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

```plaintext
a = 1;
v1 = b;
c = 2;
v2 = d;
```
The problem

<table>
<thead>
<tr>
<th>Code</th>
<th>Compiler</th>
</tr>
</thead>
</table>
| a = 1;  
v1 = b;  
c = 2;  
v2 = d; | v2 = d;  
v1 = b;  
a = 1;  
c = 2; |
## The problem

<table>
<thead>
<tr>
<th>Code</th>
<th>Compiler</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a= 1;</code></td>
<td><code>v2= d;</code></td>
<td><code>v2= d;</code></td>
</tr>
<tr>
<td><code>v1= b;</code></td>
<td><code>v1= b;</code></td>
<td><code>c= 2;</code></td>
</tr>
<tr>
<td><code>c= 2;</code></td>
<td><code>a= 1;</code></td>
<td><code>a= 1;</code></td>
</tr>
<tr>
<td><code>v2= d;</code></td>
<td><code>c= 2;</code></td>
<td><code>v1= b;</code></td>
</tr>
</tbody>
</table>
The Problem

Thread 1

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

Thread 2

```c
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

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

Thread 2

```c
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

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = 42; ready = 1;</td>
<td>assert(result == 42);</td>
</tr>
<tr>
<td>while (ready != 1);</td>
<td>while (ready != 1);</td>
</tr>
</tbody>
</table>
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

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

### Thread 2

```c
assert(result == 42);
while (ready != 1);
```
Memory barriers (jointly with atomic operations) are intended to make data changes visible in concurrent threads.
Memory barrier can be issued along with atomic op

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

or on its own (not available in MariaDB API)

```c
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

```c
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

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

Example

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

Example

```c
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!
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

```c
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

```c
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

a = 1;
v1 = b;
load(&ready, ACQUIRE);
c = 1;
v2 = d;

Read barrier

a = 1;
v1 = b;
smp_rmb();
c = 1;
v2 = d;

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);

Meaningless alone!
Acquire memory order

Valid with atomic load or atomic read-modify-write

```c
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

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

a= 1;
v1= b;
fas(&ready, 1, ACQ_REL);
c= 1;
v2= d;

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

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

```c
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

```c
b = load(&a, ACQ_REL); /* undefined, may become ACQUIRE */
store(&a, 1, ACQ_REL); /* undefined, may become RELEASE */
```
Acquire_release memory order

Thread 1

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

Thread 2

```c
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) != 1);
assert(a == 1);

Thread 2

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

b = 1;
while (load(&stage, ACQUIRE) != 2);
assert(b == 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 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

```c
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

```c
store(&a, 1, CONSUME);` // undefined, may become RELAXED
```
**Release-Consume model**

**Thread 1**

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

**Thread 2**

```c
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-Consumption 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

```
a = 1;
v1 = b;
fas(&ready, 1, SEQ_CST);
c = 1;
v2 = d;
```

Loads and stores after `Sequentially_consistent` cannot be reordered before `Sequentially_consistent`.

Loads and stores before `Sequentially_consistent` cannot be reordered after `Sequentially_consistent`. 
Sequentially consistent memory order

Valid with any atomic operation...

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

```c
fence(SEQ_CST);
```

...but there are traps

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

Core 1

Core 2

Core 3

Core 4

Cache

Cache

Cache

Cache

0

0

0

0
Cache coherent system

Core 1 → Cache
    a = 1

Core 2 → Cache
    b = 1

Core 3 → Cache

Core 4 → Cache
Cache coherent system

Core 1

Core 2

Core 3

Core 4

a = 1

b = 1

Cache

Cache

Cache

Cache
Cache coherent system

Core 1 → a = 1 → Cache

Core 2 → b = 1 → Cache

Core 3 → Cache

Core 4 → Cache
Cache coherent system

Core 1
Core 2
Core 3
Core 4

Core 1 was first
No! Core 2 was first
Sequentially consistent system

Core 1

Core 2

Core 3

Core 4

Cache

Cache

Cache

Cache

$\text{a} = 1$

$\text{b} = 1$
Sequentially consistent system

Core 1
Core 2
Core 3
Core 4

Cache
Cache
Cache
Cache

a = 1
b = 1
Sequentially consistent system

Core 1

Core 2

Core 3

Core 4

a = 1

b = 1
Sequentially consistent system

Core 1 was first

Core 1

Core 2

Core 3

Core 4

a = 1

b = 1

load a

load b

load a

load b

Yes, it was
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

```c
#define fence __atomic_thread_fence
#define RELEASE __ATOMIC_RELEASE
#define ACQUIRE __ATOMIC_ACQUIRE

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

Thread 1

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

Thread 2

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

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

Initial state

```c
#define fence __atomic_thread_fence
#define RELEASE __ATOMIC_RELEASE
#define ACQUIRE __ATOMIC_ACQUIRE

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

Thread 1

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

Thread 2

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

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

Initial state

```c
#define fence __atomic_thread_fence
#define RELEASE __ATOMIC_RELEASE
#define ACQUIRE __ATOMIC_ACQUIRE

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

Thread 1

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

Thread 2

```c
la = a;
lb = b;
if (lb == 1)
    assert(la == 1); // expectation:
    // may not fire
```
Atomic thread fence

Initial state

```c
#define fence __atomic_thread_fence
#define RELEASE __ATOMIC_RELEASE
#define ACQUIRE __ATOMIC_ACQUIRE

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

Thread 1

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

Thread 2

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

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

```
fence(RELEASE); // F
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

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

Thread B

```c
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

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

Thread B

```c
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 $A$

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

Thread $B$

```c
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

\[
\begin{align*}
    \text{b} &= 1; \\
    \text{store}(&a, 1, \text{RELEASE}); & // X
\end{align*}
\]

Thread B

\[
\begin{align*}
    \text{if (load}(&a, \text{RELAXED}) == 1) & { // Y} \\
    \text{fence}(&\text{ACQUIRE}); & // F \\
    \text{assert(b} == 1); & // never fires
\end{align*}
\]
Fence-Fence synchronization

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

Thread A

\[ \text{fence(RELEASE);} \quad // \ FA \]

Thread B

\[ \text{fence(ACQUIRE);} \quad // \ FB \]
Fence-Fence synchronization

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

```c
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

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

Thread B

```c
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

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

Thread B

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

```c
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]);
}
```
Example

```c
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

```c
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

```c
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

```c
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

https://en.wikipedia.org/wiki/Memory_ordering