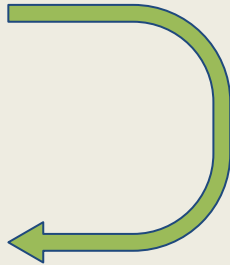
Not same as read barrier!



# Acquire memory order

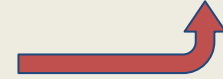
Thread 1

```
result= 42;  
ready= 1;
```



Thread 2

```
while (load(&ready, ACQUIRE) != 1);  
-----  
assert(result == 42);
```



**Meaningless alone!**



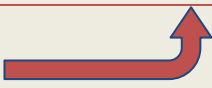
# Acquire memory order

Thread 1

```
ready= 1;  
  
result= 42;
```

Thread 2

```
while (load(&ready, ACQUIRE) != 1);  
-----  
assert(result == 42);
```

A red arrow originates from the `assert(result == 42);` line and points back to the `while (load(&ready, ACQUIRE) != 1);` line, indicating a dependency or a race condition.

## Meaningless alone!



# Acquire memory order

Valid with atomic load or atomic read-modify-write

```
b= load(&a, ACQUIRE);  
b= fas(&a, 1, ACQUIRE);  
b= add(&a, 1, ACQUIRE);  
b= cas(&a, &o, 1, ACQUIRE, ACQUIRE);
```

```
fence(ACQUIRE); // must be preceded by RELAXED atomic load or RMW
```

Not valid with atomic store

```
store(&a, 1, ACQUIRE); // undefined, may become RELAXED
```



# Release-Acquire model

Thread 1

```
result= 42;  
store(&ready, 1, RELEASE);
```

Thread 2

```
while (load(&ready, ACQUIRE) != 1);  
assert(result == 42);
```

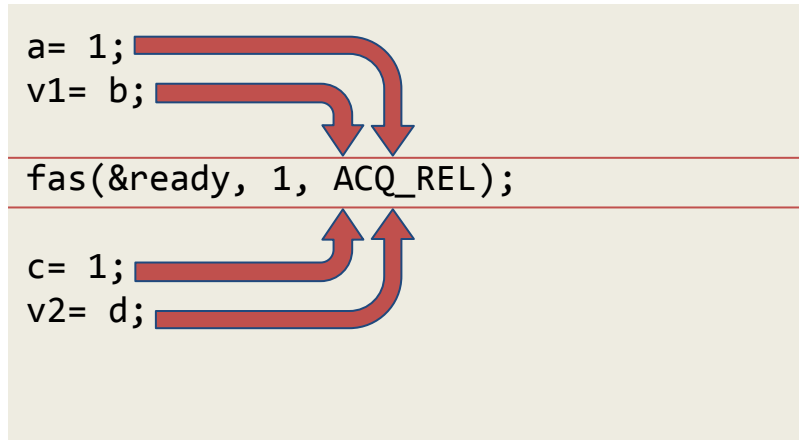
**Acquire** must be always paired with **Release** (or stronger). Only then all stores before **Release** in Thread 1 become visible after **Acquire** in Thread 2.





# Acquire\_release memory order

## Acquire\_release barrier



Loads and stores after **Acquire\_release** can not be reordered before **Acquire\_release**.

Loads and stores before **Acquire\_release** can not be reordered after **Acquire\_release**.



# Acquire\_release memory order

## Valid with atomic read-modify-write

```
b= fas(&a, 1, ACQ_REL);  
b= add(&a, 1, ACQ_REL);  
b= cas(&a, &o, 1, ACQ_REL, ACQ_REL);
```

```
fence(ACQ_REL); // must be preceded by RELAXED atomic load or RMW and  
                // followed by RELAXED atomic store or RMW
```

## Not valid with atomic load and store

```
b= load(&a, ACQ_REL); // undefined, may become ACQUIRE  
store(&a, 1, ACQ_REL); // undefined, may become RELEASE
```



# Acquire\_release memory order

Thread 1

Thread 2

```
a= 1;  
stage= 1;
```

```
while (stage != 2);  
assert(b == 1);
```

```
b= 1;
```

```
while (stage != 1);  
stage= 2;  
assert(a == 1);
```



# Acquire\_release memory order

Thread 1

```
a = 1;  
store(&stage, 1, RELEASE);  
-----  
while (load(&stage, ACQUIRE) != 2);  
-----  
assert(b == 1);
```

Thread 2

```
b = 1;  
-----  
while (load(&stage, ACQUIRE) != 1);  
store(&stage, 2, RELEASE);  
-----  
assert(a == 1);
```



# Acquire\_release memory order

## Thread 1

```
a = 1;  
store(&stage, 1, RELEASE);  
while (load(&stage, ACQUIRE) != 2);  
assert(b == 1);
```

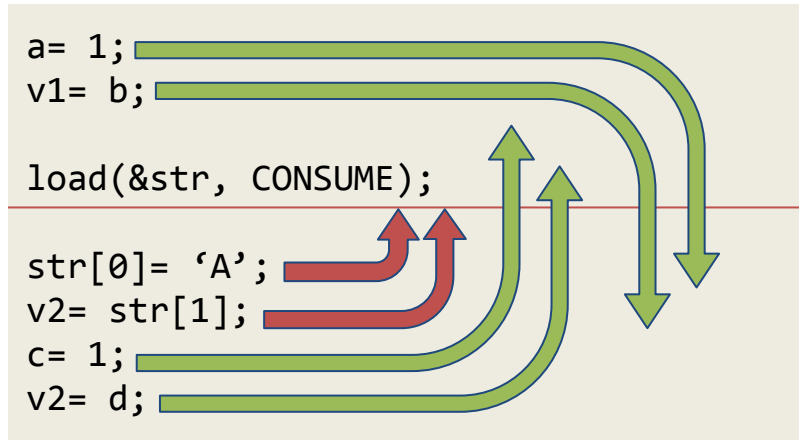
## Thread 2

```
b = 1;  
o = 1;  
while (!cas(&stage, &o, 2, ACQ_REL))  
    o = 1;  
assert(a == 1);
```



# Consume memory order

## Consume barrier



**Consume** is a weaker form of **Acquire**: loads and stores, dependent on the value currently loaded, that happen after **Consume** can not be reordered before **Consume**.



# Consume memory order

Valid with atomic load or atomic read-modify-write

```
b= load(&a, CONSUME);  
b= fas(&a, 1, CONSUME);  
b= add(&a, 1, CONSUME);  
b= cas(&a, &o, 1, CONSUME, CONSUME);
```

```
fence(CONSUME); // must be preceded by RELAXED atomic load or RMW
```

Not valid with atomic store

```
store(&a, 1, CONSUME); // undefined, may become RELAXED
```



# Release-Consume model

Thread 1

```
char *s= strdup("Hello!");  
result= 42;  
-----  
store(&str, s, RELEASE);
```

Thread 2

```
char *s;  
-----  
while (!(s= load(&str, CONSUME)));  
-----  
assert(!strcmp(s, "Hello!"));  
assert(result == 42);
```

**Consume** must be always paired with **Release** (or stronger). Only then all dependent stores before **Release** in Thread 1 become visible after **Consume** in Thread 2.





# Release-Consume model

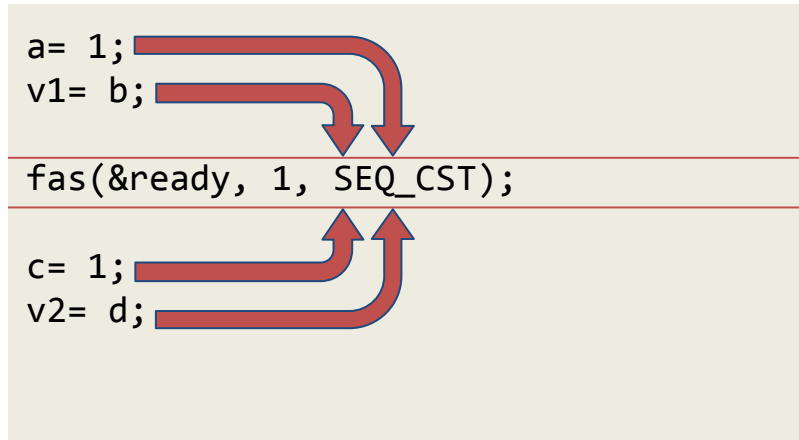
The specification of release-consume ordering is being revised, and the use of `memory_order_consume` is temporarily discouraged.

Note that as of February 2015 no known production compilers track dependency chains: consume operations are lifted to acquire operations.



# Sequentially consistent memory order

## Sequentially consistent



Loads and stores after **Sequentially\_consistent** can not be reordered before **Sequentially\_consistent**.

Loads and stores before **Sequentially\_consistent** can not be reordered after **Sequentially\_consistent**.



# Sequentially consistent memory order

Valid with any atomic operation...

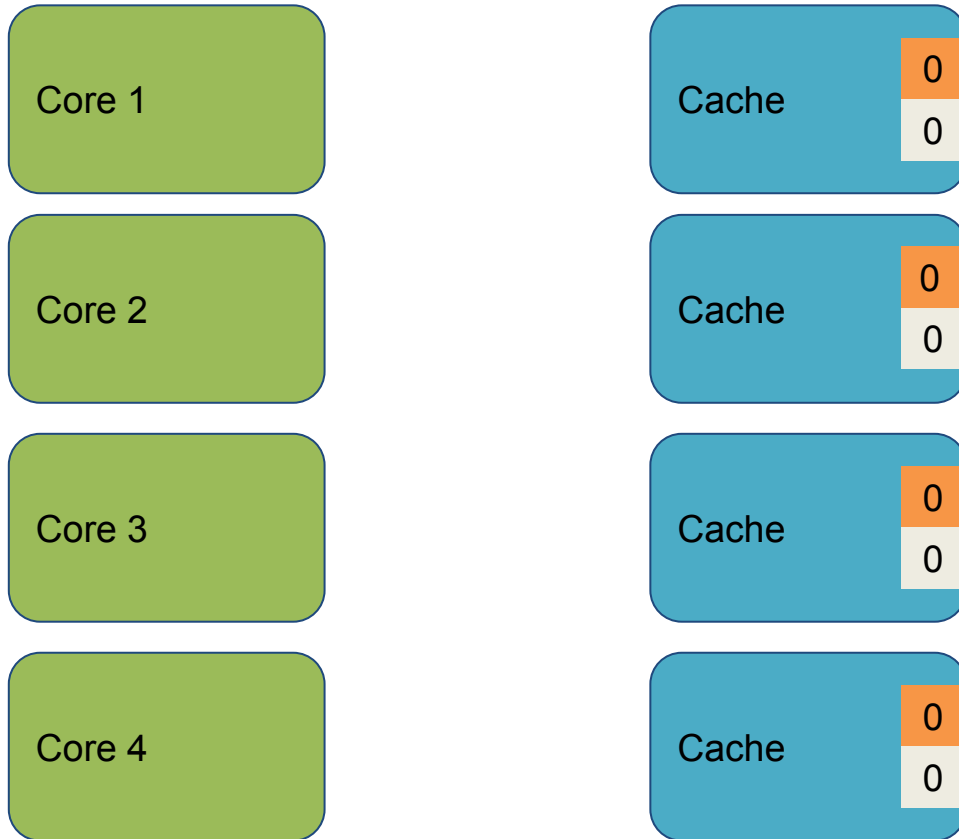
```
b= fas(&a, 1, SEQ_CST);  
b= add(&a, 1, SEQ_CST);  
b= cas(&a, &o, 1, SEQ_CST, SEQ_CST);  
  
fence(SEQ_CST);
```

...but there are traps

```
b= load(&a, SEQ_CST); // may become ACQUIRE + sync  
store(&a, 1, SEQ_CST); // may become RELEASE + sync
```

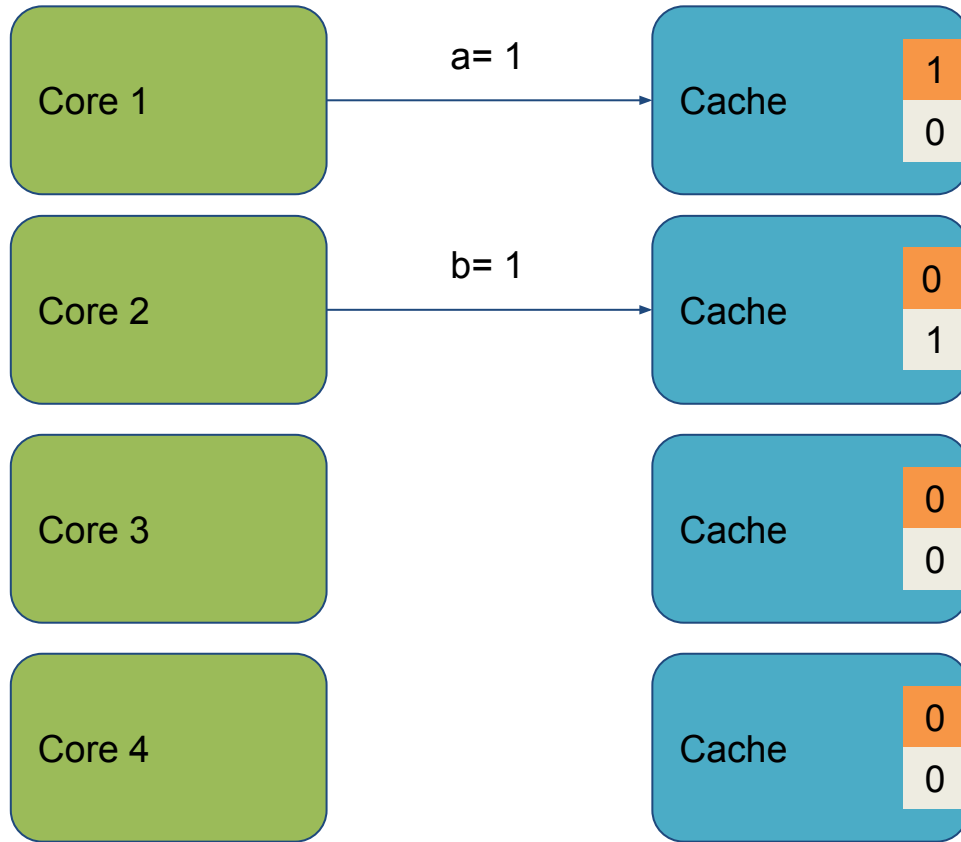


# Cache coherent system



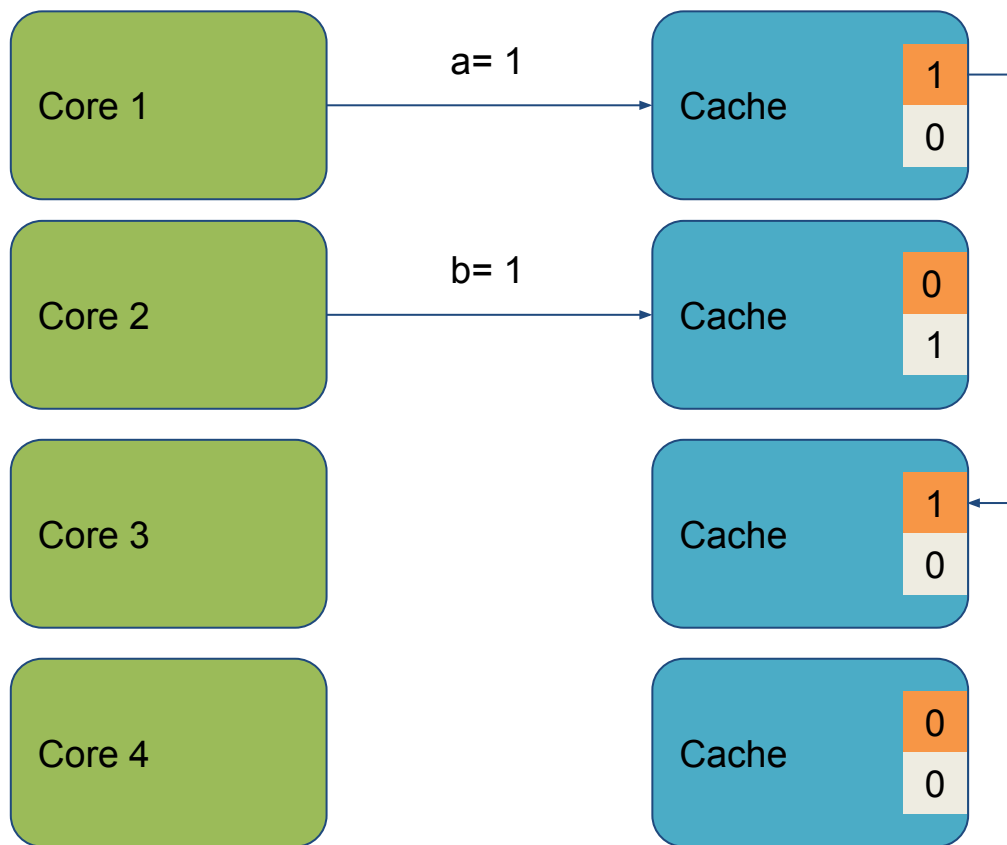


# Cache coherent system



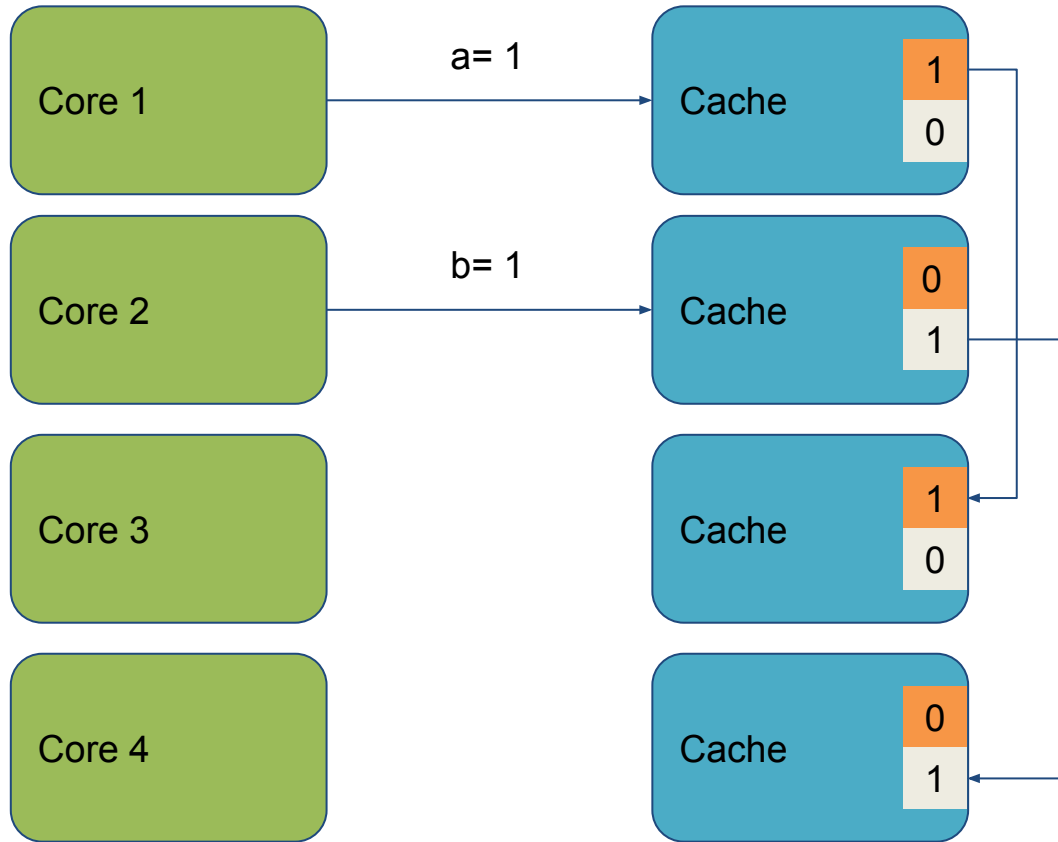


# Cache coherent system



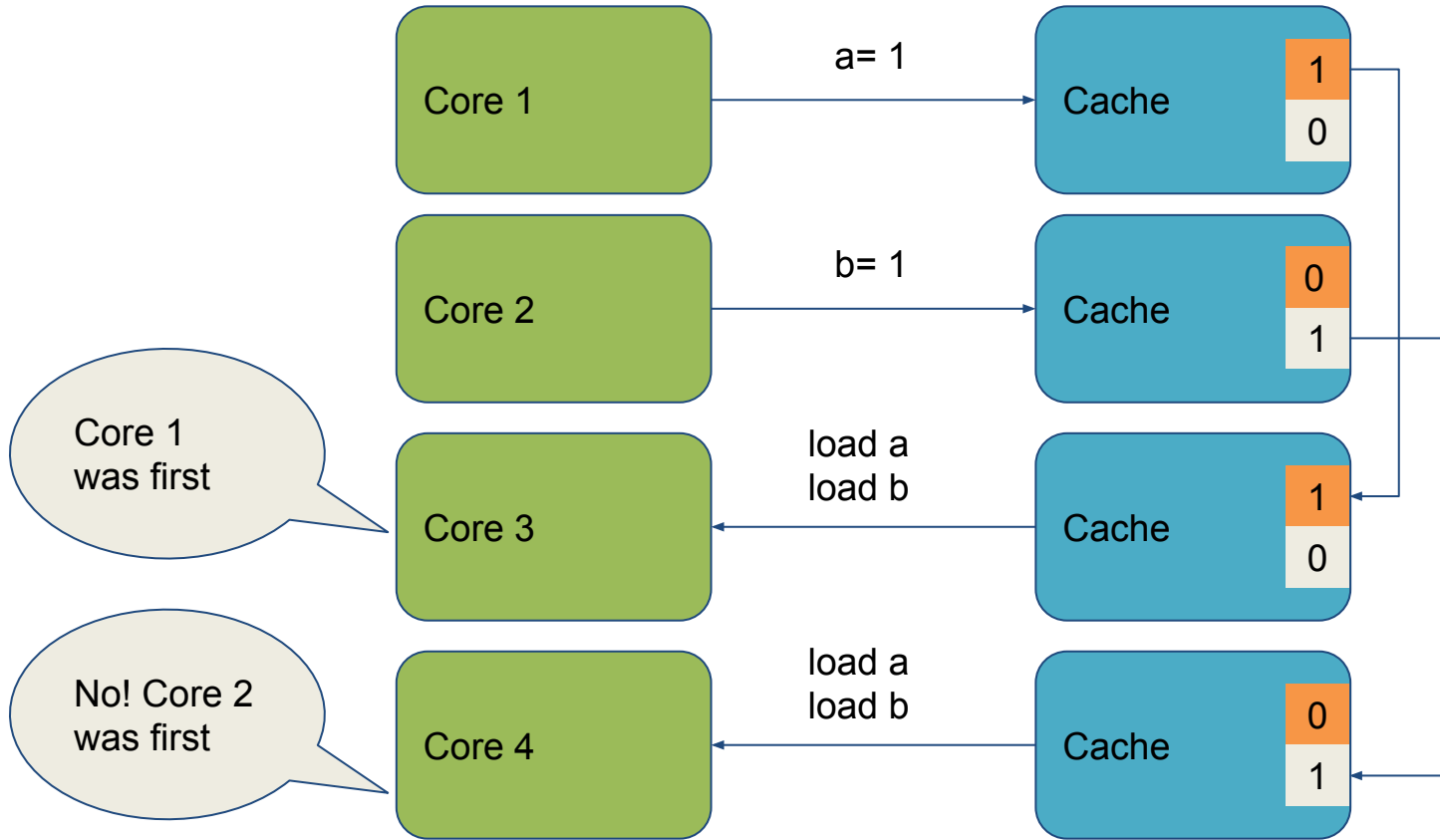


# Cache coherent system





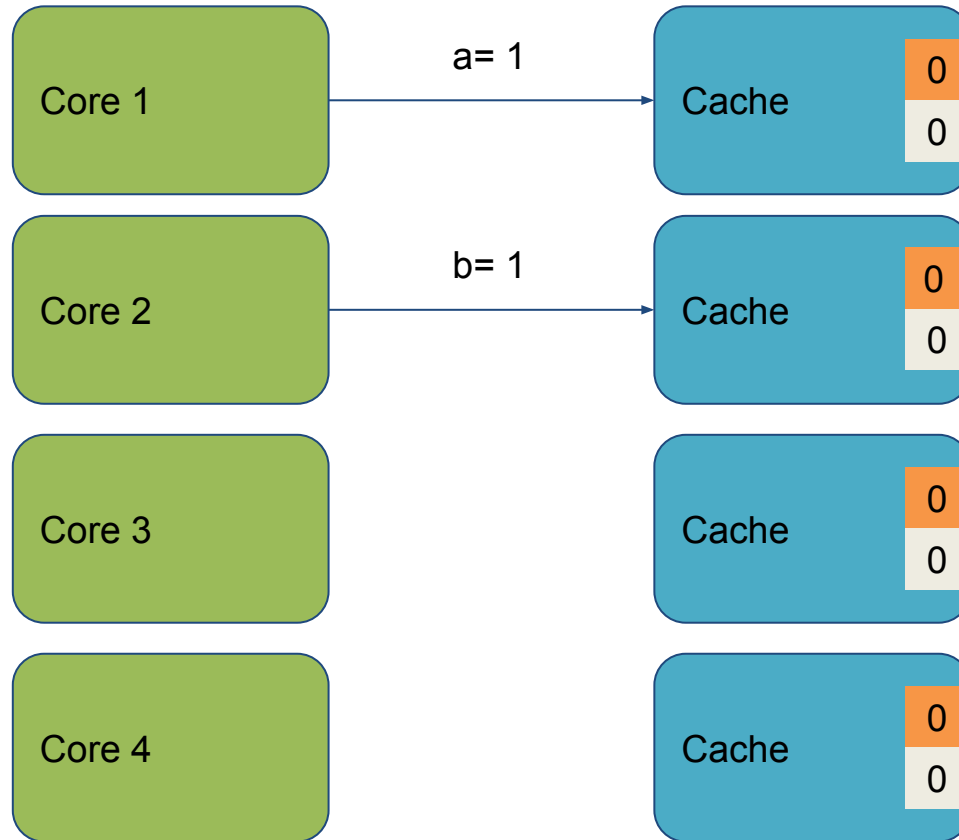
# Cache coherent system





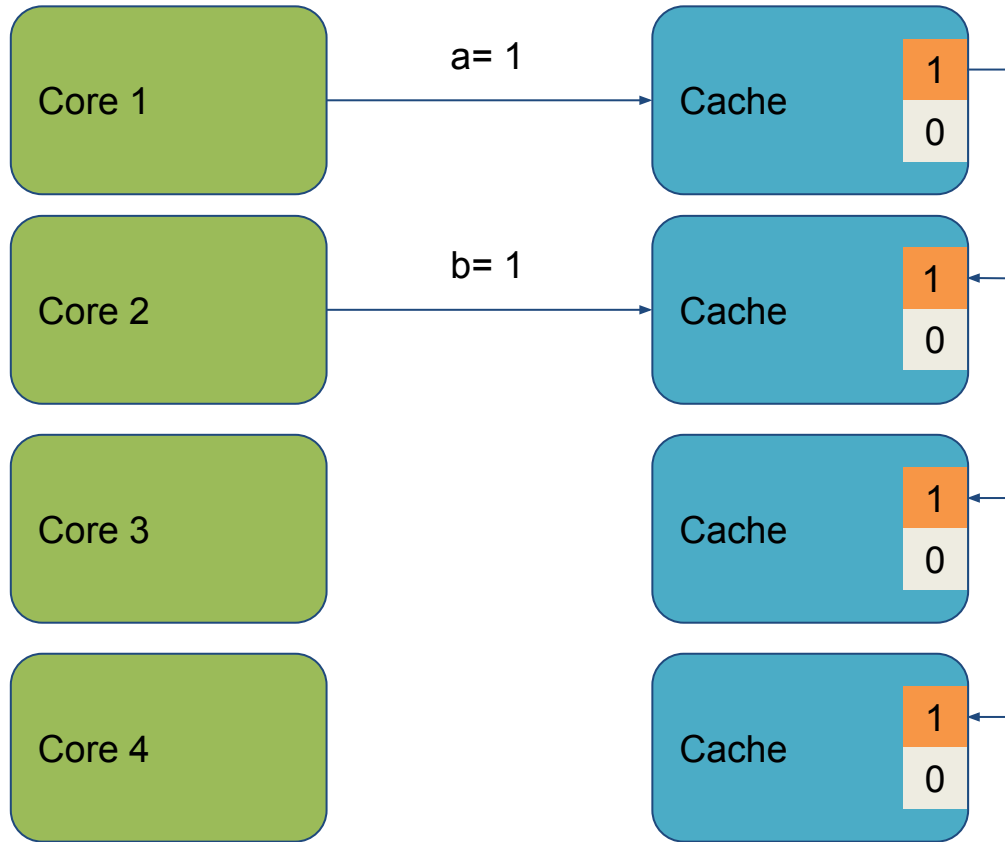


# Sequentially consistent system



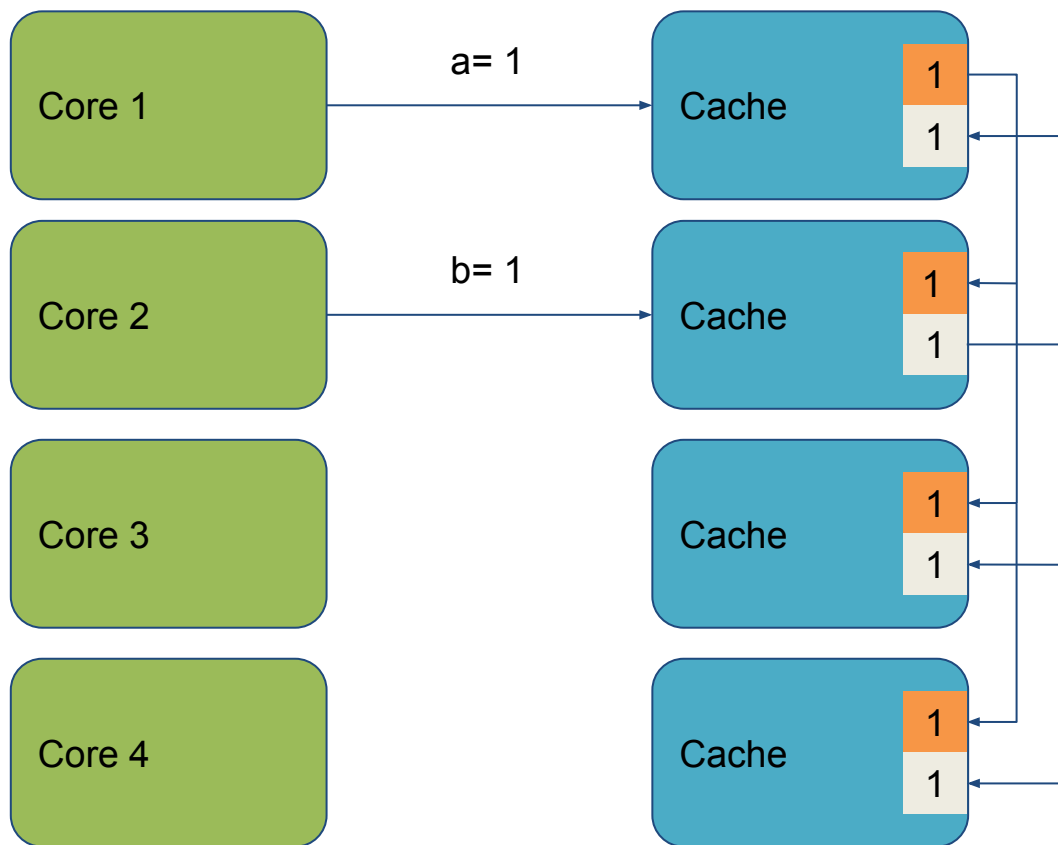


# Sequentially consistent system



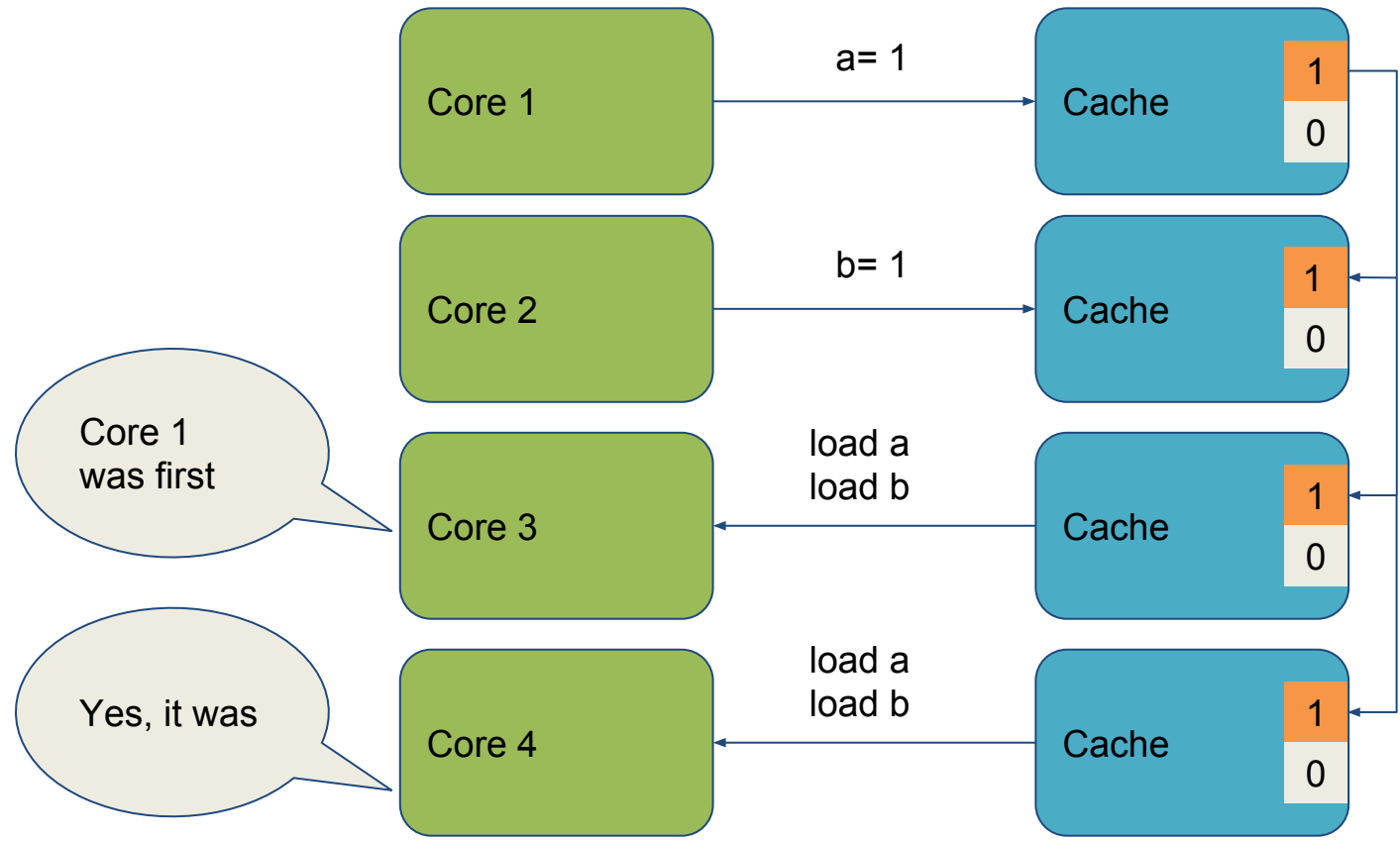


# Sequentially consistent system





# Sequentially consistent system





# Atomic thread fence

- It is possible to issue memory barrier without an associated atomic operation
- it is very advanced technology
- frequently misunderstood
- generally slower than memory barriers associated with an atomic operation



# Atomic thread fence

- non-atomic and **Relaxed** operations cannot be re-ordered after **Release** (first store)
- non-atomic and **Relaxed** operations cannot be re-ordered before **Acquire** (last load)
- still requires atomic operations to work as defined
- not implemented in MariaDB API.



# Atomic thread fence

## Initial state

```
#define fence __atomic_thread_fence

#define RELEASE __ATOMIC_RELEASE
#define ACQUIRE __ATOMIC_ACQUIRE

uint32_t a= 0, b= 0;
```

## Thread 1

```
a= 1;
fence(RELEASE);
b= 1;
```

## Thread 2

```
la= a;
fence(ACQUIRE);
lb= b;

if (lb == 1)
    assert(la == 1); // expectation:
                    // may not fire
```



# Atomic thread fence

## Initial state

```
#define fence __atomic_thread_fence

#define RELEASE __ATOMIC_RELEASE
#define ACQUIRE __ATOMIC_ACQUIRE

uint32_t a= 0, b= 0;
```

## Thread 1

```
a= 1;
b= 1;
```

## Thread 2

```
la= a;
fence(ACQUIRE);
lb= b;

if (lb == 1)
    assert(la == 1); // expectation:
                    // may not fire
```





# Atomic thread fence

## Initial state

```
#define fence __atomic_thread_fence

#define RELEASE __ATOMIC_RELEASE
#define ACQUIRE __ATOMIC_ACQUIRE

uint32_t a= 0, b= 0;
```

## Thread 1

```
a= 1;
fence(RELEASE);
b= 1;
```

## Thread 2

```
la= a;
lb= b;

if (lb == 1)
    assert(la == 1); // expectation:
                    // may not fire
```



# Atomic thread fence

## Initial state

```
#define fence __atomic_thread_fence

#define RELEASE __ATOMIC_RELEASE
#define ACQUIRE __ATOMIC_ACQUIRE

uint32_t a= 0, b= 0;
```

MAY

## Thread 1

```
a= 1;
fence(RELEASE);
b= 1;
```

## Thread 2

```
la= a;
fence(ACQUIRE);
lb= b;

if (lb == 1)
    assert(la == 1); // expectation:
                    // may not fire
```

FIRE



# Atomic thread fence

Possible synchronizations:

- Fence-Atomic
- Atomic-Fence
- Fence-Fence



# Fence-Atomic synchronization

A release fence F in thread A synchronizes-with atomic acquire operation Y in thread B, if...

## Thread A

```
fence(RELEASE); // F
```

## Thread B

```
load(&a, ACQUIRE); // Y
```



# Fence-Atomic synchronization

- there exists an atomic store X (any memory order)
- Y reads the value written by X
- F is sequenced-before X in thread A

## Thread A

```
fence(RELEASE); // F  
store(&a, 1, RELAXED); // X
```

## Thread B

```
load(&a, ACQUIRE); // Y
```



# Fence-Atomic synchronization

In this case, all non-atomic and relaxed atomic stores that happen-before X in thread A will be synchronized-with all non-atomic and relaxed atomic loads from the same locations made in thread B after F.

## Thread A

```
b= 1;  
fence(RELEASE); // F  
store(&a, 1, RELAXED); // X
```

## Thread B

```
if (load(&a, ACQUIRE) == 1) // Y  
    assert(b == 1); // never fires
```



# Atomic-Fence synchronization

An atomic release operation X in thread A  
synchronizes-with an acquire fence F in thread B, if ...

## Thread A

```
store(&a, 1, RELEASE); // X
```

## Thread B

```
fence(ACQUIRE); // F
```



# Atomic-Fence synchronization

- there exists an atomic read Y (any memory order)
- Y reads the value written by X
- Y is sequenced-before F in thread B

## Thread A

```
store(&a, 1, RELEASE); // X
```

## Thread B

```
load(&a, RELAXED); // Y  
fence(ACQUIRE); // F
```





# Atomic-Fence synchronization

In this case, all non-atomic and relaxed atomic stores that happen-before X in thread A will be synchronized-with all non-atomic and relaxed atomic loads from the same locations made in thread B after F.

## Thread A

```
b= 1;  
store(&a, 1, RELEASE); // X
```

## Thread B

```
if (load(&a, RELAXED) == 1) { // Y  
    fence(ACQUIRE); // F  
    assert(b == 1); // never fires  
}
```



# Fence-Fence synchronization

A release fence FA in thread A synchronizes-with an acquire fence FB in thread B, if ...

## Thread A

```
fence(RELEASE); // FA
```

## Thread B

```
fence(ACQUIRE); // FB
```



# Fence-Fence synchronization

- there exists an atomic store X (any memory order)
- FA is sequenced-before X in thread A

## Thread A

```
fence(RELEASE); // FA  
store(&a, 1, RELAXED); // X
```

## Thread B

```
fence(ACQUIRE); // FB
```



# Fence-Fence synchronization

- there exists an atomic read Y (any memory order)
- Y reads the value written by X
- Y is sequenced-before FB in thread B

## Thread A

```
fence(RELEASE); // FA
store(&a, 1, RELAXED); // X
```

## Thread B

```
load(&a, RELAXED); // Y
fence(ACQUIRE); // FB
```



# Fence-Fence synchronization

In this case, all non-atomic and relaxed atomic stores that happen-before FA in thread A will be synchronized-with all non-atomic and relaxed atomic loads from the same locations made in thread B after FB.

## Thread A

```
b= 1;  
fence(RELEASE); // FA  
store(&a, 1, RELAXED); // X
```

## Thread B

```
if (load(&a, RELAXED) == 1) { // Y  
    fence(ACQUIRE); // FB  
    assert(b == 1); // never fires }
```



# Fence-Fence synchronization

## Example

```
char *data[10];

void producer()
{
    for (int i= 0; i < 10; i++)
        data[i]= strdup("some long string");
}

void consumer()
{
    for (int i= 0; i < 10; i++)
        puts(data[i]);
}
```



# Fence-Fence synchronization

## Example

```
char *data[10];
uint32_t ready[10]= { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

void producer()
{
    for (int i= 0; i < 10; i++)
    {
        data[i]= strdup("some long string");
        my_atomic_store32_explicit(&ready[i], 1, MY_MEMORY_ORDER_RELEASE);
    }
}

void consumer()
{
    for (int i= 0; i < 10; i++)
    {
        if (my_atomic_load32_explicit(&ready[i], MY_MEMORY_ORDER_ACQUIRE) == 1)
            puts(data[i]);
    }
}
```



# Fence-Fence synchronization

## Example

```
char *data[10];
uint32_t ready[10]= { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

void producer()
{
    for (int i= 0; i < 10; i++)
        data[i]= strdup("some long string");
    fence(MY_MEMORY_ORDER_RELEASE);
    for (int i= 0; i < 10; i++)
        my_atomic_store32_explicit(&ready[i], 1, MY_MEMORY_ORDER_RELAXED);
}

void consumer()
{
    for (int i= 0; i < 10; i++)
    {
        if (my_atomic_load32_explicit(&ready[i], MY_MEMORY_ORDER_ACQUIRE) == 1)
            puts(data[i]);
    }
}
```





# Fence-Fence synchronization

## Example

```
char *data[10];
uint32_t ready[10]= { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

void producer()
{
    for (int i= 0; i < 10; i++)
    {
        data[i]= strdup("some long string");
        my_atomic_store32_explicit(&ready[i], 1, MY_MEMORY_ORDER_RELEASE);
    }
}

void consumer()
{
    uint32_t tmp[10];
    for (int i= 0; i < 10; i++)
        tmp[i]= my_atomic_load32_explicit(&ready[i], MY_MEMORY_ORDER_RELAXED);
    fence(MY_MEMORY_ORDER_ACQUIRE);
    for (int i= 0; i < 10; i++)
        if (tmp[i] == 1)
            puts(data[i]);
}
```



# Fence-Fence synchronization

## Example

```
char *data[10];
uint32_t ready[10]= { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

void producer()
{
    for (int i= 0; i < 10; i++)
        data[i]= strdup("some long string");
    fence(MY_MEMORY_ORDER_RELEASE);
    for (int i= 0; i < 10; i++)
        my_atomic_store32_explicit(&ready[i], 1, MY_MEMORY_ORDER_RELAXED);
}

void consumer()
{
    uint32_t tmp[10];
    for (int i= 0; i < 10; i++)
        tmp[i]= my_atomic_load32_explicit(&ready[i], MY_MEMORY_ORDER_RELAXED);
    fence(MY_MEMORY_ORDER_ACQUIRE);
    for (int i= 0; i < 10; i++)
        if (tmp[i] == 1)
            puts(data[i]);
}
```



# References

[http://en.cppreference.com/w/cpp/atomic/memory\\_order](http://en.cppreference.com/w/cpp/atomic/memory_order)

[https://en.wikipedia.org/wiki/Memory\\_ordering](https://en.wikipedia.org/wiki/Memory_ordering)

[http://preshing.com/20140709/the-purpose-of-memory\\_order\\_consume-in-cpp11/](http://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/)