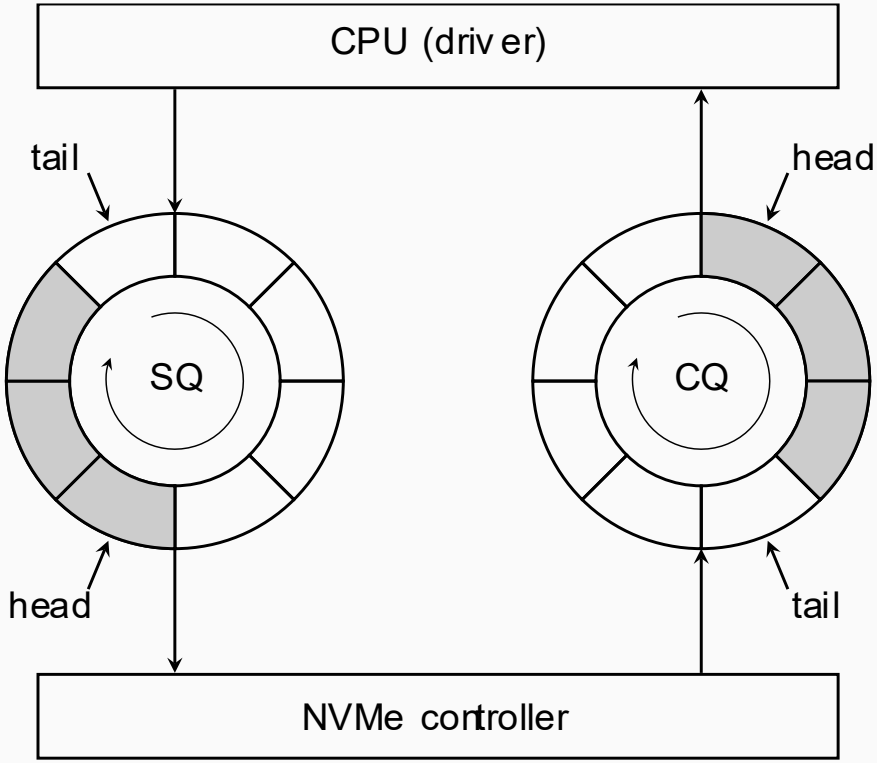
Same Fence, Different Silicon

Memory Fences & Interacting with NVMe Controllers in x86 and ARM

C++ Under the Sea 2026 - Emir Hurturk

The NVMe Model

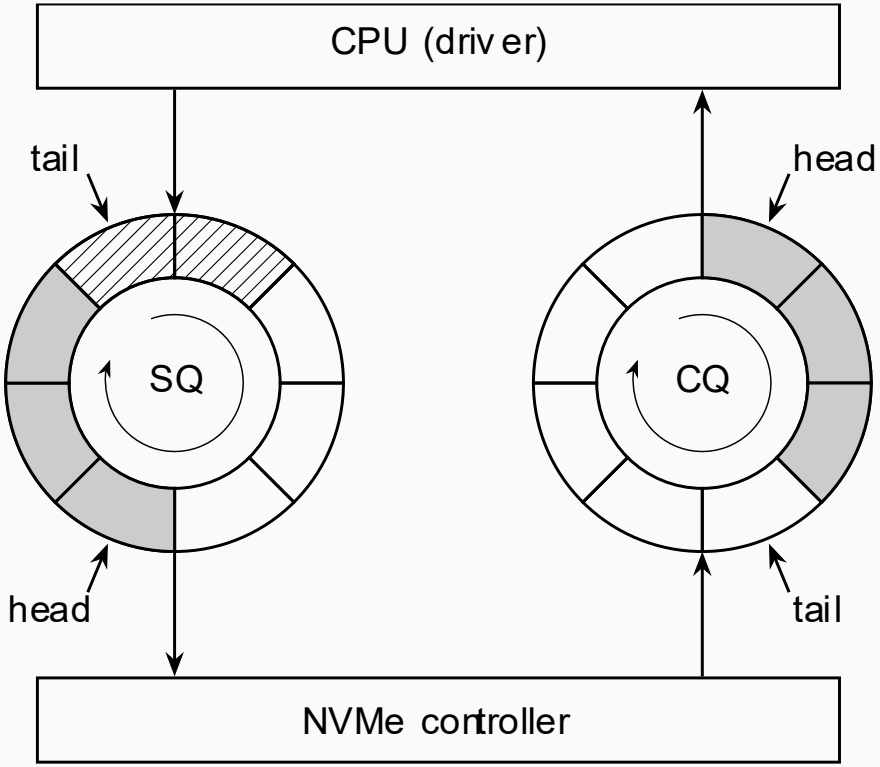


The NVMe Model: Command Submission

Insert 2 Commands to the Submission Queue

```
bool push(const Entry& e) {
    if (!write(e)) {
        return false;
    }

    commit();
    return true;
}
```

```
auto* dst = dma_region_.uva
          + (tail_ * sizeof(Entry));
std::memcpy(dst, &e.data(), sizeof(Entry));
```

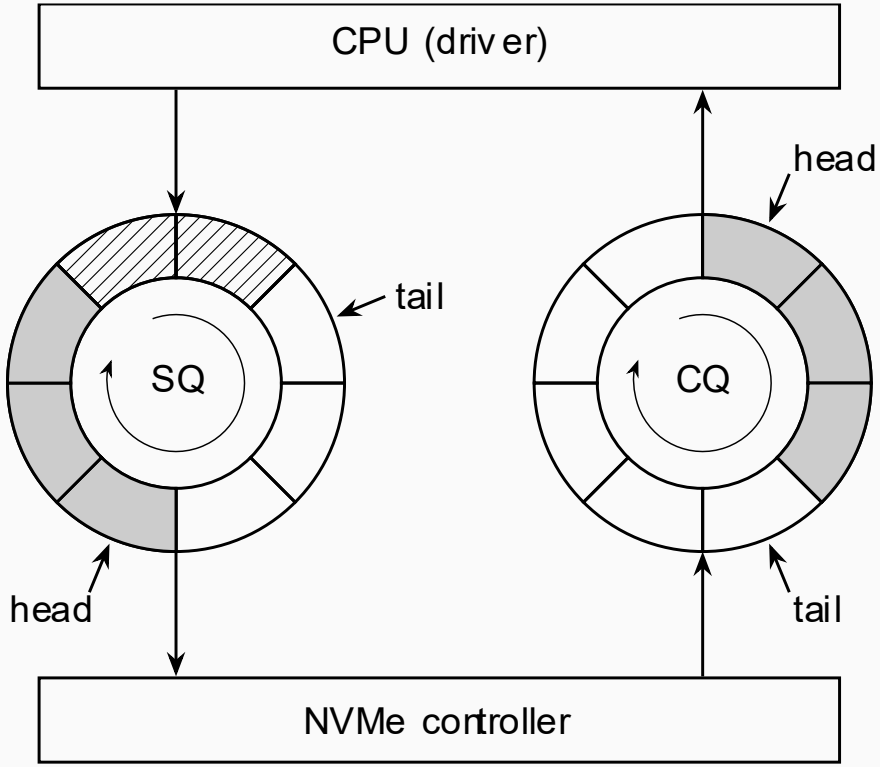


The NVMe Model: Command Submission

```
bool push(const Entry& e) {
    if (!write(e)) {
        return false;
    }

    commit();
    return true;
}
```

```
tail_ = (tail_ + 1) & (depth_ - 1);
```

Update the Tail Index

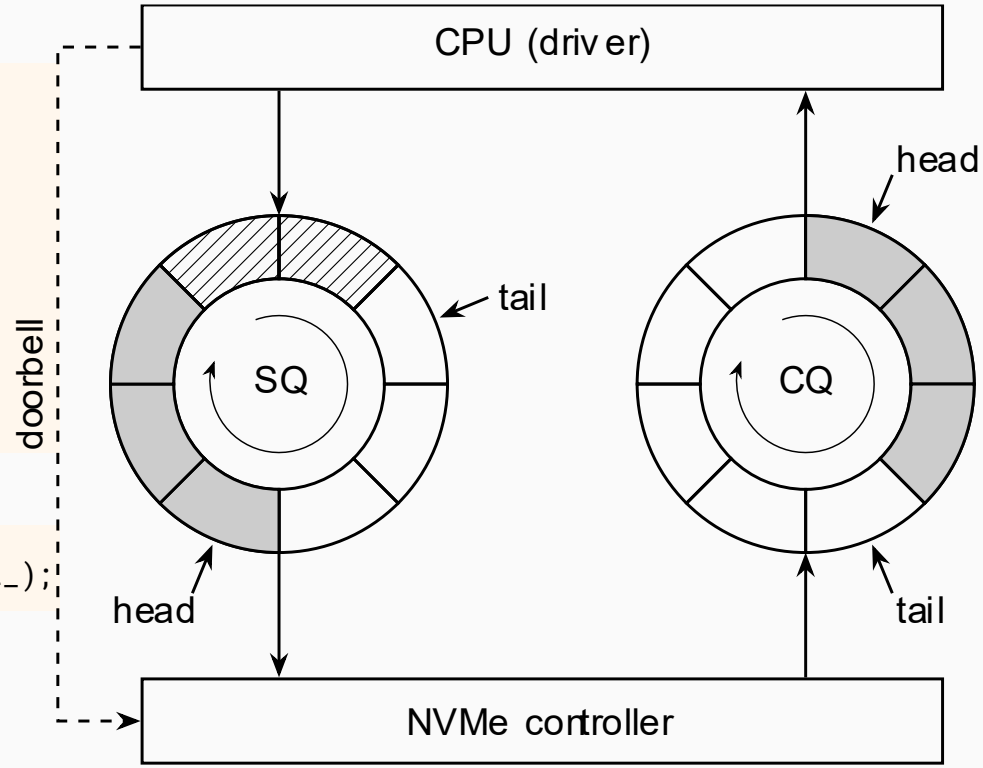


The NVMe Model: Command Submission

Ring the SQ tail doorbell with the updated tail index

```
bool push(const Entry& e) {
    if (!write(e)) {
        return false;
    }

    commit();
    return true;
}
```

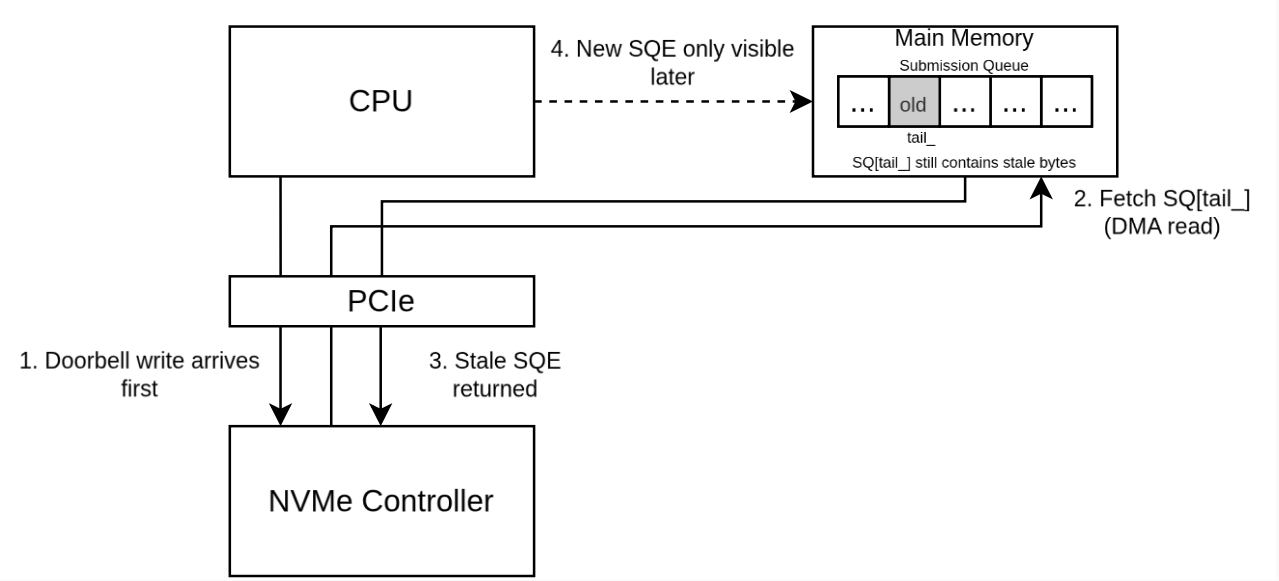
```
*doorbell_ = static_cast<std::uint32_t>(tail_);
```



The NVMe Model: Ordering Matters

1. Insert the SQ Entry
2. Update the Tail Pointer
3. Ring the doorbell

Only after steps 1 and 3 execute **in order,** then the controller should **fetch** the command from provided DMA-able memory.



If the doorbell write becomes visible **before** the SQ entry, the controller DMAs stale bytes and executes a garbage command.

How can we establish the device ordering?

STL Options

There are number of options that the standard library offers for memory synchronization orderings.

Memory synchronization ordering	
`memory_order`	(C++11)
`kill_dependency`	(C++11) (deprecated in C++26)
`atomic_thread_fence`	(C++11)
`atomic_signal_fence`	(C++11)

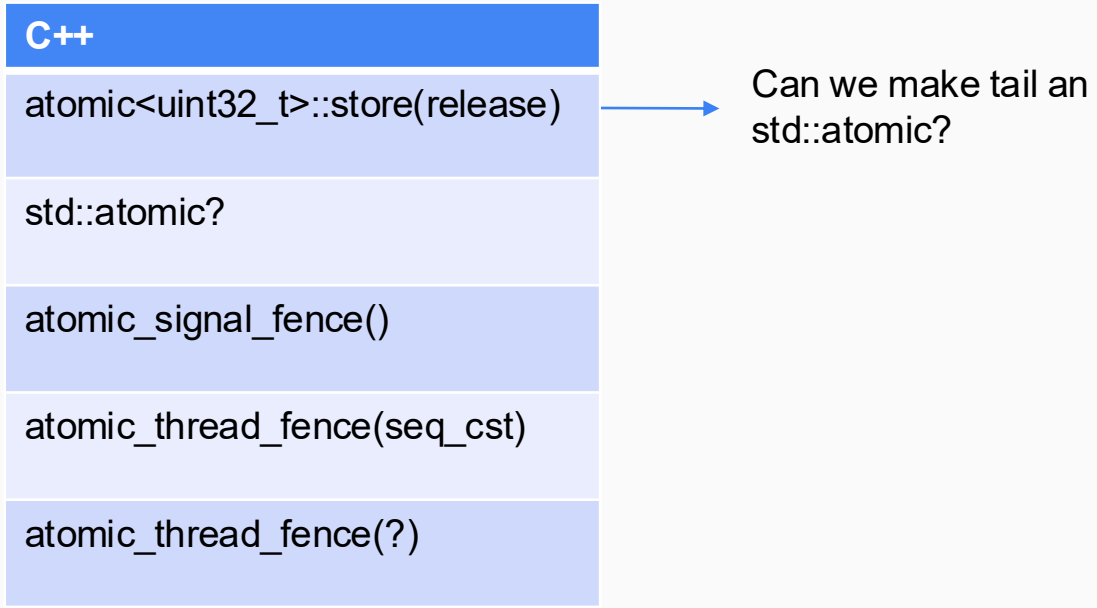
(taken from cppreference.com)

C++
atomic<uint32_t>::store(release)
Make doorbell std::atomic?
atomic_signal_fence()
atomic_thread_fence(seq_cst)
atomic_thread_fence(?)

Release needs a partner

Our other side is a controller, which never performs an acquire load.

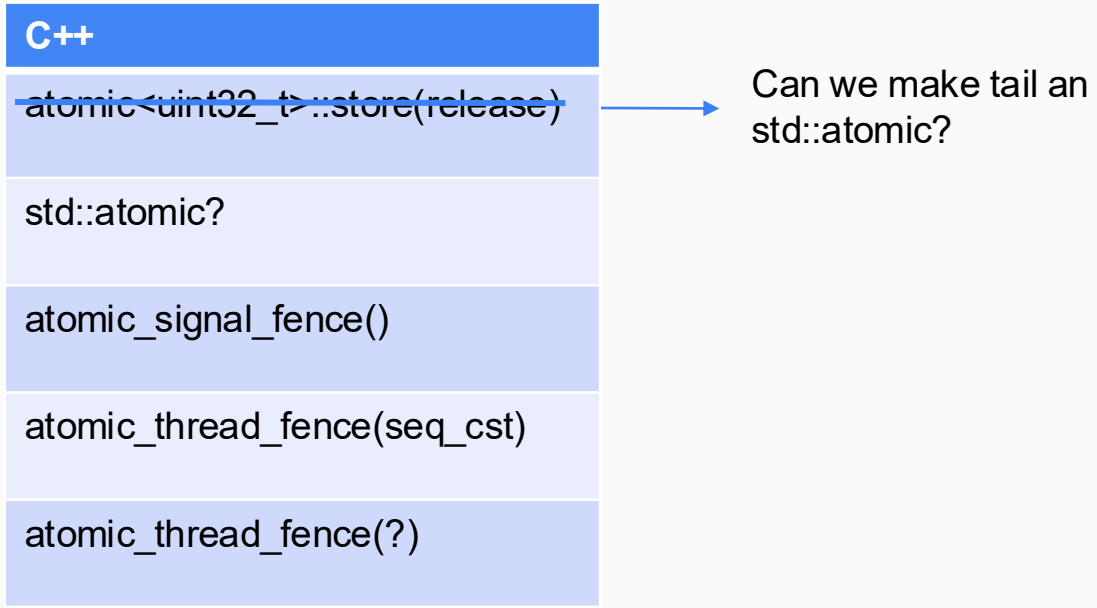
- tail is the wrong variable anyway, the controller only reads the SQE and the doorbell



Release needs a partner

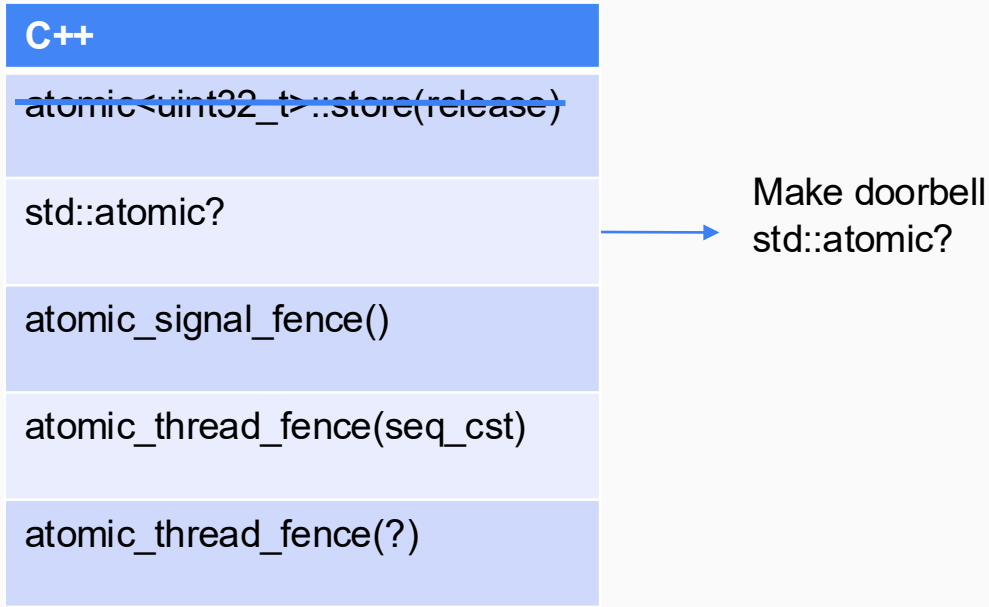
Our other side is a controller, which never performs an acquire load.

- tail is the wrong variable anyway, the controller only reads the SQE and the doorbell



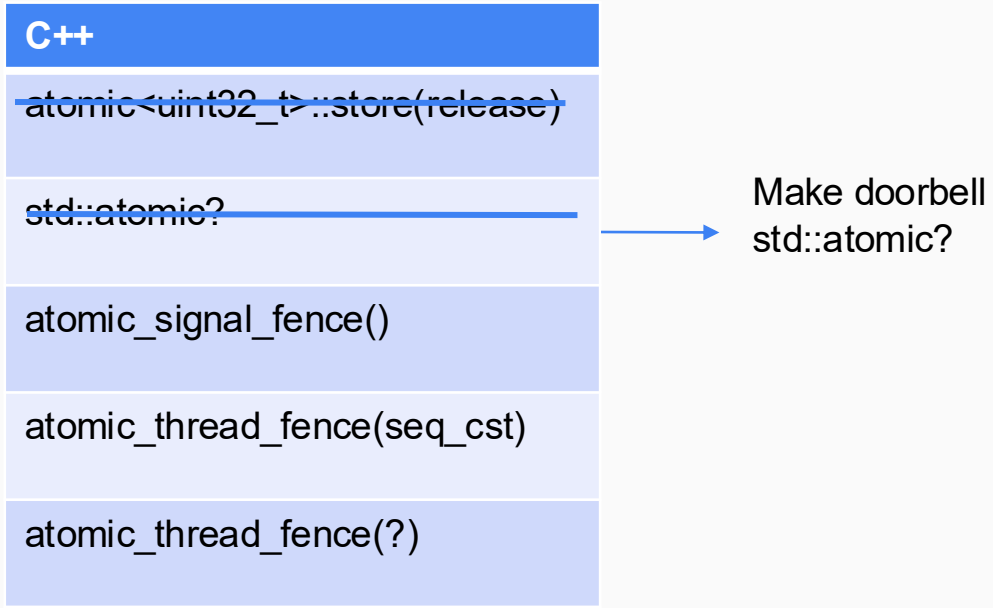
Atomics

Doorbell is a MMIO register, and there is no std::atomic object living in an MMIO address.



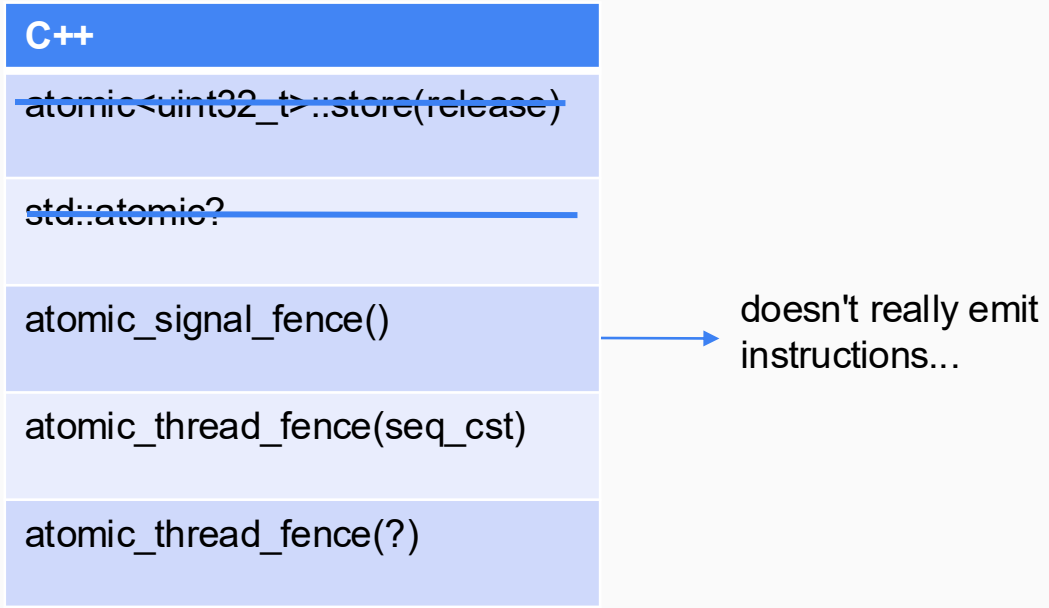
Atomics

Doorbell is a MMIO register, and there is nostd::atomic object living in an MMIO address.



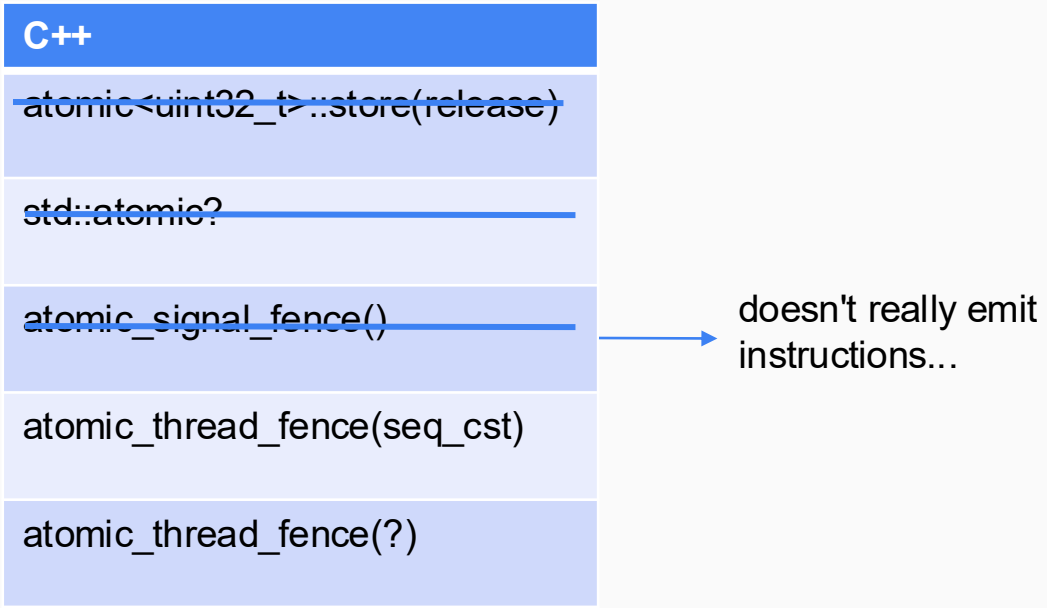
Stops the compiler, not the CPU

For a thread and a signal handler on the same thread, and **doesn't** issue any CPU instructions



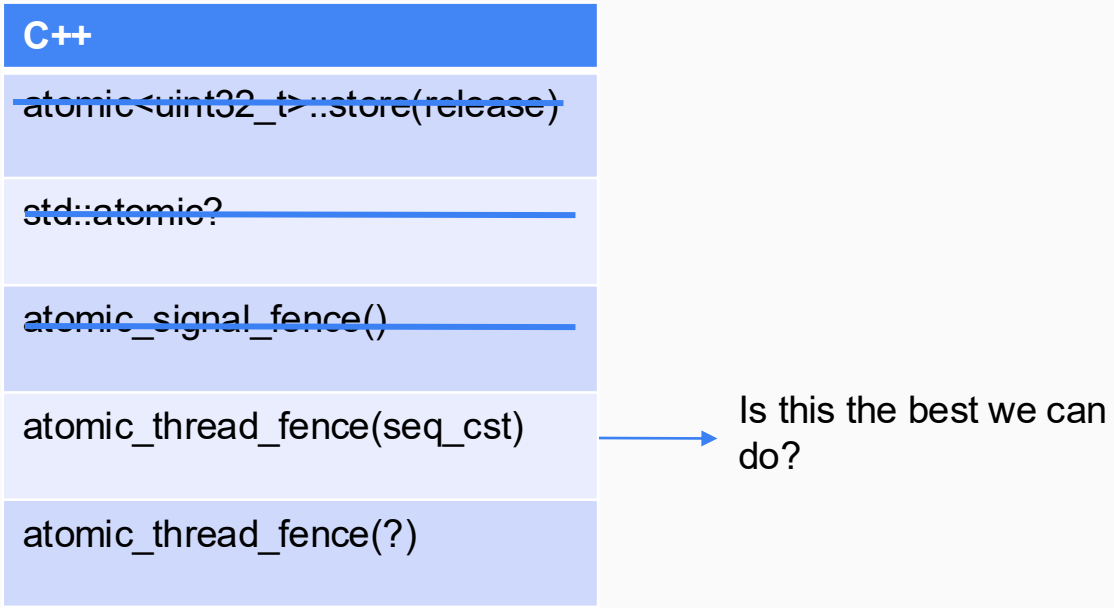
Stops the compiler, not the CPU

For a thread and a signal handler on the same thread, and **doesn't** issue any CPU instructions



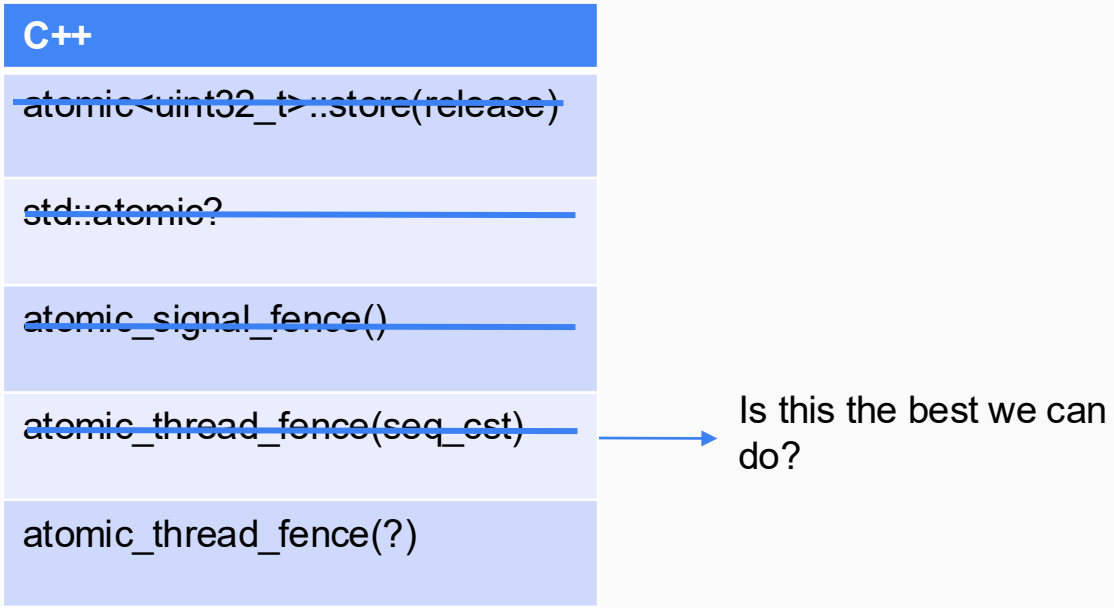
Works…

… but let's also consider weaker orderings



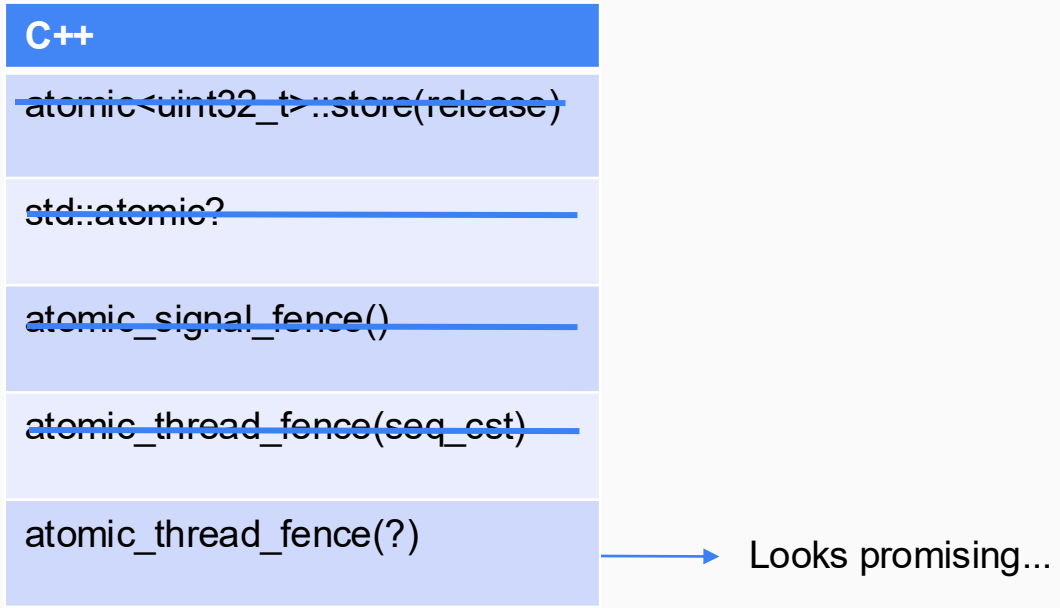
Works…

… but let's also consider weaker orderings



Weaker Ordering

Let's see which ordering is suitable and weaker



std::atomic_thread_fence()

We can utilize the standard library's `std::atomic_thread_fence` to have stronger orderings for the push() events

1. Insert the SQ Entry
2. Update the Tail Pointer

3. Ring the doorbell

```
std::memcpy(dst, &e, sizeof(Entry));     // (A)
tail_ = (tail_ + 1) & (depth_ - 1);      // (B)
std::atomic_thread_fence(...);
*doorbell_ = tail_;                      // (C)
```

Which memory order should we use?

std::atomic_thread_fence()

Sounds like the `std::memory_order_release` memory order for this case

1. Insert the SQ Entry
2. Update the Tail Pointer

Ensure that whenever the controller sees the updated tail, the correct SQ entry is present in memory

3. Ring the doorbell

```
std::memcpy(dst, &e, sizeof(Entry));     // (A)
tail_ = (tail_ + 1) & (depth_ - 1);      // (B)
std::atomic_thread_fence(
    std::memory_order_release);
*doorbell_ = tail_;                      // (C)
```

Which memory order should we use?

Under the Hood

How does std::atomic_thread_fence establish this ordering of events?

X86_64, GCC (trunk), Debug build (O0)

```
// prepare memcpy arguments...
mov    $0x40,%edx  // 64 bytes
call   400940 <memcpy@plt> // std::memcpy

// tail_ = (tail_ + 1) & (depth_ - 1)
lea    0x1(%rax),%rbx      // tail_ + 1
and    %rax,%rbx           // & (depth_ - 1)
// ...
mov    %rbx,0x28(%rax)     // stores tail_

movl   $0x3,-0x4(%rbp) // 0x3 = memory_order_release
lock orq $0x0,(%rsp)   // full barrier???

// *doorbell_ = tail_
mov    0x18(%rax),%rbx   // reload doorbell_
mov    0x28(%rax),%r12   // reload tail_???
mov    %r12d,(%rbx)       // MMIO doorbell write
```

ARM64, GCC (trunk), Debug build (O0)

```
// prepare memcpy arguments...
mov x2, #0x40      // 64 bytes
bl  memcpy  // std::memcpy

// tail_ = (tail_ + 1) & (depth_ - 1)
add  x10, x10, #0x1        // tail + 1
and  x10, x10, x11         // & (depth - 1)
str  x10,  [x9, #0x28]     // stores tail_


dmb ish // the barrier

// *doorbell_ = tail_
ldr  x10, [x9, #0x28]  // x10 = this->tail_
ldr  x9,  [x9, #0x18]  // x9  = this->doorbell_
// ...
str  w10,  [x9]  // *doorbell_ = (uint32_t)tail_
```

We asked for release barrier but got a `seq_cst` barrier...

Under the Hood

How does std::atomic_thread_fence establish this ordering of events?

X86_64, GCC (trunk), Release build (O3 + LTO)

```
// memcpy(...): written as 2 32-byte AVX stores
// inlined...

// tail_ = (tail_ + 1) & (depth_ - 1)
vpaddq 0x20(%rdi),%xmm0,%xmm0
vpsrldq $0x8,%xmm0,%xmm1
vpand  %xmm1,%xmm0,%xmm0
vmovq  %xmm0,0x28(%rdi)  // stores tail_

// *doorbell_ = tail_
mov    0x18(%rdi),%rax   // reload doorbell_
mov    0x28(%rdi),%rdx   // reload tail_???
mov    %edx,(%rax)       // MMIO doorbell write
```

ARM64, GCC (trunk), Release build (O3+LTO)

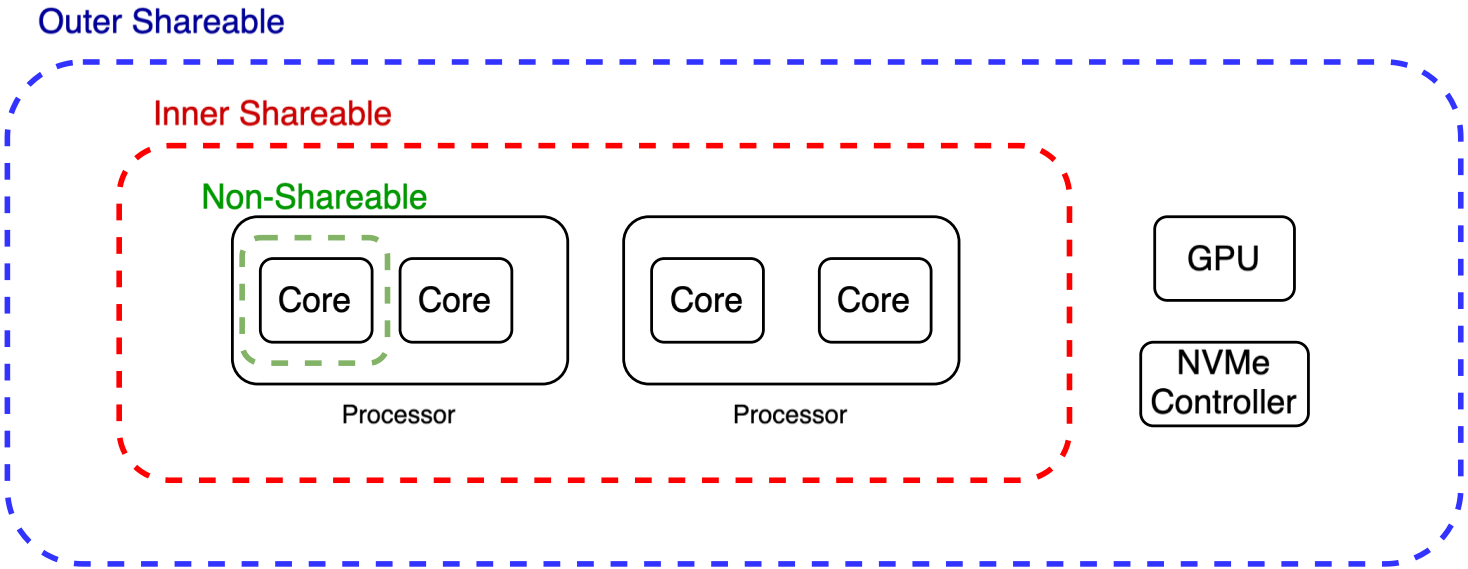
```
// memcpy(...):
// inlined...

// tail_ = (tail_ + 1) & (depth_ - 1)
add  x10, x10, #0x1 // tail + 1
sub  x11, x11, #0x1 // depth - 1
and  x10, x11, x10  // &
str  x10, [x8, #0x60] // stores tail_

dmb ish

// *doorbell_ = tail_
ldr  x8, [x8, #0x50] // reload doorbell
str  w10, [x8]       // MMIO doorbell write
```

Now there is the correct barrier: nothing

ARM64: DMB *ISH*?

<option>	Description	Ordered Access	Sharebility Domain
ISH	R/W before barrier can't be reordered to after	Any - Any	Inner Shareable
OSH	R/W before barrier can't be reordered to after	Any - Any	Outer Shareable
ISHLD	Loads before barrier must be completed	Load – Load/Store	Inner Shareable
ISHST	Stores before barrier must be completed, loads can be reordered freely	Store - Store	Inner Shareable
OSHLD	Loads before barrier must be completed	Load – Load/Store	Outer Shareable
OSHST	Stores before barrier must be completed, loads can be reordered freely	Store - Store	Outer Shareable

Bonus: ARM Shareability Domains



Bonus: How Linux Does It

The NVMe driver in the Linux kernel uses the following functions for dealing with DMA devices:

```
// drivers/nvme/host/pci.c
static inline void nvme_write_sq_db(struct nvme_queue *nvmeq, bool write_sq) {
    // ...
    if (nvme_dbbuf_update_and_check_event(nvmeq->sq_tail, nvmeq->dbbuf_sq_db, nvmeq->dbbuf_sq_ei))
        writel(nvmeq->sq_tail, nvmeq->q_db);
    nvmeq->last_sq_tail = nvmeq->sq_tail;
}

// include/arch-generic/io.h
static inline void writel(u32 value, volatile void __iomem *addr) {
    // ...
    __io_bw();
    __raw_writel((u32 __force)__cpu_to_le32(value), addr);
    __io_aw();
    // ...
}

// arch/arm64/include/asm/io.h:
#define __io_bw() dma_wmb()

static __always_inline void __raw_writel(u32 val, volatile void __iomem *addr) {
    // ...
    asm volatile("str %w0, %1" : : "rZ" (val), "Qo" (*ptr));
}

// arch/arm64/include/asm/barrier.h:
#define __dma_wmb() dmb(oshst)

#define dmb(opt) asm volatile("dmb " #opt : : : "memory")
```

writel:

```
dmb oshst // all earlier stores (the SQE) ordered
str w0, [x1] // ...the doorbell store
```