Martin Ombura Jr.

# Rust's Atomic Memory Model

A dive into memory consistency and memory ordering in Rust

EuroRust 2025

# Bio

## Martin Ombura Jr.

Senior Backend Software Engineer

Previously **@GoDaddy**
MSc. Computer Science (Evolutionary Algorithms)

LinkedIn/Github: **@martinomburajr**
Email Me @ **rust@martinomburajr.com**

# ToC

# 10,000 Ft View

1. Hardware Micro-architecture
2. Compiler Optimizations
3. Program Execution

# Micro-Architecture

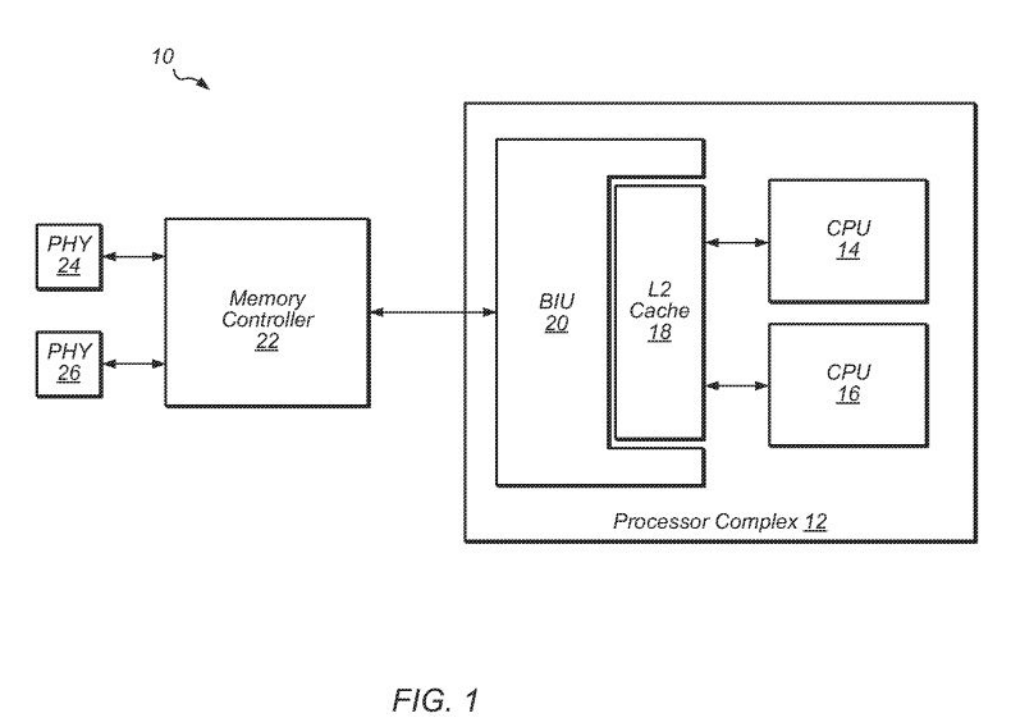
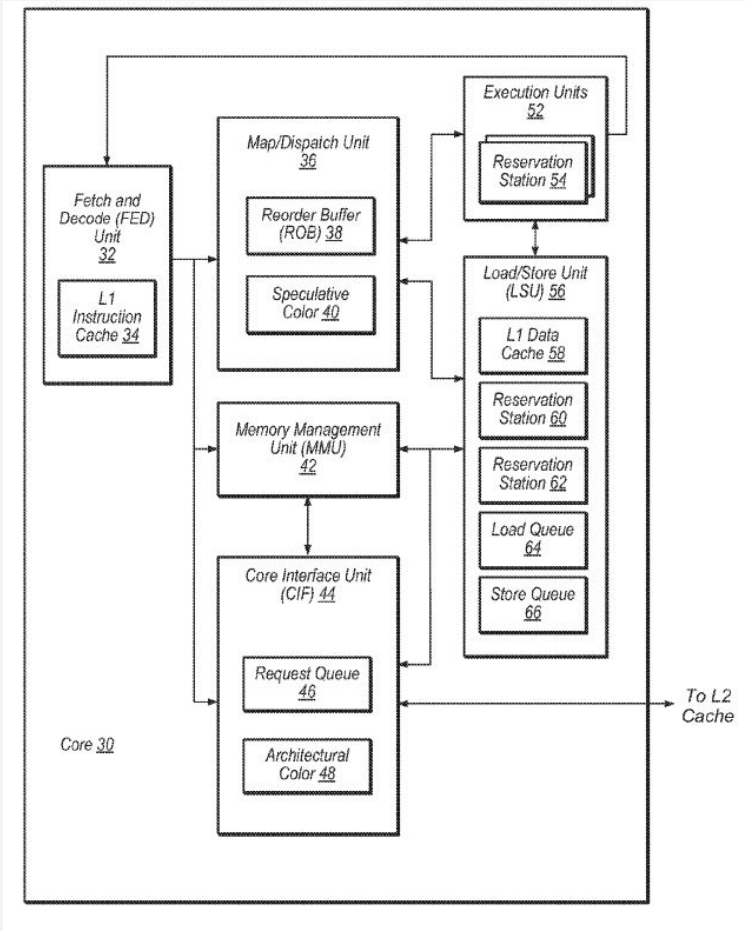



FIG. 1




*"Processor and method for implementing barrier operation using speculative and architectural color values"* - ***Apple Inc***

https://patentimages.storage.googleapis.com/73/f6/3a/61ae2b83fe4284/US9582276.pdf

# Compiler Optimizations

```rust
static mut A: u32 = 0;
static mut B: u32 = 0;
static mut C: u32 = 0;

fn main() {
    unsafe {
        A = 3;
        B = 4;
        A = A + B;
        C = B; // This might be moved up
        println!("{A} {B} {C}"); // When only a single thread is involved, the results are as expected: the line `7 4 4` gets printed.
        C = A; // Unused
    }
}
```

```rust
fn main() {
        println!("7 4 4"); // After extreme optimizations
}
```

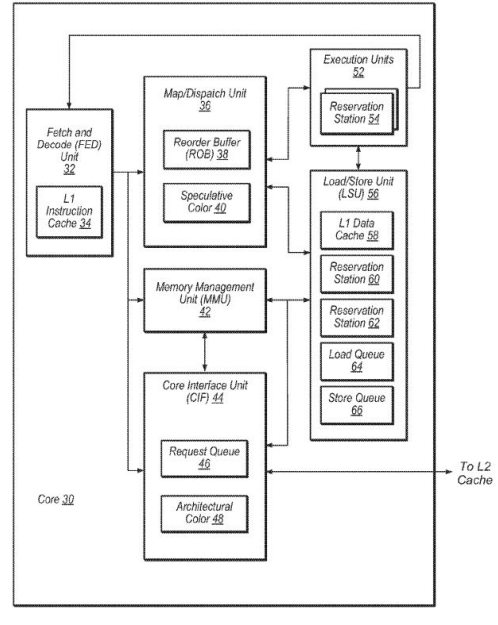
Under the ***as-if rule***, the compiler may perform any transformation that preserves observable behavior. If it proves the output is always `"7 4 4"`, it may replace the whole body with a direct print of that literal.

***“The Compiler knows what you said not what you meant”***
***- Fedor Pikus***

https://doc.rust-lang.org/beta/std/sync/index.html

# Program Execution

```
STR R12, [R1]     ; Access-1: store R12 into memory at R1
LDR R0, [SP], #4  ; Access-2: load from [SP], then post-increment SP by 4
LDR R2, [R3, #8]  ; Access-3: load from [R3 + 8]
```



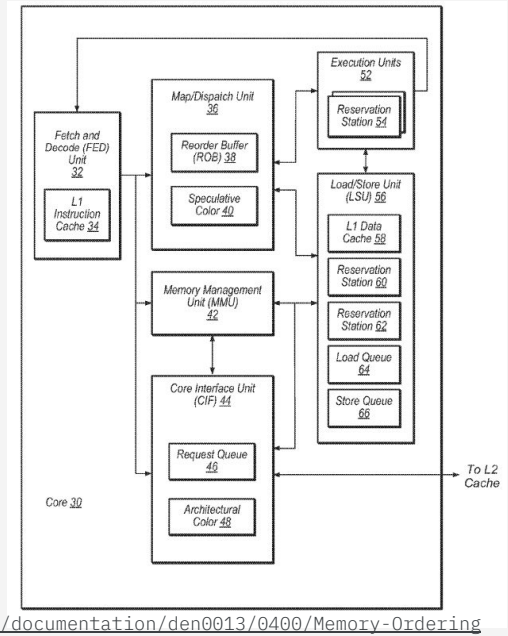

**Plausible Execution Scenario**

1. **Access 1** - performs a write that goes to write buffer
2. **Access 2** - performs a read requiring a cache hit, we can assume it misses meaning we have to fetch the block from main-memory (or another CPUs cache)
3. **Access 3**- performs a read that hits the cache and immediately returns and **completes**
4. **Access 2** - finds the required cache line and returns the data and **completes**
5. **Access 1** - the write buffer finally flushes it's write and **completes**

https://developer.arm.com/documentation/den0013/0400/Memory-Ordering

# Program Execution: Single Threaded

```
STR R12, [R1]     ; Access-1: store R12 into memory at R1
LDR R0, [SP], #4  ; Access-2: load from [SP], then post-increment SP by 4
LDR R2, [R3, #8]  ; Access-3: load from [R3 + 8]
```



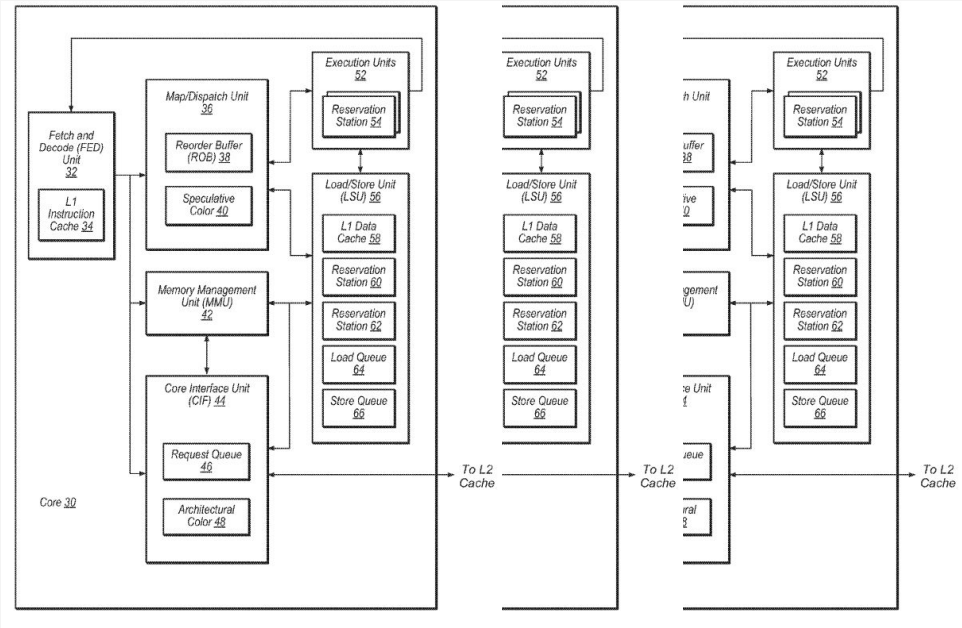

## Single Threaded Execution

- In a single thread, instructions retire in **program order**.
- Instructions may *complete* out of order, but the retirement machinery ensures that they *retire* in program order.
- The architecture guarantees **“as-if”** the thread executed sequentially, even if the *microarchitecture* overlaps or reorders work internally
- These differences in latency are handled opaquely by the internal caches/queues
- In architectures like ARM & x86, a CPU always observes its own loads/stores in program order

https://developer.arm.com/documentation/den0013/0400/Memory-Ordering
AMD64 Programmer Manual (2024) - Chapter 7.2

# Program Execution: Multi-Threaded

```
STR R12, [R1]     ; Access-1: store R12 into memory at R1
LDR R0, [SP], #4  ; Access-2: load from [SP], then post-increment SP by 4
LDR R2, [R3, #8]  ; Access-3: load from [R3 + 8]
```




**Multi-threaded Execution**

**Weak-Ordered System (ARM/POWER)**

- Stores may sit in a store-buffer before they reach the point of coherence. Resulting in other cores still seeing older values
- Without synchronization, independent loads and stores to different addresses are observed in different orders by other cores

**Outcome?**

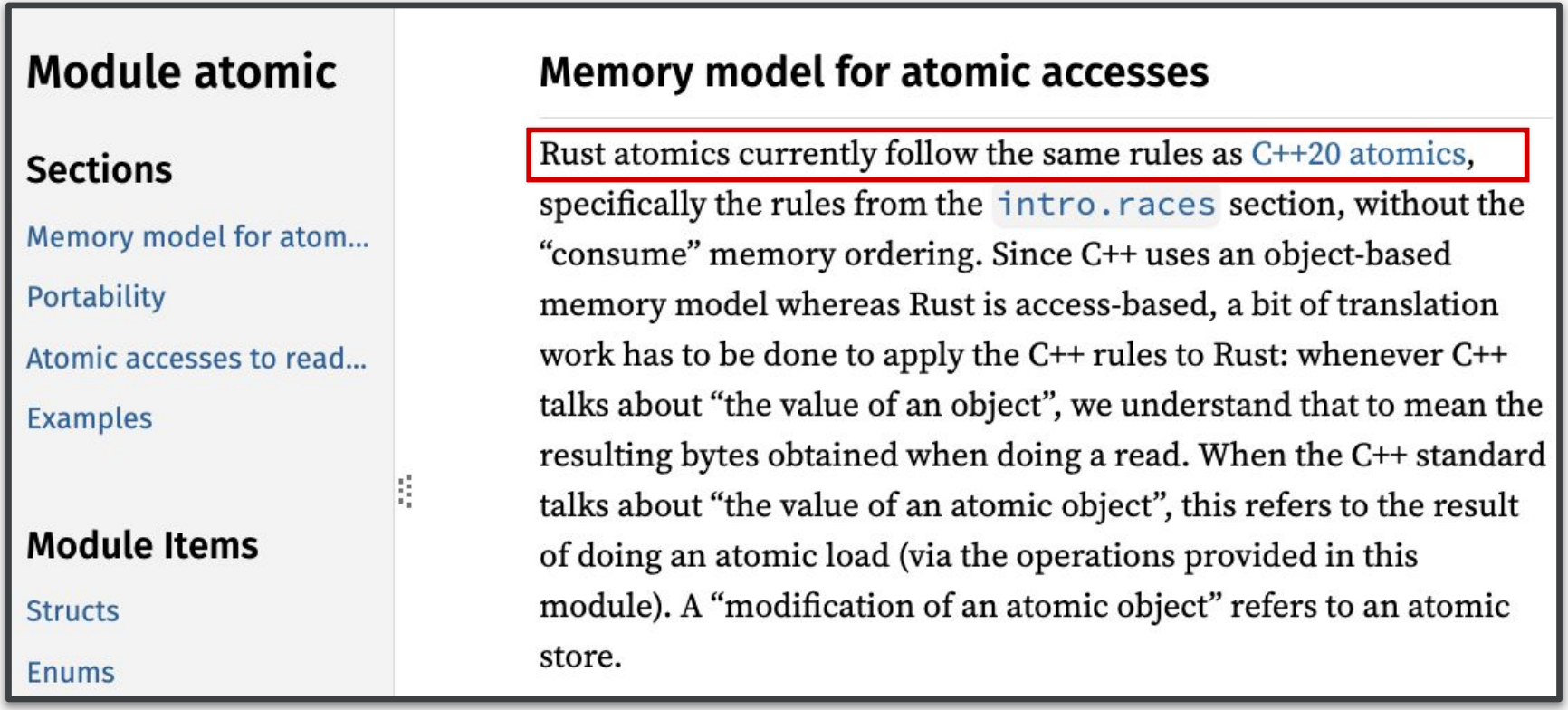
- **Data Races**
  - Other cores can observe Access-3's load before Access-1's store becomes visible.
  - Or even observe stores in opposite order
- **Stale Data**

https://developer.arm.com/documentation/den0013/0400/Memory-Ordering

# Rust's Atomic Memory Model

1. Rust's Atomic Access Model
2. Atomic Variables
3. Atomics & Coherence
4. Memory Ordering
5. Atomic Methods

# Rust's Atomic Access Memory Model



## Module atomic

### Sections

Memory model for atom...

Portability

Atomic accesses to read...

Examples

### Module Items

Structs

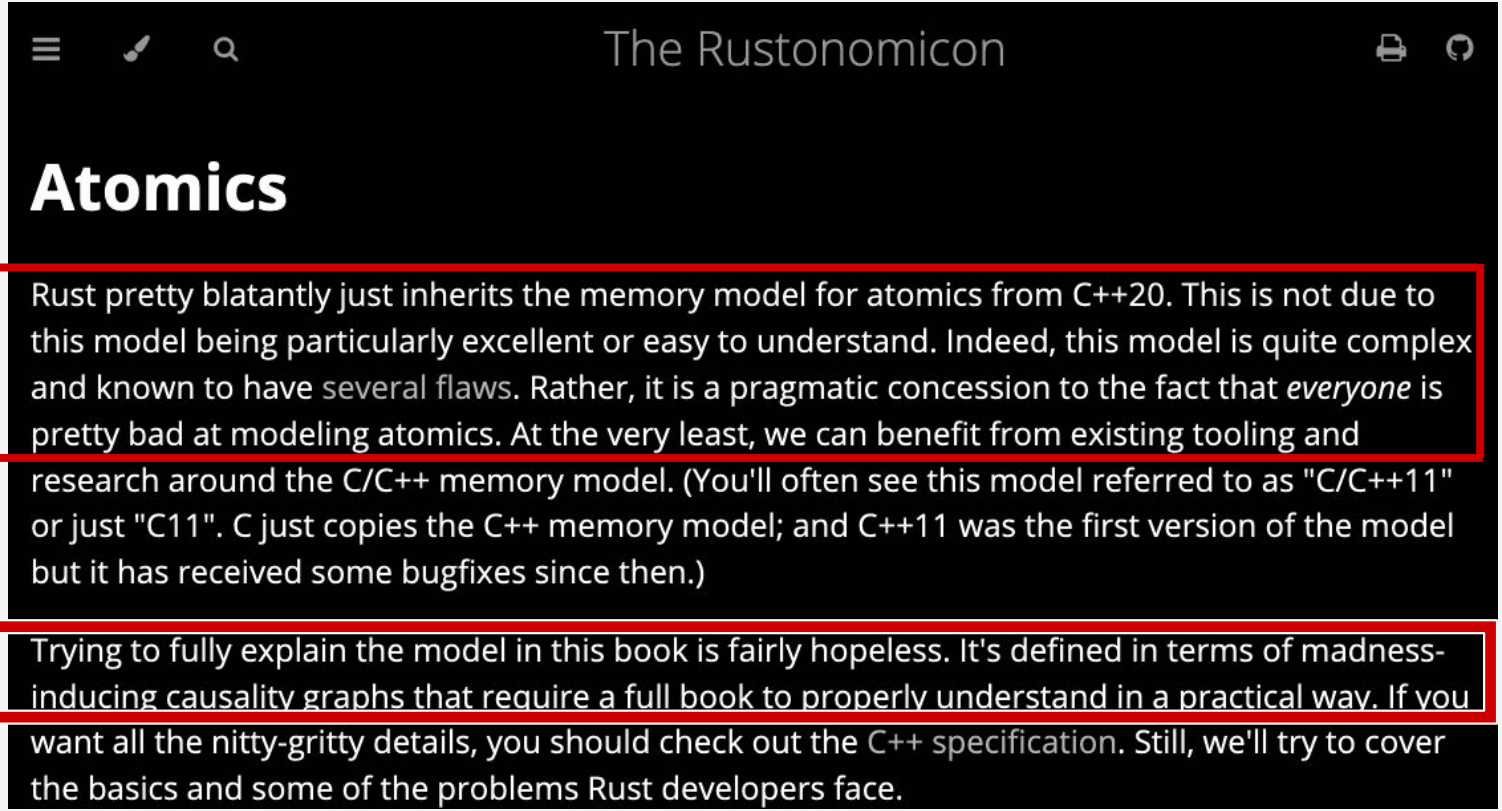
Enums

## Memory model for atomic accesses

Rust atomics currently follow the same rules as C++20 atomics, specifically the rules from the `intro.races` section, without the "consume" memory ordering. Since C++ uses an object-based memory model whereas Rust is access-based, a bit of translation work has to be done to apply the C++ rules to Rust: whenever C++ talks about "the value of an object", we understand that to mean the resulting bytes obtained when doing a read. When the C++ standard talks about "the value of an atomic object", this refers to the result of doing an atomic load (via the operations provided in this module). A "modification of an atomic object" refers to an atomic store.

https://doc.rust-lang.org/std/sync/atomic/

# Rust's Atomic Access Memory Model




https://doc.rust-lang.org/nomicon/atomics.html

# Atomic Variables

- An atomic variable is a memory location that the language and hardware treat as a single, indivisible unit for specific operations
- Atomic types provide primitive shared-memory communication between threads, and are the building blocks of other concurrent types.

In Rust, the `std::sync::atomic` module defines these types and orderings:

- AtomicBool
  - A boolean type which can be safely shared between threads.
- Atomic{I,U}{8,16,32,64}
  - An integer type which can be safely shared between threads e.g. AtomicU16.
- AtomicPtr
  - A raw pointer type which can be safely shared between threads.
- Atomic{I128|U128}
  - Available on nightly releases
  - May have stricter alignment on some targets Atomic{I,U}{8,16,32,64}

**https://doc.rust-lang.org/std/sync/atomic/**

# Atomic Variable Characteristics/Usage

**Usage**

- Implement both `Send` and `Sync` marker traits
- But, they do not themselves provide the mechanism for sharing and follow the Rust threading model.
  - Most common way to share an atomic variable is in an `Arc`, and as a `static (for lazy global init)`

**Portability**

- All atomic types in `std` are guaranteed to be **lock-free** if they are available, i.e they don't internally acquire a global mutex.
- Atomic types and operations are **not guaranteed to be wait-free.** i.e operations like `fetch_or` may be implemented with a compare-and-swap loop.

**Ordering**

Memory ordering options applied to atomic operation can impose ordering on surrounding memory accesses to participate in inter-thread synchronization.

**Hardware Restrictions**

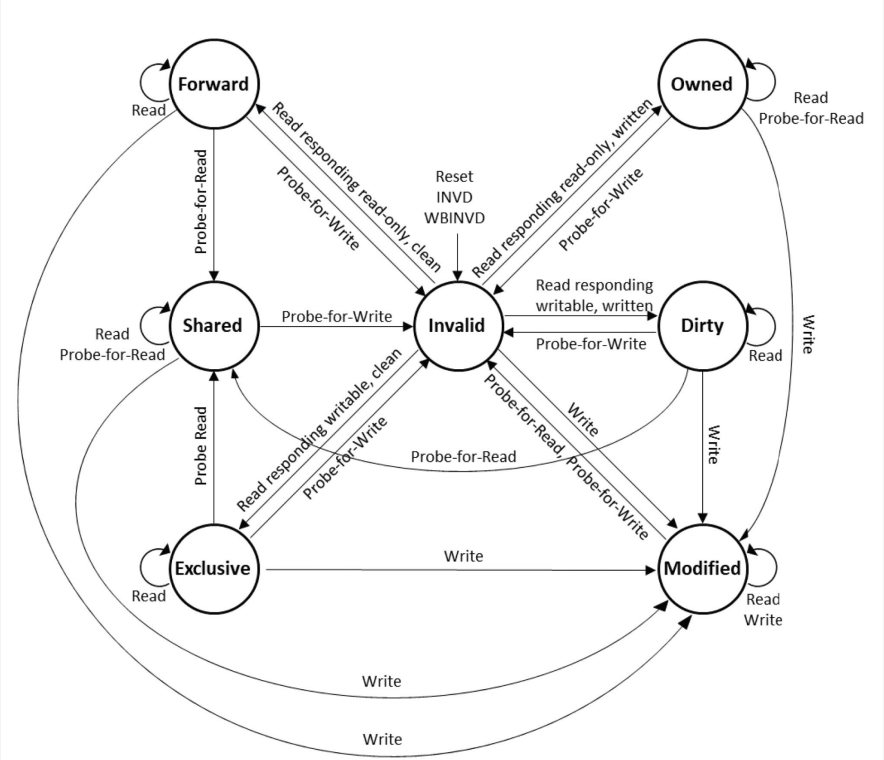
Hardware atomicity guarantees typically require that:

- Atomics are naturally aligned
- Atomics do not straddle cache-line boundaries efficient atomicity
  - ***Note*** *x86 locked instructions remain atomic even across a cache-line boundary*
- Avoid “false-sharing”
  - Separate atomics on the same cache-line causes back and forth invalidations
- Non-synchronized conflicting atomic accesses may not partially overlap
  - e.g. using AtomicU16 and a transmuted &AtomicU8

https://doc.rust-lang.org/std/sync/atomic/
https://doc.rust-lang.org/nomicon/atomics.html
https://doc.rust-lang.org/src/core/sync/atomic.rs.html

# Atomics & Cache Coherence

- Where does “atomicity” come from?
- Atomic operations execute within the cache-coherence hierarchy. They acquire ownership of a cache line (or observe it) using the coherence protocol, and their effects become visible to other cores via cache-to-cache transfers, invalidations, and writebacks
- Coherence Protocols (MESI/MOESI) provide atomic guarantees e.g.
  - Single writer at a time per cache-line
- Store buffers, invalidate queues and memory ordering interact with coherence framework. The coherence framework decides when other cores can view the outcome of writes.
  - This microarchitectural outcomes underpin release/acquire/seqcst behaviors




MOESDIF Protocol in x86 (AMD64)

AMD64 Architecture Programmer’s Manual

# Memory Ordering Theory

1. Sequenced-Before
2. Modification Order
3. Synchronizes With
4. Happens-Before

# Memory Ordering: Key Definitions

**Sequenced-Before (sb)**

- A strict, program order of operand and statement evaluation.
- ***Note:*** This is a C++ spec term to describe “evaluation order”

**Modification Order (mo)**

- The total order of all modifications to any particular atomic variable

**Synchronizes with (sw)**

- For a given atomic X and two threads A and B, if:
  - An atomic store in A is a **release** operation on X, and,
  - An atomic load in B is an **acquire** operation on X, and,
  - the acquire load in B reads the value in the release store written by A

  then the load in A synchronizes with (sw) the load in B

**Happens-Before (hb)**

- **Intra-Thread HB**
  - Same as sb but now expressed in hb terms.
- **Inter-Thread HB**
  - When evaluation A synchronizes-with (sw) evaluation B

https://en.cppreference.com/w/cpp/atomic/memory_order.html

# Modification Order (MO)

**Definition:** All modifications to any particular atomic variable occur in a total order that is specific to a given atomic variable.

- E.g. `X.store(1); X.store(2); X.store(3)` gives a MO of `X: 1->2->3`

- MO is ordering-agnostic.

**Coherence Requirements**

- Effectively constrain the compilers reordering of atomic operations

- There are 4 different variants
  1. **Write-Write (WW) Coherence**
  2. **Read-Read (RR) Coherence**
  3. **Write-Read (WR) Coherence**
  4. **Read-Write (RW) Coherence**

https://en.cppreference.com/w/cpp/atomic/memory_order.html

```
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.store(42, ???); // W1
    X.store(100, ???); // W2
})
```

**single-threaded**

```
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.store(42, ???); // W1
})

// Thread 2:
thread::spawn(|| {
    X.store(100, ???); // W2
})
```

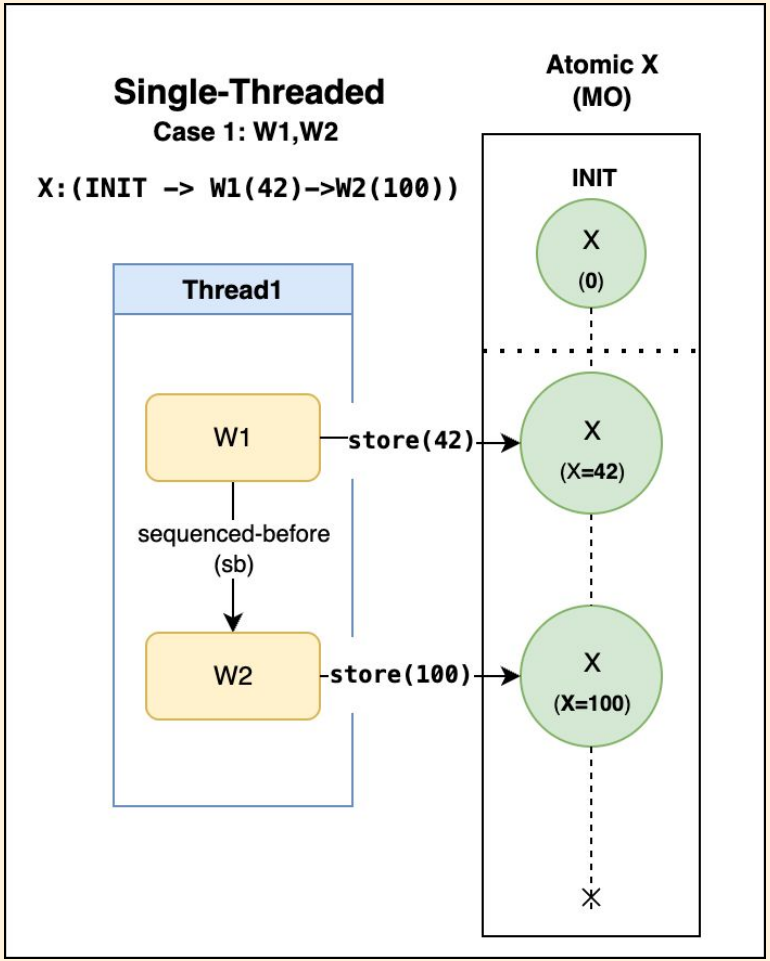
**multithreaded**

# Write-Write Coherence

If write W1 to atomic `X` **happens before** write W2 to atomic X, then W1 is earlier than W2 in the X's **modification order**.

# WW-Single-Threaded: Case 1





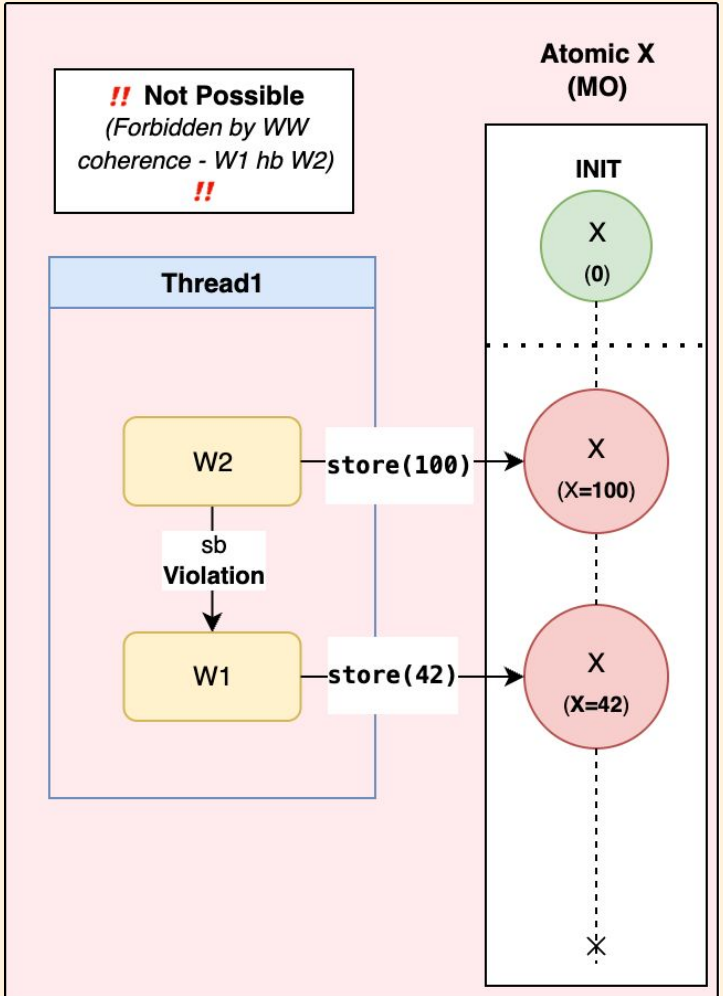

```rust
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.store(42, ???); // W1
    X.store(100, ???); // W2
})
```

## Write-Write Coherence

If write W1 to atomic `X` **happens before** write W2 to atomic X, then W1 is earlier than W2 in the X's **modification order**.

# WW-Single-Threaded: Case 2



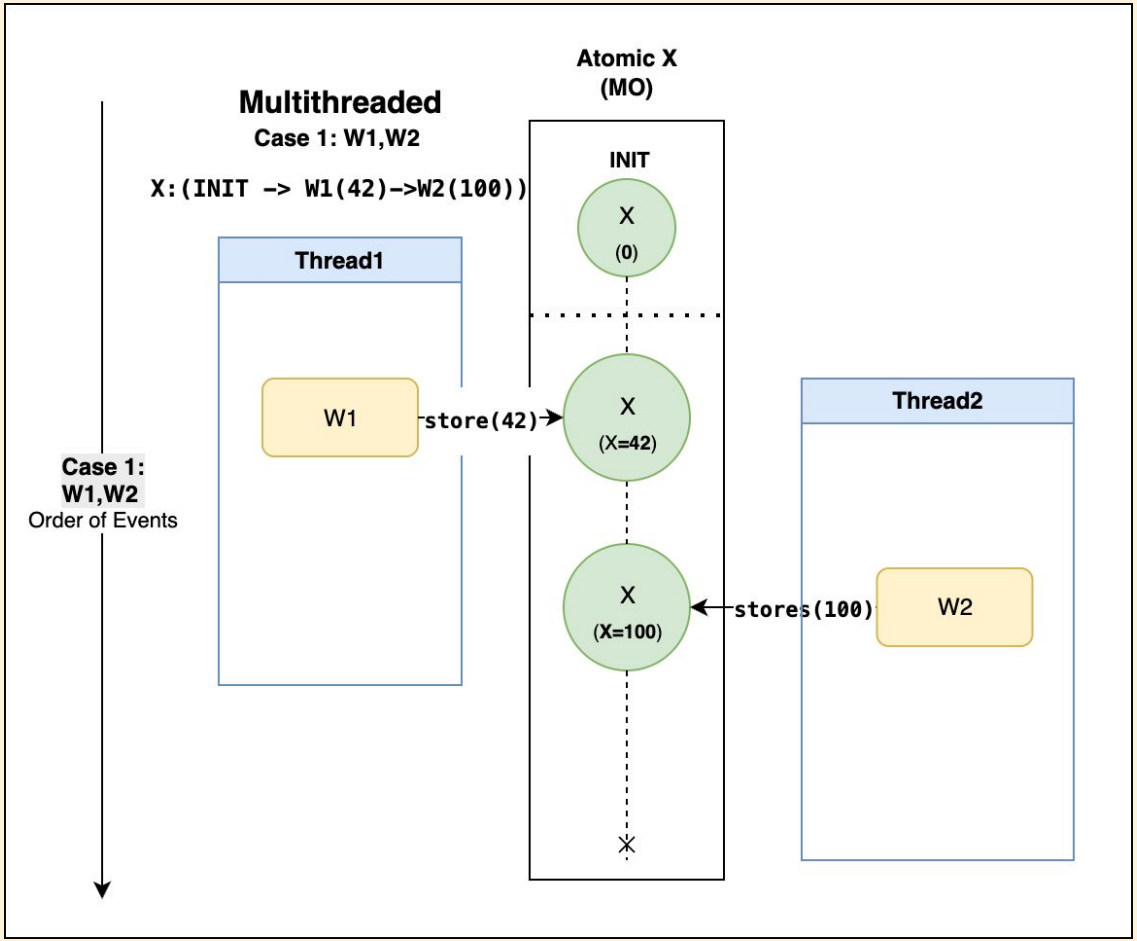



```rust
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.store(42, ???); // W1
    X.store(100, ???); // W2
})
```

## Write-Write Coherence

If write W1 to atomic `X` **happens before** write W2 to atomic X, then W1 is earlier than W2 in the X's **modification order**.

# WW- Multithreaded: Case 1



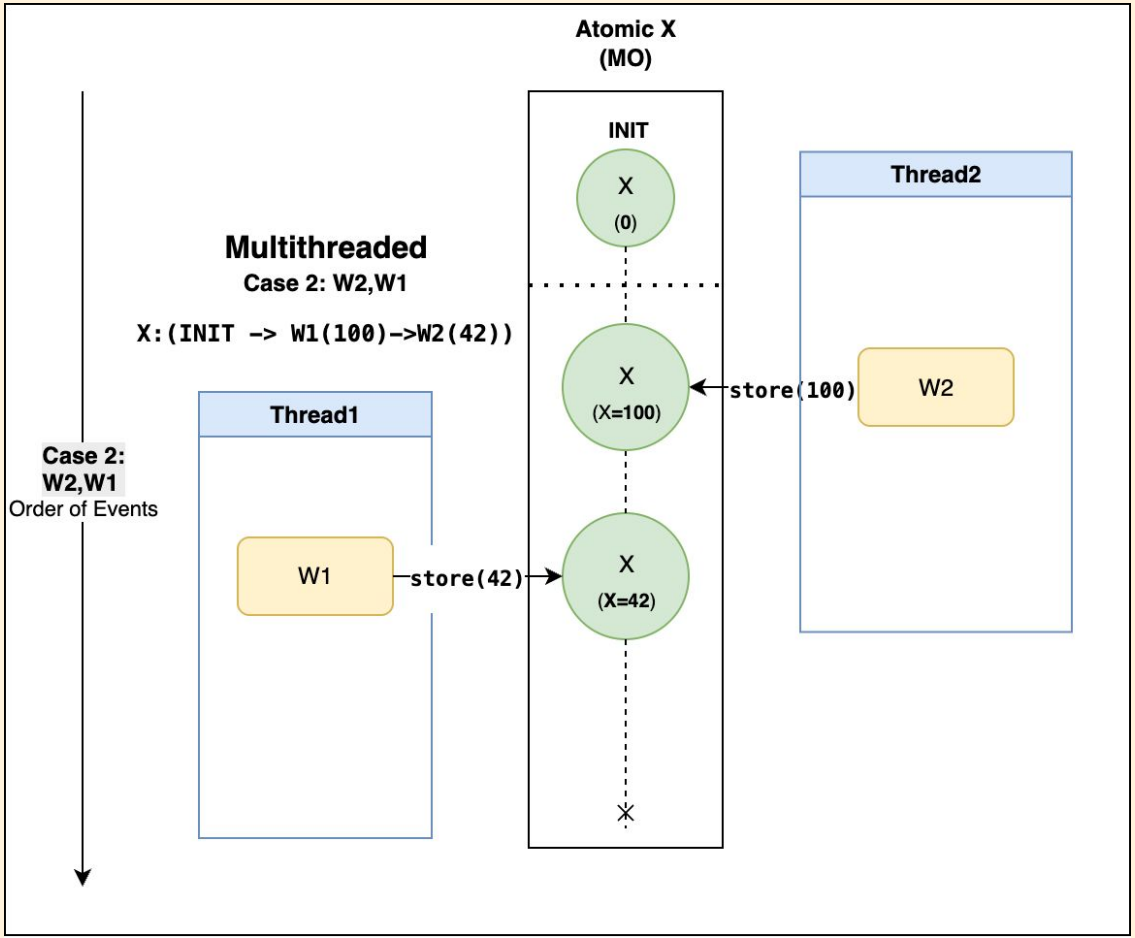

**Modification Order**

```rust
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.store(42, ???); // W1
})

// Thread 2:
thread::spawn(|| {
    X.store(100, ???); // W2
})
```

## Write-Write Coherence

If write W1 to atomic `X` **happens before** write W2 to atomic X, then W1 is earlier than W2 in the X's **modification order**.

# WW-Multithreaded: Case 2




Modification Order

```rust
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.store(42, ???); // W1
})

// Thread 2:
thread::spawn(|| {
    X.store(100, ???); // W2
})
```

## Write-Write Coherence

If write W1 to atomic `X` **happens before** write W2 to atomic X, then W1 is earlier than W2 in the X's **modification order**.

```
1 static X: AtomicI32 = AtomicI32::new(0);
2
3 // Thread 1:
4 thread::spawn(|| {
5     X.store(42, ???); // W1
6     X.load(???); // R1
7     X.load(???); // R2
8 })
```

**single-threaded**

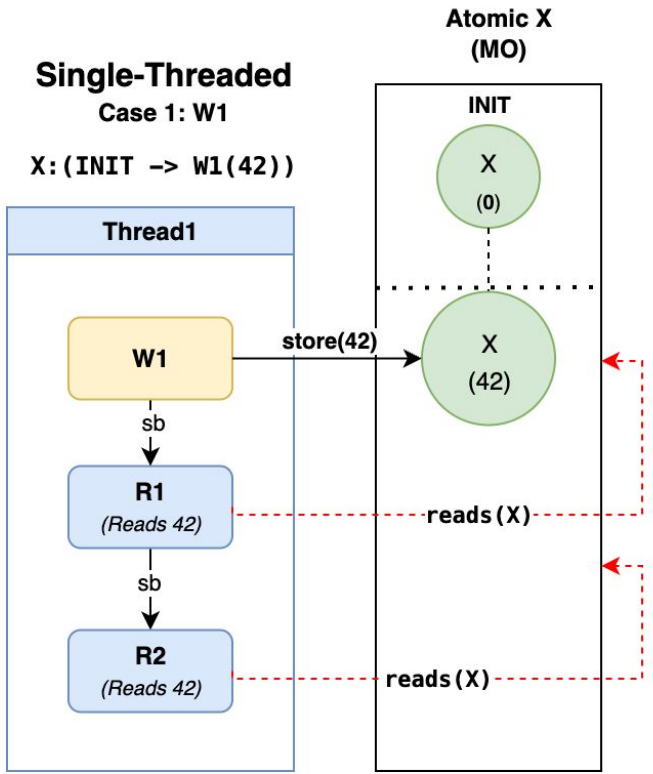
```
1 static X: AtomicI32 = AtomicI32::new(0);
2
3 // Thread 1:
4 thread::spawn(|| {
5     X.load(???); // R1
6     X.load(???); // R2
7 })
8 // Thread 2:
9 thread::spawn(|| {
10     X.store(42, ???); // W1
11 })
```

**multithreaded**

# Read-Read Coherence

If a read R1 on atomic X **happens-before** a read R2 on X, and R1 took its value from a write W1 on X, then R2 sees either W1, or some later write W2 on X that appears later than W1 in the *modification order* of X

# RR-Single-Threaded: Case 1




| R1 & R2 Outcomes | |
|---|---|
| **R1** | **R2** |
| 42 | 42 |
| 0 | 0 |
| 0 | 42 |
| 42 | 0 |

Due to intra-thread happens-before (sequenced before) + WR coherence, a read after a write must be coherent.

If W1 sets 42, R1 must be 42. Consequently R2 = 42 (unless another write occurs between R1 & R2, then R2 would equal that write

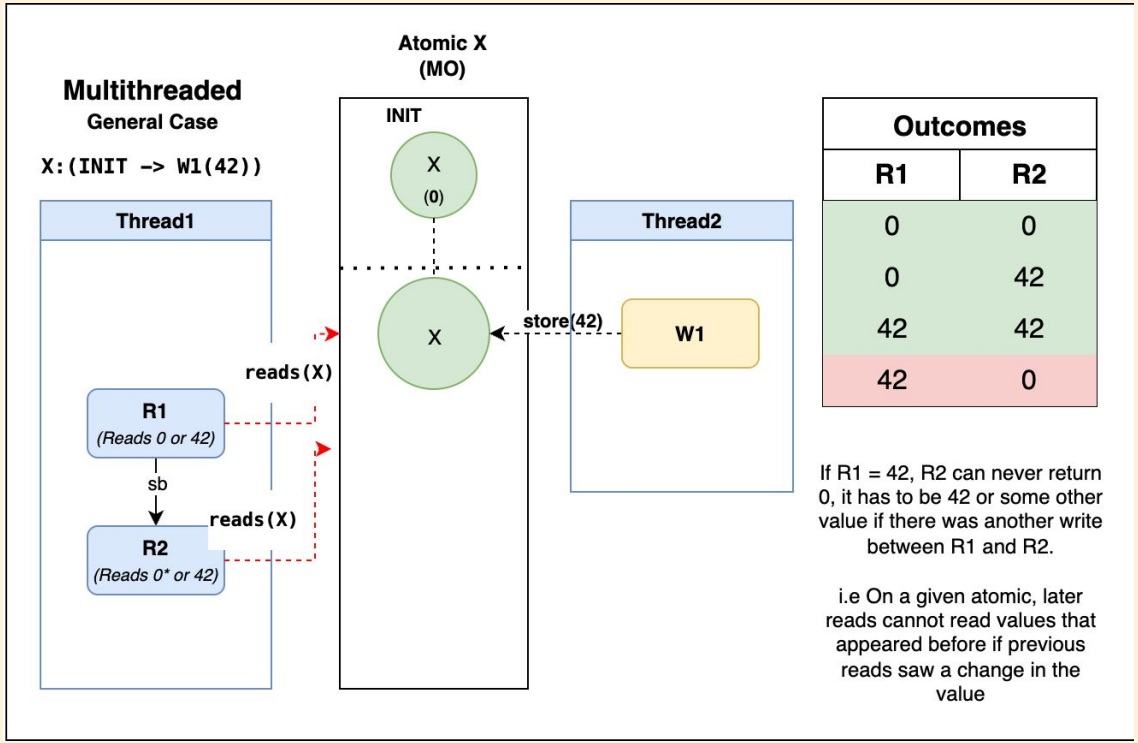
**Modification Order**

```rust
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.store(42, ???); // W1
    X.load(???); // R1
    X.load(???); // R2
})
```

## Read-Read Coherence

If a read R1 on atomic X **happens-before** a read R2 on X, and R1 took its value from a write W1 on X, then R2 sees either W1, or some later write W2 on X that appears later than W1 in the *modification order* of X

# RR-MultiThreaded: General Case




| Outcomes | |
|---|---|
| R1 | R2 |
| 0 | 0 |
| 0 | 42 |
| 42 | 42 |
| 42 | 0 |

If R1 = 42, R2 can never return 0, it has to be 42 or some other value if there was another write between R1 and R2.

i.e On a given atomic, later reads cannot read values that appeared before if previous reads saw a change in the value



```rust
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.load(???); // R1
    X.load(???); // R2
})
// Thread 2:
thread::spawn(|| {
    X.store(42, ???); // W1
})
```

## Read-Read Coherence

If a read R1 on atomic X **happens-before** a read R2 on X, and R1 took its value from a write W1 on X, then R2 sees either W1, or some later write W2 on X that appears later than W1 in the *modification order* of X

```rust
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.load(???); // R1
    X.store(42, ???); // W1
})
```

**single-threaded**

```rust
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.load(???); // R1
    X.load(???); // R2
})
// Thread 2:
thread::spawn(|| {
    X.store(42, ???); // W1
})
```

**multithreaded**

# Read-Write Coherence

If a read R1 of some atomic X *happens-before* a write W1 on X, then the value of R1 comes from the write W0 that appears earlier than W1 in the modification order of X.

```
1 static X: AtomicI32 = AtomicI32::new(0);
2
3 // Thread 1:
4 thread::spawn(|| {
5     X.store(42, ???); // W1
6     X.load(???); // R1
7     X.load(???); // R2
8 })
```

**single-threaded**

```
1 static X: AtomicI32 = AtomicI32::new(0);
2
3 // Thread 1:
4 thread::spawn(|| {
5     X.load(???); // R1
6     X.load(???); // R2
7 })
8 // Thread 2:
9 thread::spawn(|| {
10     X.store(42, ???); // W1
11 })
```

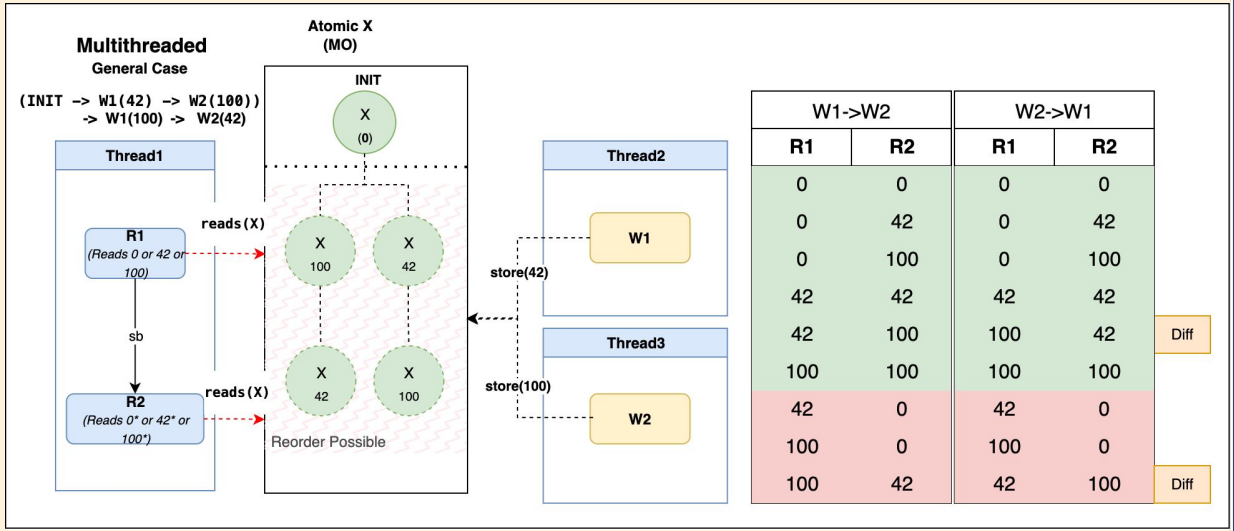
**multithreaded**

# Write-Read Coherence

If a write W1 on an atomic object X **happens-before** a read R1 of X, then the read R1 shall take its value from W1 or from a different write W2 that follows W1 in the *modification order* of X.

# Mixed Example






| W1->W2 | | W2->W1 | | |
|---|---|---|---|---|
| R1 | R2 | R1 | R2 | |
| 0 | 0 | 0 | 0 | |
| 0 | 42 | 0 | 42 | |
| 0 | 100 | 0 | 100 | |
| 42 | 42 | 42 | 42 | |
| 42 | 100 | 100 | 42 | Diff |
| 100 | 100 | 100 | 100 | |
| 42 | 0 | 42 | 0 | |
| 100 | 0 | 100 | 0 | |
| 100 | 42 | 42 | 100 | Diff |

```rust
static X: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.load(???); // R1
    X.load(???); // R2
})

// Thread 2:
thread::spawn(|| {
    X.store(42, ???); // W1
})

// Thread 3:
thread::spawn(|| {
    X.store(100, ???); // W2
})
```

# Memory Ordering

- **Definition**: Memory ordering is the set of rules at the language, compiler and hardware levels that constrain the relative visibility and sequencing of memory operations across threads
  - e.g when one threads writes must become observable to another

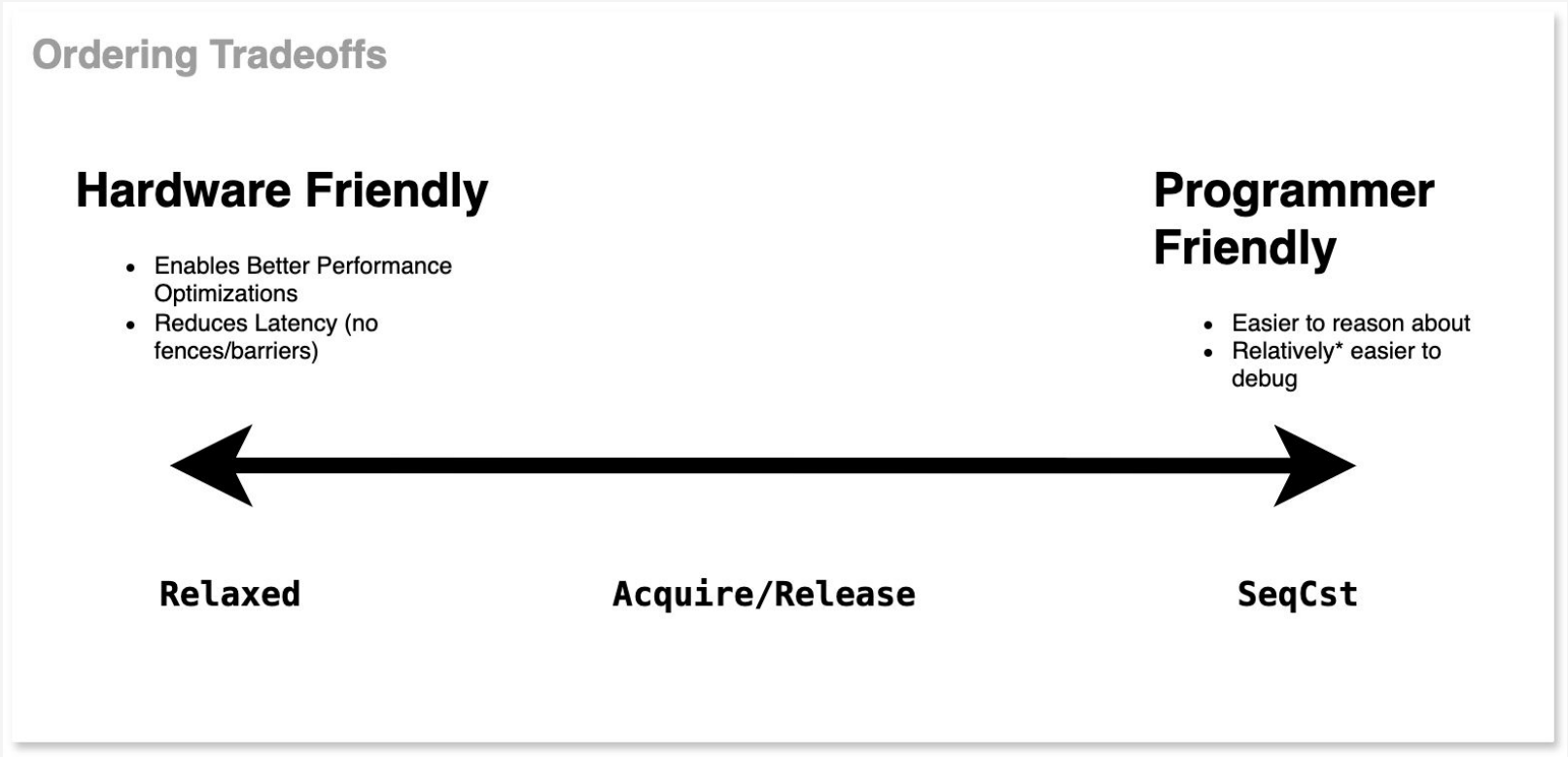
- Each atomic access can be marked with an ordering that specifies what kind of relationship it establishes with other accesses.

- The set of orderings Rust exposes are:

  1. `Relaxed`
  2. `Release`
  3. `Acquire`
  4. `AcqRel`
  5. Sequentially Consistent (`SeqCst`)

https://doc.rust-lang.org/nomicon/atomics.html#hardware-reordering

# General Ordering Tradeoffs




[^1]-https://www.cs.cmu.edu/afs/cs/academic/class/15740-s18/www/lectures/08-09-consistency.pdf

# Relaxed Ordering

- Relaxed Ordering is not a synchronization operation.
  - It does not impose any order among concurrent memory accesses.
  - It only guarantees atomicity and modification order consistency
- Despite Relaxed Ordering not being a synchronization operation, it cannot contribute to data races
- However mixing relaxed atomic operations with unsynchronized non-atomic accesses to the same location is a data race → UB

**Usage**

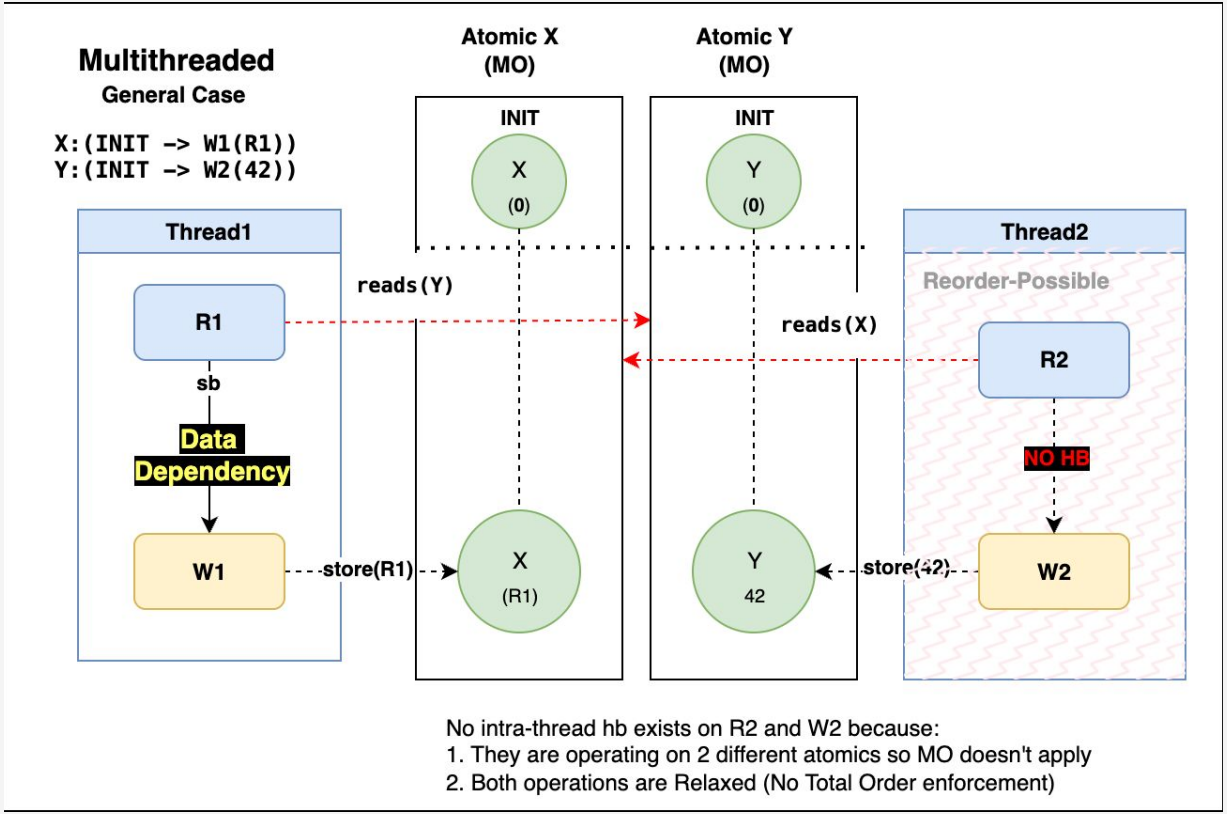
- Primarily used when you care about each atomic operation taking place, but don't care about the ordering such as
- e.g. Increment/Decrement counters
  - These can happen in any order but the total end state will be preserved

- https://eel.is/c%2B%2Bdraft/intro.multithread#intro.races-3.sentence-5
- https://en.cppreference.com/w/cpp/atomic/memory_order.html

# Relaxed Ordering: Use Cases

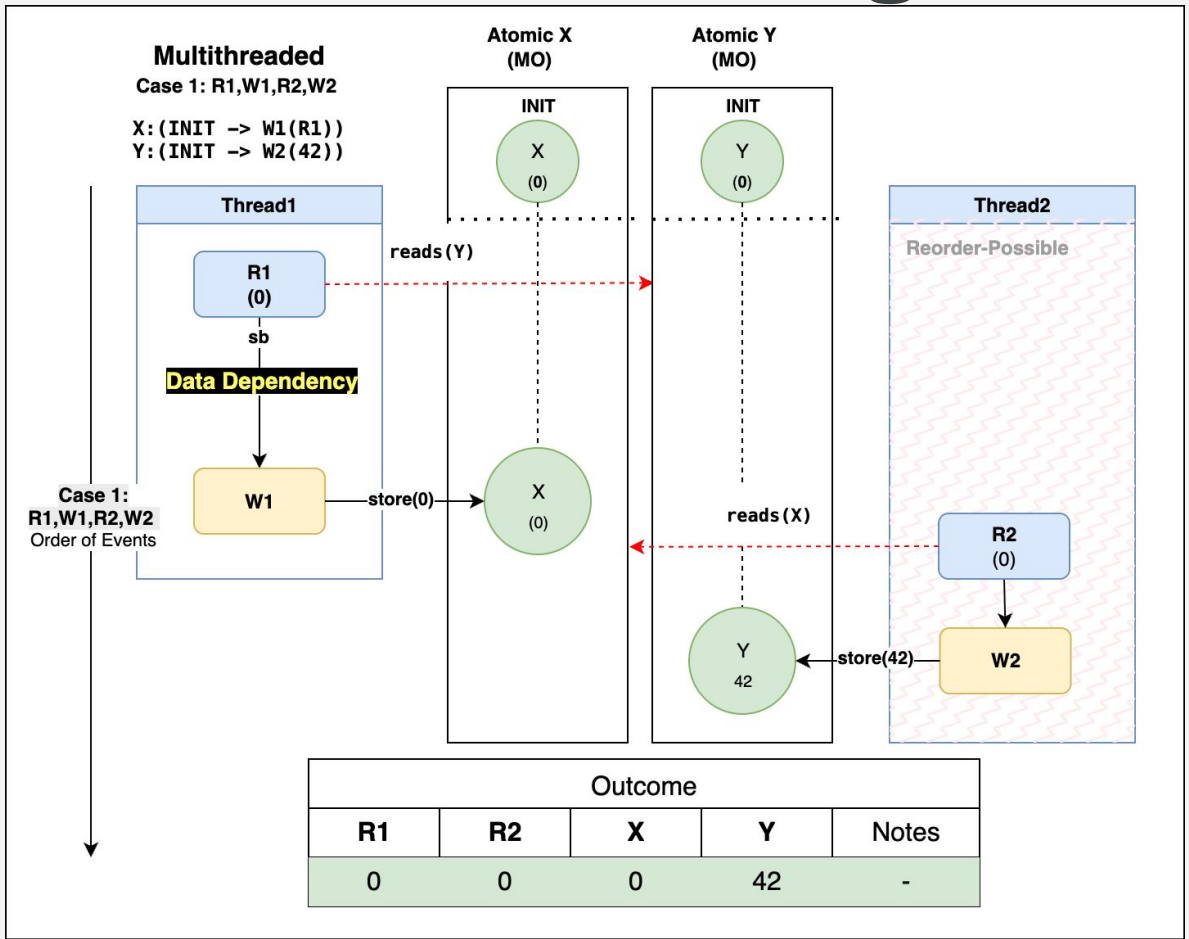
```
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

# Relaxed Ordering: Example




```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

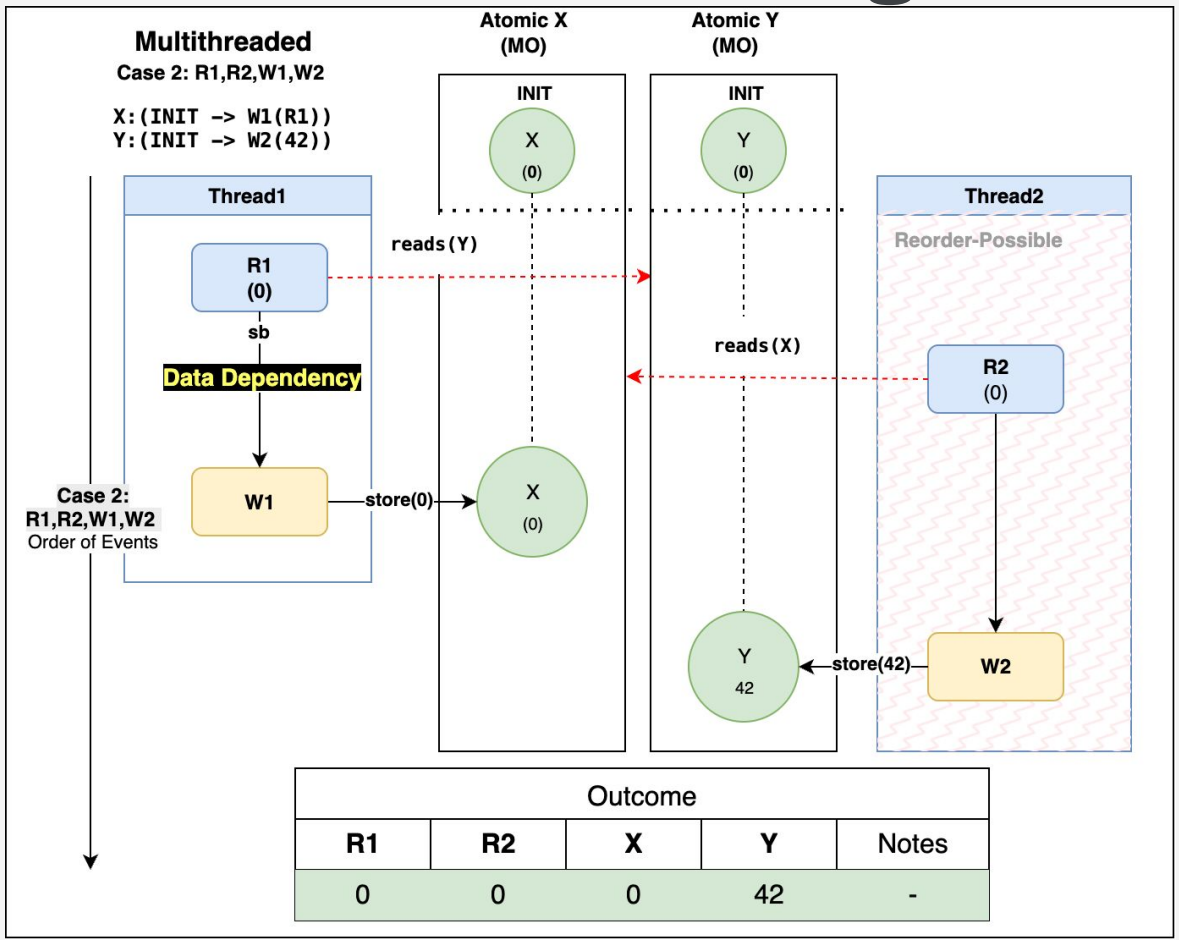
# Relaxed Ordering: Case 1




| Outcome | | | | |
|---|---|---|---|---|
| **R1** | **R2** | **X** | **Y** | Notes |
| 0 | 0 | 0 | 42 | - |

```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

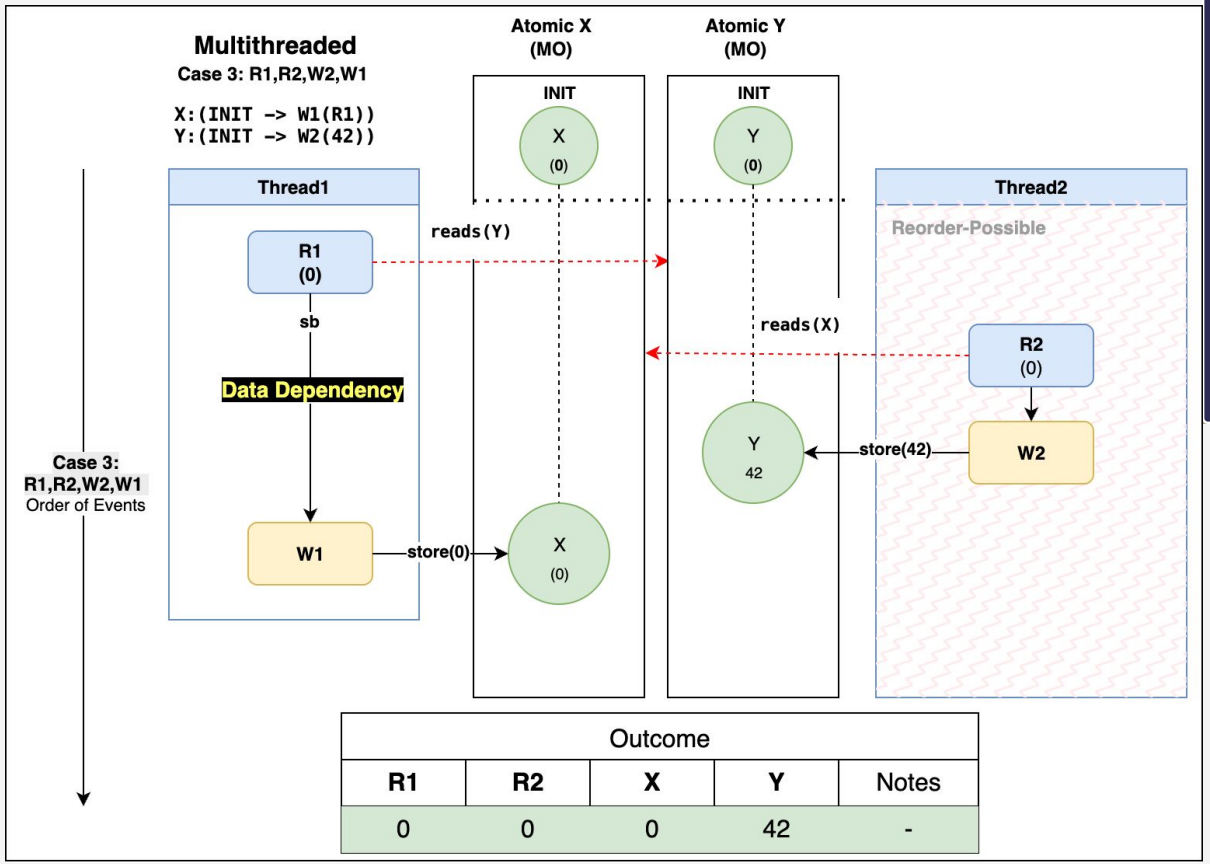
# Relaxed Ordering: Case 2




| Outcome | | | | |
|---|---|---|---|---|
| **R1** | **R2** | **X** | **Y** | Notes |
| 0 | 0 | 0 | 42 | - |

```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

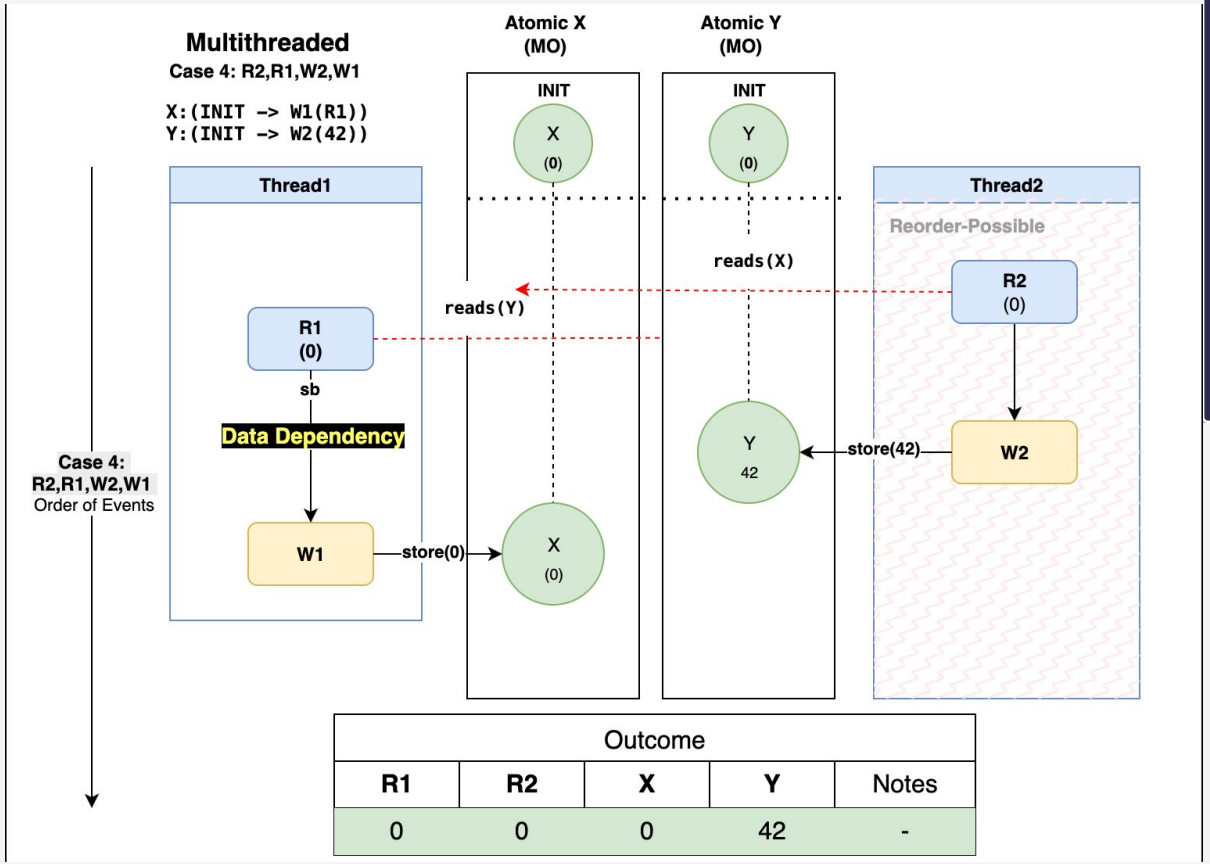
# Relaxed Ordering: Case 3




| Outcome | | | | |
|---|---|---|---|---|
| R1 | R2 | X | Y | Notes |
| 0 | 0 | 0 | 42 | - |

```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

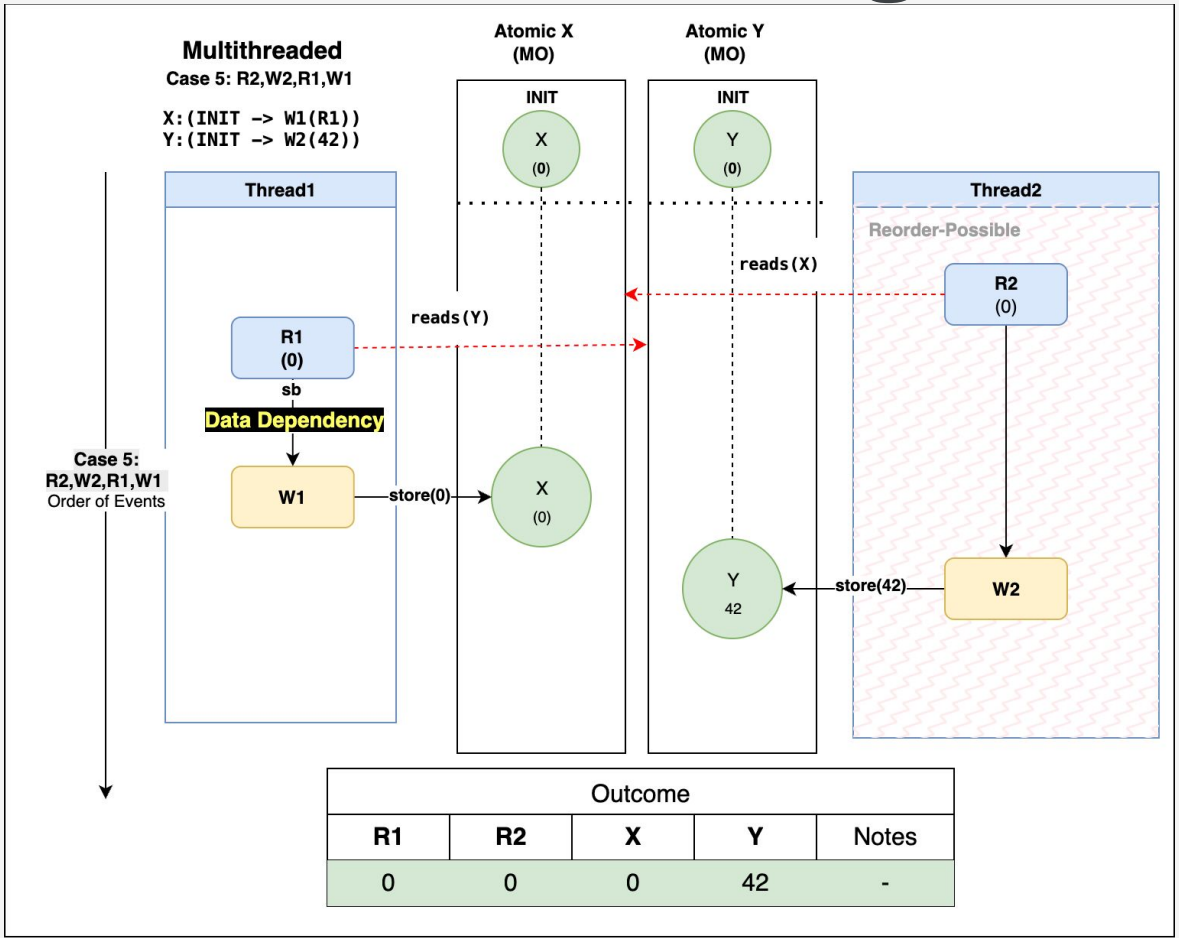
# Relaxed Ordering: Case 4




| Outcome | | | | |
|---|---|---|---|---|
| **R1** | **R2** | **X** | **Y** | Notes |
| 0 | 0 | 0 | 42 | - |

```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

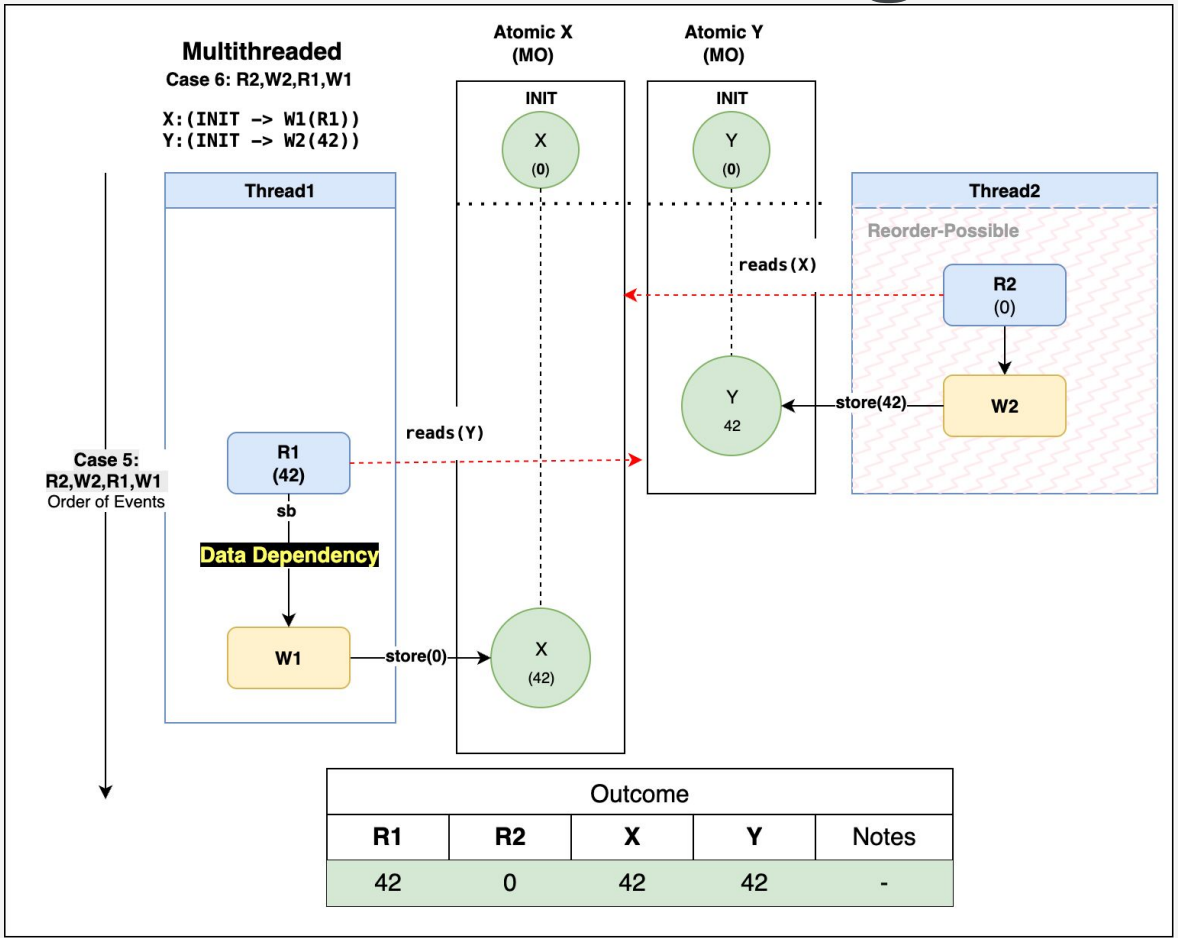
# Relaxed Ordering: Case 5




| Outcome | | | | |
|---|---|---|---|---|
| **R1** | **R2** | **X** | **Y** | Notes |
| 0 | 0 | 0 | 42 | - |

```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

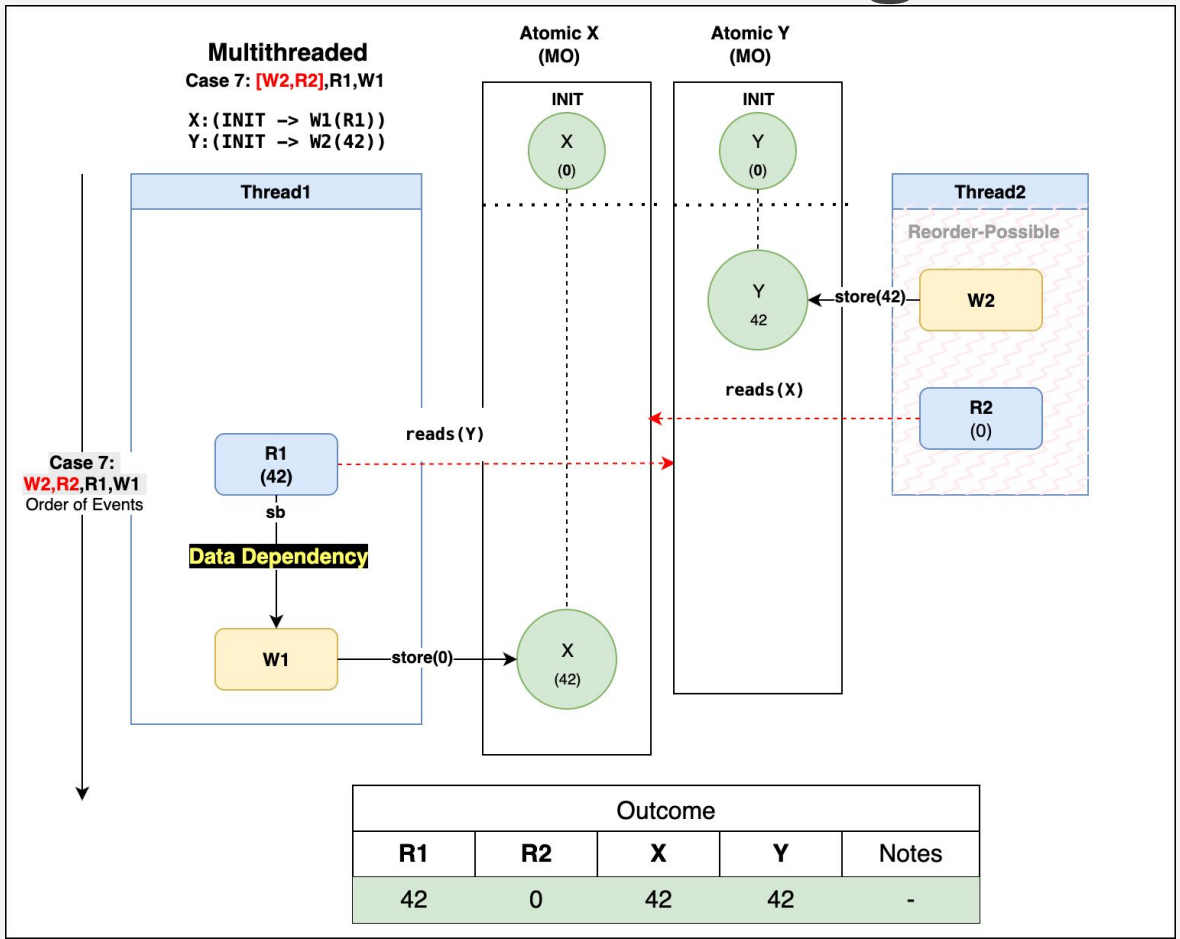
# Relaxed Ordering: Case 6




| Outcome | | | | |
|---|---|---|---|---|
| **R1** | **R2** | **X** | **Y** | Notes |
| 42 | 0 | 42 | 42 | - |

```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

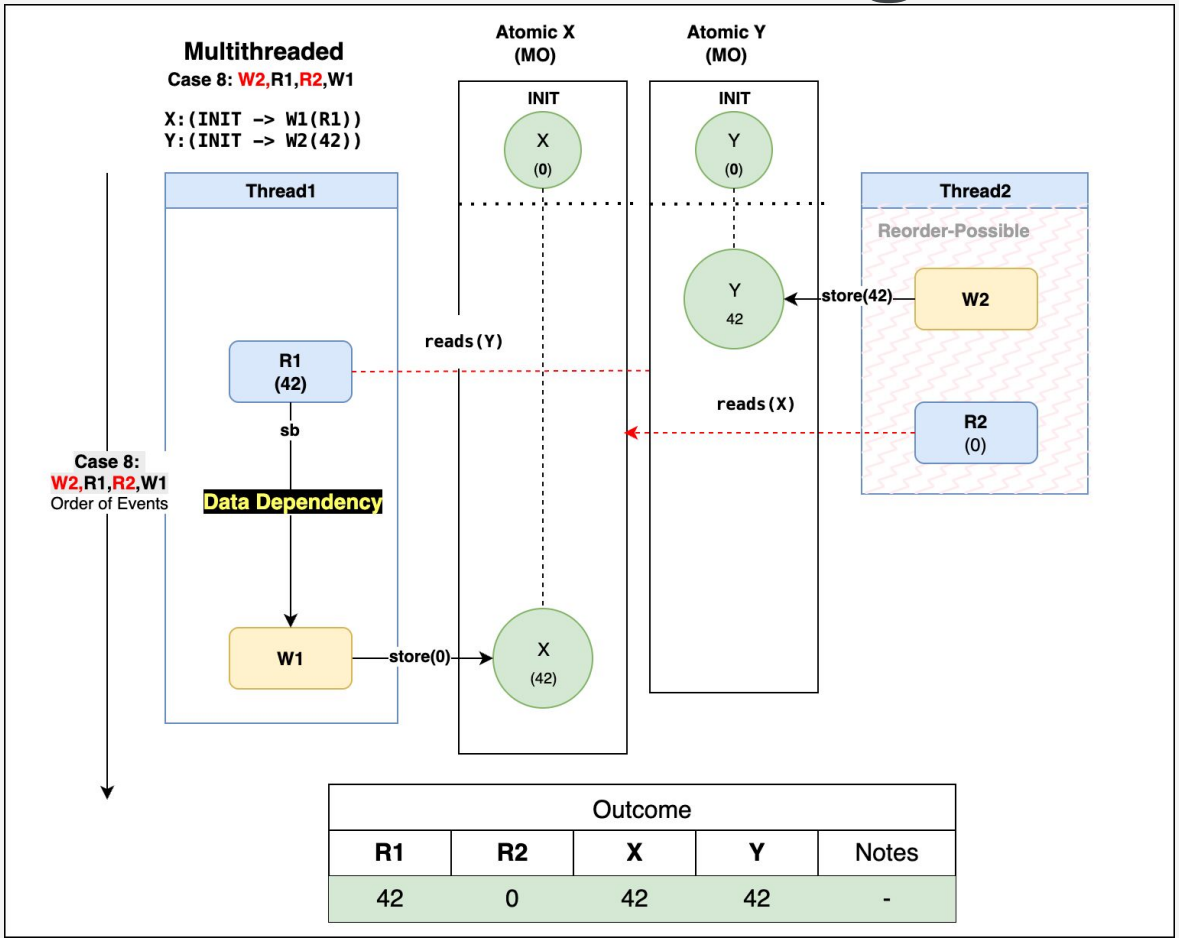
# Relaxed Ordering: Case 7




| Outcome | | | | |
|---|---|---|---|---|
| **R1** | **R2** | **X** | **Y** | Notes |
| 42 | 0 | 42 | 42 | - |

```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

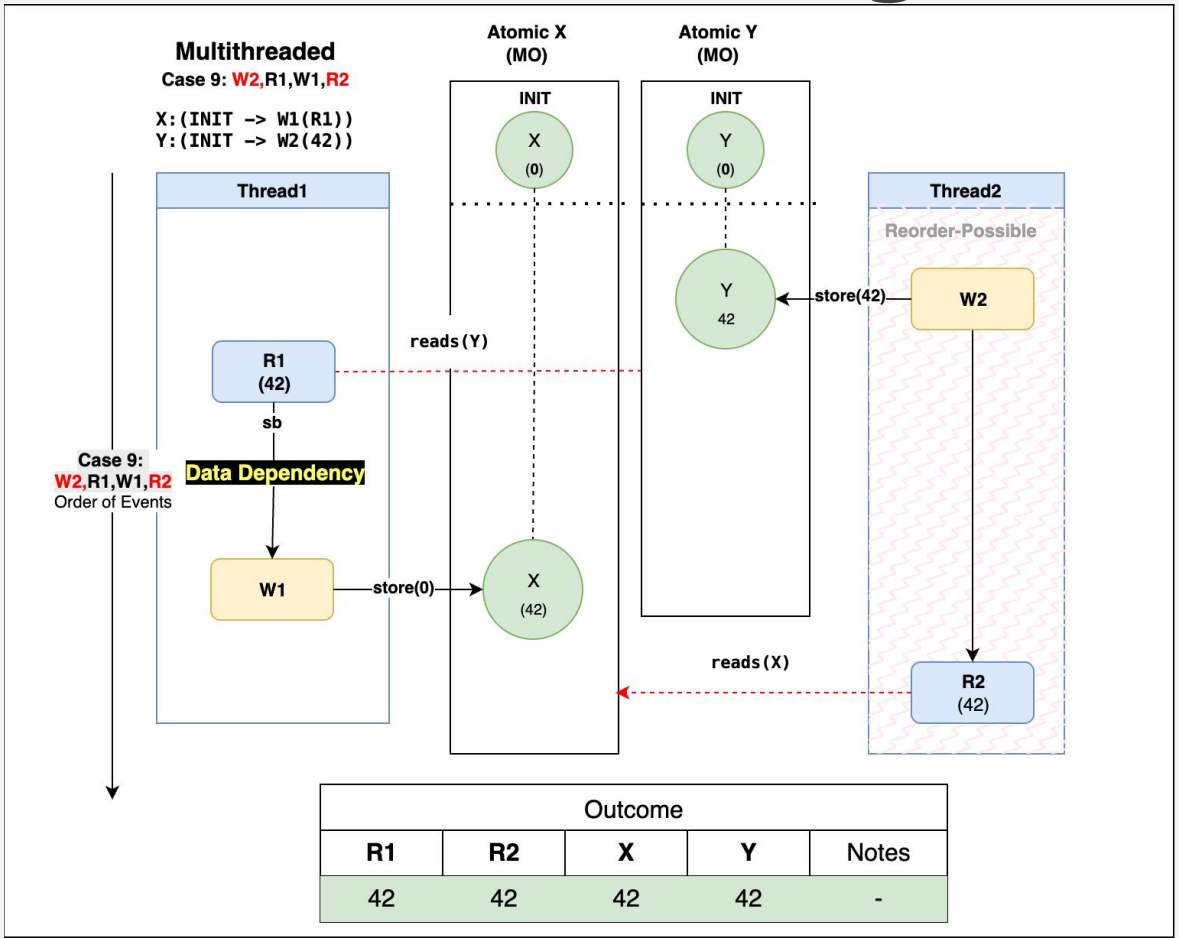
# Relaxed Ordering: Case 8




| Outcome | | | | |
|---|---|---|---|---|
| **R1** | **R2** | **X** | **Y** | Notes |
| 42 | 0 | 42 | 42 | - |

```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

# Relaxed Ordering: Case 9




| Outcome | | | | |
|---|---|---|---|---|
| **R1** | **R2** | **X** | **Y** | Notes |
| 42 | 42 | 42 | 42 | - |

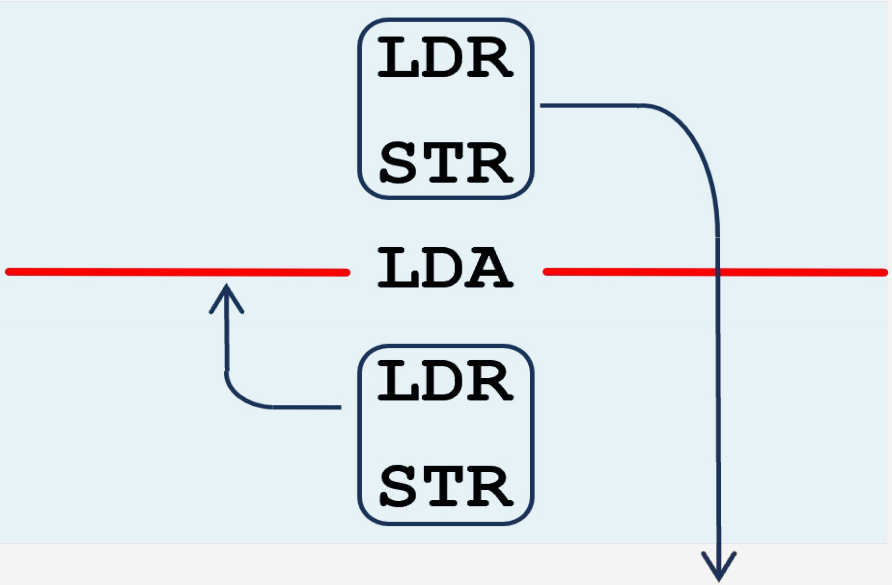
```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

# Relaxed Ordering: All Outcomes

| All Outcomes | | | | | | |
|---|---|---|---|---|---|---|
| | Case | R1 | R2 | X | Y | Notes |
| 1 | R1,W1,R2,W2 | 0 | 0 | 0 | 42 | - |
| 2 | R1,R2,W1,W2 | 0 | 0 | 0 | 42 | |
| 3 | R1,R2,W2,W1 | 0 | 0 | 0 | 42 | |
| 4 | R2,R1,W2,W1 | 0 | 0 | 0 | 42 | |
| 5 | R2,R1,W1,W2 | 0 | 0 | 0 | 42 | |
| 6 | R2,W2,R1,W1 | 42 | 0 | 42 | 42 | |
| 7 | W2,R2,R1,W1 | 42 | 0 | 42 | 42 | W2 & R2 Reordered |
| 8 | W2,R1,R2,W1 | 42 | 0 | 42 | 42 | W2 & R2 Reordered |
| 9 | W2,R1,W1,R2 | 42 | 42 | 42 | 42 | W2 & R2 Reordered |
| 10 | Other orderings where W1 hb R1 are invalid | | | | | |

```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
// Thread 1:
thread::spawn(|| {
    let r1 = Y.load(Ordering::Relaxed); // R1
    X.store(r1, Ordering::Relaxed); // W1
})
// Thread 2:
thread::spawn(|| {
    let r2 = X.load(Ordering::Relaxed); // R2
    Y.store(42, Ordering::Relaxed);  // W2
})
```

# Acquire Ordering

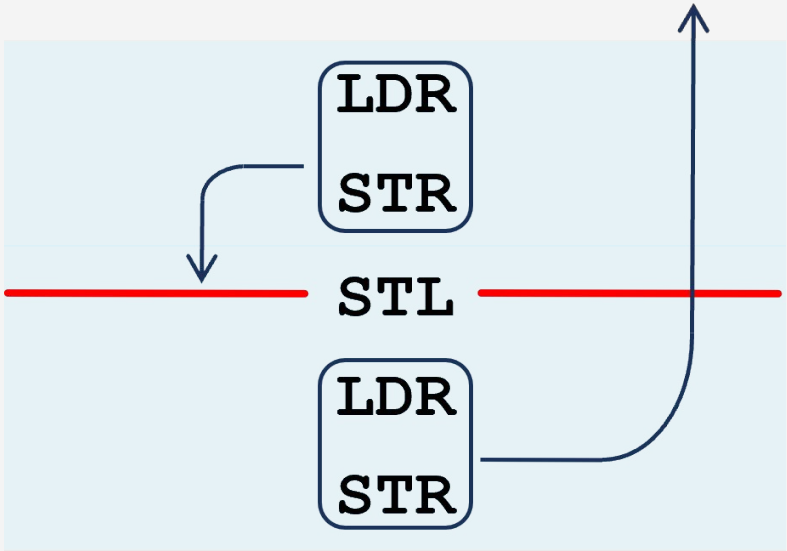



Accesses can cross a barrier in one direction but not the other

- Also known as (Load-Acquire) and can only be applied to loads or atomic operations that perform a load
- Acts as a barrier that prevents instructions that come after it in program order from being lifted before it.
- An Acquire that reads from a Release (or its release sequence) guarantees the acquiring thread can see all prior writes in the releasing thread
- Nearly always used in tandem with Release Ordering to establish “critical region” and enable visibility from that region once a different thread releases it
- **(Diagram)** A Load-Acquire (LDA) instruction ensures that all reads and writes caused by loads and stores that appear in program order after the LDA are observed after the LDA. However, accesses before the LDA are not affected.

https://developer.arm.com/documentation/107565/0101/Memory-system/Memory-barriers/Load-Acquire-and-Store-Release-instructions

# Release Ordering

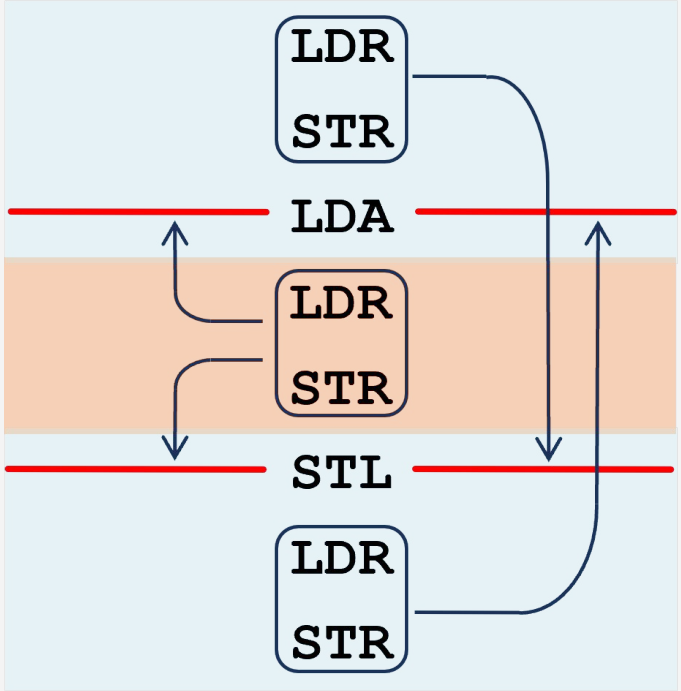



Accesses can cross a barrier in one direction but not the other

- Also known as (Store-Release)
- Acts as a barrier that prevents instructions that come **before** it in program order, from being reordered after it.
- When coupled with an Acquire (or stronger) on the **same atomic**, all previous writes(**atomic and non-atomic**) before the Release will be made available to the coherence system and visible to loads the synchronize-with the release.
- Can only be applied to stores or atomic operations that perform stores (e.g. RMWs)
- **(Diagram)** A Store-Release (STL) instruction ensures that all reads and writes caused by loads and stores that appear in program order before the STL are observed before the STL. However, accesses after the STL are not affected.

https://developer.arm.com/documentation/107565/0101/Memory-system/Memory-barriers/Load-Acquire-and-Store-Release-instructions

# Acquire-Release




Critical Code Section

- Load-Acquire and Store-Release instructions can be combined in a pair to protect a critical section of code. Combining these instructions ensures:
  - Accesses that are made within the critical code section are not reordered outside of the critical section.
  - Accesses inside the critical code section are not affected, and can be reordered

https://doc.rust-lang.org/nomicon/races.html
https://developer.arm.com/documentation/107565/0101/Memory-system/Memory-barriers/Load-Acquire-and-Store-Release-instructions

# AcqRel Ordering

- Has the effects of both Acquire and Release together: For loads it uses Acquire ordering. For stores it uses the Release ordering.

- This ordering is only applicable for operations that combine both loads and stores (load-with-store operations)

- For conditional CAS e.g. `compare_exchange` it is possible that the operation ends up not performing any store and hence fails resulting in a `Relaxed` load on failure.

- https://doc.rust-lang.org/std/sync/atomic/enum.Ordering.html

# SeqCst Ordering

- Each thread's SeqCst operations appear in sequential order within the global order typically called S
- This means that in each concrete run, there must exist exist a single total order S over all SeqCst operations (as well as SeqCst fences) that every thread can agree on.
  - Non SeqCst operations do not participate in S
- S must also be consistent with atomic modification order AND happens-before

**Strongly Happens-Before (shb)**
S ensures it follows **strongly-happens-before**

Evaluation A strongly happens-before evaluation B ***if either***:

- A is sequenced-before B (in the same thread).
- A synchronizes-with B (e.g., release → acquire on the same atomic
- X such that A shb X and X shb B (Transitivity)

- https://doc.rust-lang.org/std/sync/atomic/enum.Ordering.html
- https://en.cppreference.com/w/cpp/atomic/memory_order.html

# SeqCst Single Total Order

SeqCst requires a **single total order** S over {A,B,C,D} that preserves **each thread's program order**, i.e. A < B and C < D must hold in S.

1. A < B < C < D
2. A < C < B < D
3. A < C < D < B
4. C < D < A < B
5. C < A < D < B
6. C < A < B < D

Note: Any of the 6 Global orders can be observed across different executions.

**Benefit [1]**

1. **Happens-Before intuition satisfied**
2. **Correctness Maintained**

```
1 static X: AtomicI32 = AtomicI32::new(0);
2 static Y: AtomicI32 = AtomicI32::new(0);
3
4 // Thread 1
5 thread::spawn(|| {
6     X.store(1, Ordering::SeqCst); // A
7     let r1 = Y.load(Ordering::SeqCst) // B
8 })
9
10 // Thread 2
11 thread::spawn(|| {
12     Y.store(1, Ordering::SeqCst) // C
13     let r2 = X.load(Ordering::SeqCst) // D
14 })
```

[^1]-https://www.cs.cmu.edu/afs/cs/academic/class/15740-s18/www/lectures/08-09-consistency.pdf
[^2]- (Lamport, 1979)

# SeqCst Ordering: Pitfalls

## Repairing Sequential Consistency in C/C++11


Ori Lahav
MPI-SWS, Germany *
orilahav@mpi-sws.org

Viktor Vafeiadis
MPI-SWS, Germany *
viktor@mpi-sws.org

Jeehoon Kang
Seoul National University, Korea
jeehoon.kang@sf.snu.ac.kr

Chung-Kil Hur
Seoul National University, Korea
gil.hur@sf.snu.ac.kr

Derek Dreyer
MPI-SWS, Germany *
dreyer@mpi-sws.org



levels, from very weak consistency ("relaxed") to strong, sequential consistency ("SC"). Unfortunately, as we observe in this paper, the semantics of SC atomic accesses in C/C++11, as well as in all proposed strengthenings of the semantics, is flawed, in that (contrary to previously published results) both suggested compilation schemes to the Power architecture are unsound. We propose a model, called RC11 (for Repaired C11), with a better semantics for SC accesses that restores the soundness of the compilation schemes to Power, maintains the DRF-SC guarantee, and provides stronger, more useful, guarantees to SC fences. In addition, we formally prove, for the first time, the correctness of the proposed stronger compilation schemes to Power that preserve load-to-store ordering and avoid "out-of-thin-air" reads.


https://plv.mpi-sws.org/scfix/paper.pdf

# SeqCst Ordering: Pitfalls

```rust
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);

// Thread 1:
thread::spawn(|| {
    X.store(1, Ordering::SeqCst); // A
    Y.store(1, Ordering::Release); // B
})

// Thread 2:
thread::spawn(|| {
    let r1 = Y.fetch_add(1, Ordering::SeqCst); // C
    let r2 = Y.load(Ordering::Relaxed); // D
}

// Thread 3:
thread::spawn(|| {
    Y.store(3, Ordering::SeqCst); // E
    let r3 = X.load(Ordering::SeqCst); // F
}
```

is allowed to produce r1 == 1 && r2 == 3 && r3 == 0, where A happens-before C, but C precedes A in the single total order C-E-F-A of memory_order_seq_cst

- Can be quite tricky to debug
  - Weaker orderings can interpose in a way that makes SeqCst counter-intuitive [1]
  - SeqCst operations are reorderable with respect to other atomics performed in the same thread[1]

- SeqCst Total Order is not guaranteed to be consistent with **happens-before** 🥲. This allows for more efficient Acquire/Release on some CPUs but produces surprising results when you starting mixing Acquire/Release and SeqCst[1]

- The main problem arises in programs that mix SC and non-SC accesses to the same location. Although not common, such mixing is freely permitted by the C11 standard, and has legitimate uses, e.g., as a way of enabling faster (non-SC) reads from an otherwise quite strongly synchronized data structure[2]

[^1] - https://en.cppreference.com/w/cpp/atomic/memory_order.html
[^2] - https://plv.mpi-sws.org/scfix/paper.pdf

# Atomic Methods: Operations

1. **Plain Operations**
2. **Read Modify Write (RMW) Operations**
    a. **Conditional**
    b. **Unconditional**

https://en.wikipedia.org/wiki/Compare-and-swap

# Atomic Methods: Plain (Non-RMW) Operations

- `pub fn load(&self, order: Ordering) -> usize`
- `pub fn store(&self, val: usize, order: Ordering)`

https://doc.rust-lang.org/std/primitive.usize.html

# Atomic Methods: Unconditional RMW

**Below are Unconditional “(Always Succeed)” RMW methods for `AtomicUsize`**

**Exchange**

- `pub fn swap(&self, val: usize, order: Ordering) -> usize`
  - writes a new value and returns the old one

**Fetch-Ops**

- `pub fn fetch_{add,sub,and,nand,or,xor,max,min}(&self, val: usize, order: Ordering) -> usize`
  - compute new value atomically and store it, then return the old value.
  - They always complete the update (there is no “failure” result to the caller).
  - Internally they may use a single instruction, LL/SC (Load-Linked / Store-Conditional), or a CAS loop, but the API guarantees the update occurs exactly once.

https://en.wikipedia.org/wiki/Compare-and-swap

# Atomic Methods: Conditional RMWs

**Below are CAS methods for AtomicUsize**

- 
```
pub fn compare_exchange{_weak}(
    &self,
    current: usize,
    new: usize,
    success: Ordering,
    failure: Ordering,
) -> Result<usize, usize>
```

- CAS compares the contents of a memory location with a given (current) value.
  - **success**: If self==current it modifies the contents of that memory location to new returning Ok(prev)
    - ***Ordering***: Release/AcqRel/SeqCst
  - **failure**: If self!=current, or "spurious failure" on **compare_exchange_weak**, it returns Err(actual) where actual is the value it found.
    - ***Ordering***: Acquire/Relaxed/SeqCst
- They are RMWs on success, and behave like an atomic load on failure

- **compare_exchange**(strong CAS) is guaranteed to succeed and exchange if self==current
- **compare_exchange_weak**(weak CAS) allows the exchange to fail even if self==current
  - Allowing it to return Err(actual) where actual==current
  - This behavior is architecture dependent
  - Typically used in retry-loops to to retry on Err(actual)to mitigate this
  - On x86 weak and strong CAS compile to the same instructions***

https://doc.rust-lang.org/std/sync/atomic/struct.AtomicUsize.html

# Weak & Strong Architectures

**RISC-V memory ordering models**

- **WMO**: Weak memory order (default)
- **TSO**: Total store order (only supported with the Ztso extension)

**SPARC memory ordering modes**

- **TSO**: Total store order (default)
- **RMO**: Relaxed-memory order (not supported on recent CPUs)
- **PSO**: Partial store order (not supported on recent CPUs)

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

**Memory ordering in some architectures**[5][16]

| Type | Alpha | ARMv7 | MIPS | RISC-V | | PA-RISC | POWER | SPARC | | | x86 [a] | AMD64 | IA-64 | z/Architecture |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | WMO | TSO | | | RMO | PSO | TSO | | | | |
| Loads can be reordered after loads | Y | Y | depend on implementation | Y | | Y | Y | Y | | | | | Y | |
| Loads can be reordered after stores | Y | Y | | Y | | Y | Y | Y | | | | | Y | |
| Stores can be reordered after stores | Y | Y | | Y | | Y | Y | Y | Y | | | | Y | |
| Stores can be reordered after loads | Y | Y | | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| Atomic can be reordered with loads | Y | Y | | Y | | | Y | Y | | | | | Y | |
| Atomic can be reordered with stores | Y | Y | | Y | | | Y | Y | Y | | | | Y | |
| Dependent loads can be reordered | Y | | | | | | | | | | | | | |
| Incoherent instruction cache/pipeline | Y | Y | | Y | Y | | Y | Y | Y | Y | Y | | Y | |

a. ^ This column indicates the behaviour of the vast majority of x86 processors. Some rare specialised x86 processors (IDT WinChip manufactured around 1998) may have weaker 'oostore' memory ordering.[17]

1. One-Shot Lock
2. MPSC Queues (Channels)

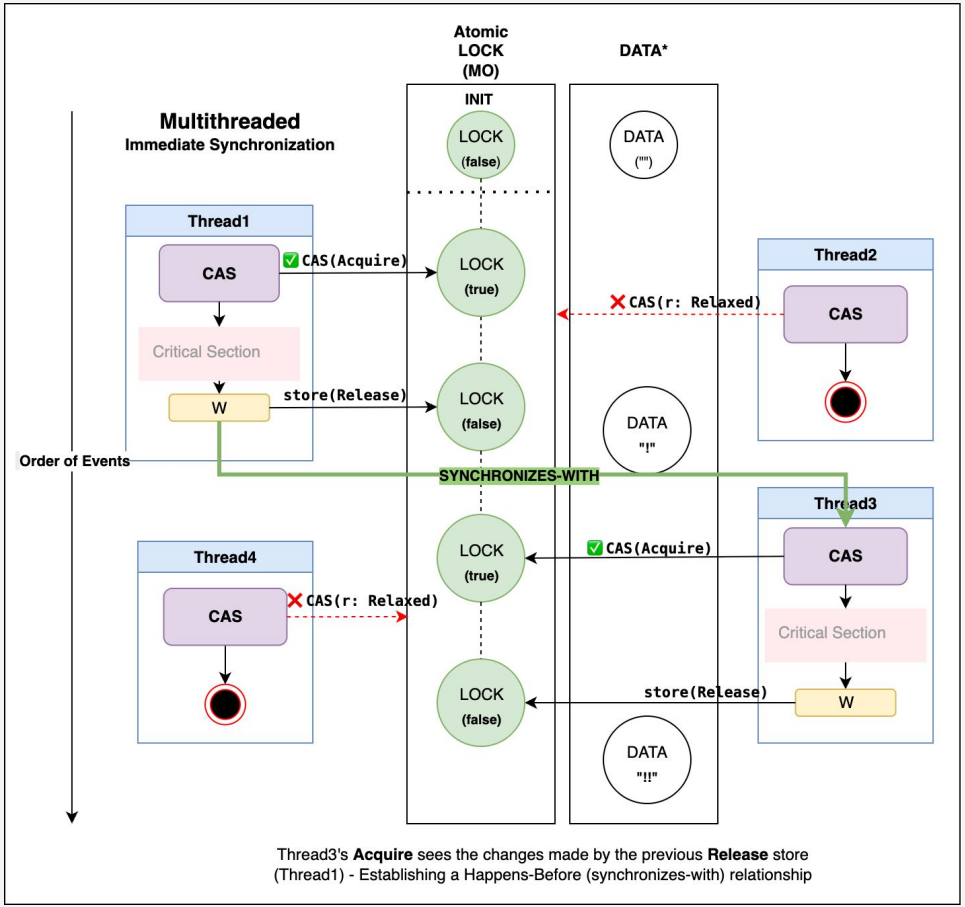
# Examples

# Acquire/Release Example: One-Shot Lock

```rust
static mut DATA: String = String::new();
static LOCK: AtomicBool = AtomicBool::new(false);

fn f() {
    if LOCK.compare_exchange(false, true, Acquire, Relaxed).is_ok() {
        // Safety: We hold the exclusive lock, so nothing else is accessing DATA.
         unsafe { DATA.push('!') };
         LOCK.store(false, Release);
     }
}

fn main() {
    thread::scope(|s| {
    for _ in 0..100 {
       s.spawn(f);
       }
   });
}
```

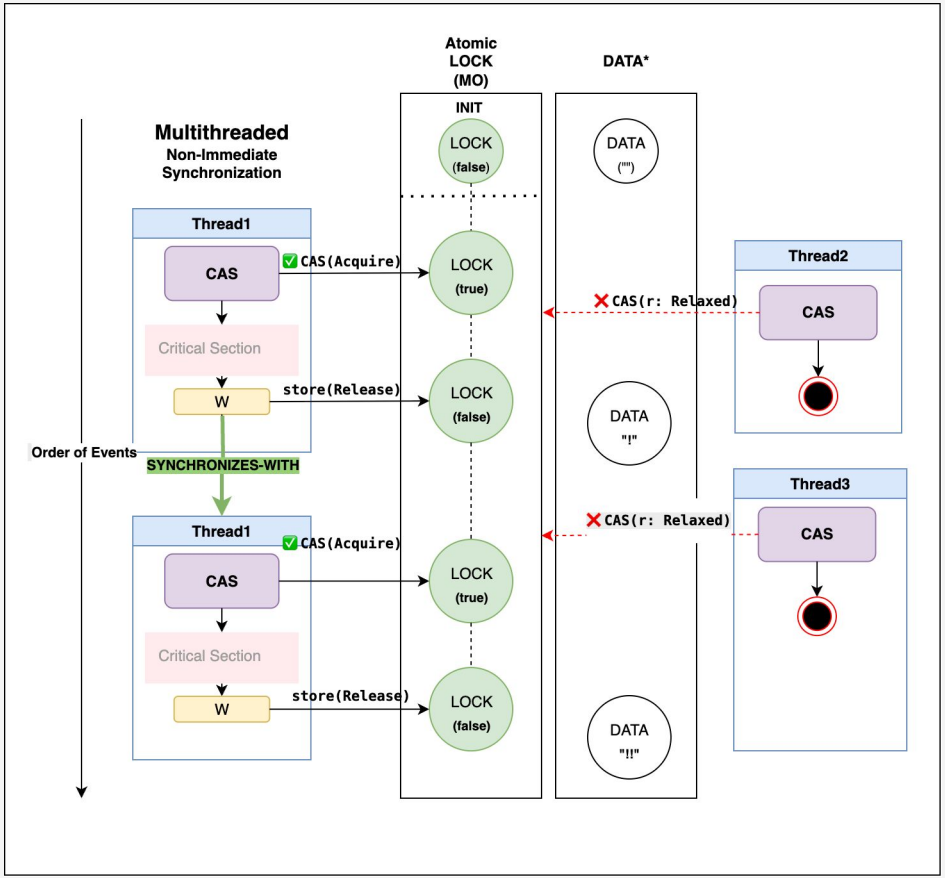
https://marabos.nl/atomics/memory-ordering.html

# Acquire/Release Example: One-Shot Lock




```rust
static mut DATA: String = String::new();
static LOCK: AtomicBool = AtomicBool::new(false);

fn f() {
    if LOCK.compare_exchange(false, true, Acquire, Relaxed).is_ok() {
        // Safety: We hold the exclusive lock, so nothing else is accessing DATA.
        unsafe { DATA.push('!') };
        LOCK.store(false, Release);
    }
}

fn main() {
    thread::scope(|s| {
    for _ in 0..100 {
       s.spawn(f);
       }
   });
}
```

# Acquire/Release Example: One-Shot Lock




```rust
static mut DATA: String = String::new();
static LOCK: AtomicBool = AtomicBool::new(false);

fn f() {
    if LOCK.compare_exchange(false, true, Acquire, Relaxed).is_ok() {
        // Safety: We hold the exclusive lock, so nothing else is accessing DATA.
        unsafe { DATA.push('!') };
        LOCK.store(false, Release);
    }
}

fn main() {
    thread::scope(|s| {
    for _ in 0..100 {
       s.spawn(f);
       }
   });
}
```

# MPSC Queues (Channels)

- FIFO Multi-producer, Single-Consumer Queues provide message-based communication over channels
- Available in `std::sync::mpsc::{channel, sync_channel}` module.
- The user uses `Sender` and `Receiver` types to interact with the channel

```rust
use std::thread;
use std::sync::mpsc::channel;

// Create a simple streaming channel
let (tx, rx): (Sender<i32>, Receiver<i32>) = mpsc::channel();

thread::spawn(move || {
    tx.send(10).unwrap();
});

assert_eq!(rx.recv().unwrap(), 10);

```

https://doc.rust-lang.org/std/sync/mpsc/index.html

# Channel: Array Flavor (Bounded)

```rust
struct Slot<T> {
    /// The head of the channel.
    stamp: AtomicUsize,
    /// The message in this slot.
    msg: UnsafeCell<MaybeUninit<T>>,
}

pub struct Channel<T> {
    head: CachePadded<AtomicUsize>,
    tail: CachePadded<AtomicUsize>,
    /// The buffer holding slots.
    buffer: Box<[Slot<T>]>,
    /// The channel capacity.
    cap: usize,
    /// A stamp with the value of `{ lap: 1, mark: 0, index: 0 }`.
    one_lap: usize,
    /// If this bit is set in the tail, that means the channel is disconnected.
    mark_bit: usize,
    /// Senders waiting while the channel is full.
    senders: SyncWaker,
    /// Receivers waiting while the channel is empty and not disconnected.
    receivers: SyncWaker,
}

```

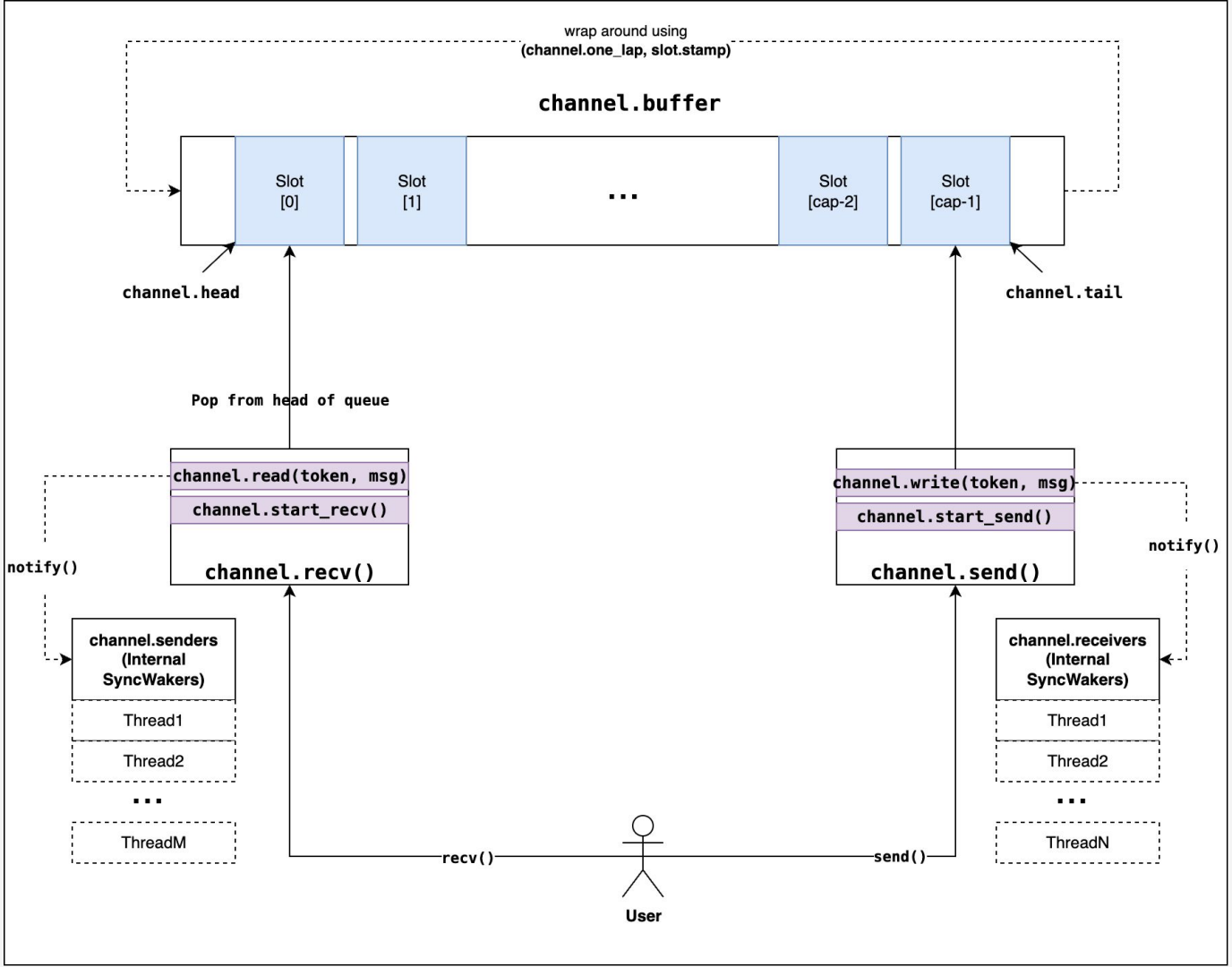
https://doc.rust-lang.org/std/sync/mpsc/index.html

# Channel: Array Flavor (Bounded)

```rust
impl<T> Channel<T> {
    pub(crate) fn recv(&self, deadline: Option<Instant>) -> Result<T, _>
    pub(crate) fn send(&self,msg: T,deadline: Option<Instant>) -> Result<(), _>

    pub unsafe fn read(&self, token: &mut Token) -> Result<T, ()>
    pub unsafe fn write(&self, token: &mut Token, msg: T) -> Result<(), T>

    fn start_send(&self, token: &mut Token) -> bool
    fn start_recv(&self, token: &mut Token) -> bool

    // Other helpers
    pub(crate) fn is_empty(&self) -> bool
    pub(crate) fn is_disconnected(&self) -> bool
    pub(crate) fn is_full(&self) -> bool {
}
```

https://doc.rust-lang.org/std/sync/mpsc/index.html

# Channel: Bounded




https://doc.rust-lang.org/std/sync/mpsc/index.html

# Channel: start_send

**Goal**: Reserve a slot to write into or decide if channel is full/disconnected without writing the message

**AtomicOps**

1. Load `tail` with `Relaxed`
   `// BEGIN LOOP`
2. Load `stamp` with `Acquire`
3. CAS `tail` with `new_tail` using `SeqCst`
   a. OK: Reserve `slot`
   b. Err: Backoff + reload `tail` with `Relaxed`
4. Establish `atomic::fence(SeqCst)`
5. Load `head` with `Relaxed`
   a. If Channel Full: Return
   b. Backoff + reload `tail` with `Relaxed`

https://doc.rust-lang.org/std/sync/mpsc/index.html

```rust
fn start_send(&self, token: &mut Token) -> bool {
        let mut tail = self.tail.load(Ordering::Relaxed);

        loop {
            // 1. Check if the channel is disconnected.
            // 2. Deconstruct the tail and get the `index`
            // 3. Inspect the corresponding slot.
            let slot = unsafe { self.buffer.get_unchecked(index) };
            let stamp = slot.stamp.load(Ordering::Acquire);

            // If the tail and the stamp match, we may attempt to push.
            if tail == stamp {
                let new_tail = ...// calculate new tail based on if index is < cap (if not wrap around)

                // Try moving the tail with global visibility.
                match self.tail.compare_exchange_weak(
                    tail,
                    new_tail,
                    Ordering::SeqCst, // maintains head/tail movements in a single total order
                    Ordering::Relaxed,
                ) {
                    Ok(_) => {
                        // Prepare the token for the follow-up call to `write`.
                        token.array.slot = slot as *const Slot<T> as *const u8;
                        token.array.stamp = tail + 1;
                        return true // slot has been reserved
                    }
                    Err(_) => {
                        backoff.spin_light();
                        tail = self.tail.load(Ordering::Relaxed);
                    }
                }
            } else if stamp.wrapping_add(self.one_lap) == tail + 1 {
                atomic::fence(Ordering::SeqCst);
                let head = self.head.load(Ordering::Relaxed);

                // If the head lags one lap behind the tail as well...
                if head.wrapping_add(self.one_lap) == tail {
                    // ...then the channel is full.
                    return false;
                }

                backoff.spin_light();
                tail = self.tail.load(Ordering::Relaxed);
            } else {
                // Snooze because we need to wait for the stamp to get updated.
                backoff.spin_heavy();
                tail = self.tail.load(Ordering::Relaxed);
            }
        }
    }
```

# Channel: write()

**Goal**: Goal: after reserving, place msg into the chosen slot and make it visible.

```rust
pub unsafe fn write(&self, token: &mut Token, msg: T) -> Result<(), T>
{
        // If there is no slot, the channel is disconnected.
        if token.array.slot.is_null() {
            return Err(msg);
        }

        let slot: &Slot<T> = &*(token.array.slot as *const Slot<T>);

        // Write the message into the slot and update the stamp.
        slot.msg.get().write(MaybeUninit::new(msg));
        slot.stamp.store(token.array.stamp, Ordering::Release);

        // Wake a sleeping receiver.
        self.receivers.notify();
        Ok(())
    }

```

1. If the supplied token.array.slot is null, the channel is disconnected
2. Otherwise:
   a. Write the payload to slot.msg (plain store into the UnsafeCell).
   b. Store the prepared token.array.stamp into slot.stamp with **Release**.
   c. Notify a receiver to awaken

**Use of Release Semantics**

-Ensures that the consumer that later reads slot.stamp with Acquire will see the written slot.msg contents before it proceeds.

https://doc.rust-lang.org/std/sync/mpsc/index.html

# Channel: start_recv

**Goal**: Tries to reserve the next readable slot for the consumer by advancing the global head **iff** the target's slot stamp shows the producer is done publishing to that slot

**AtomicOps**

1. Load head with Relaxed
   // BEGIN LOOP
2. Load stamp with Acquire
3. CAS head with new_head using SeqCst
   a. OK: Reserve slot
   b. Err: Backoff + reload tail with Relaxed
4. Establish atomic::fence(SeqCst)
5. Load head with `Relaxed
   a. If Channel Full: Return
   b. Backoff + reload tail with Relaxed

https://doc.rust-lang.org/std/sync/mpsc/index.html

```rust
fn start_recv(&self, token: &mut Token) -> bool {
        let mut head = self.head.load(Ordering::Relaxed);

        loop {
            // Deconstruct the head to get the index
            // Inspect the corresponding slot.
            let slot = unsafe { self.buffer.get_unchecked(index) };
            let stamp = slot.stamp.load(Ordering::Acquire);

            // If the stamp is ahead of the head by 1, we may attempt to pop.
            if head + 1 == stamp {
                let new_head = ...//

                // Try moving the head.
                match self.head.compare_exchange_weak(
                    head,
                    new_head,
                    Ordering::SeqCst,
                    Ordering::Relaxed,
                ) {
                    Ok(_) => {
                        // Prepare the token for the follow-up call to `read`.
                        token.array.slot = slot as *const Slot<T> as *const u8;
                        token.array.stamp = head.wrapping_add(self.one_lap);
                        return true;
                    }
                    Err(_) => {
                        backoff.spin_light();
                        head = self.head.load(Ordering::Relaxed);
                    }
                }
            } else if stamp == head {
                atomic::fence(Ordering::SeqCst);
                let tail = self.tail.load(Ordering::Relaxed);

                // If the tail equals the head, that means the channel is empty.
                // If the channel is disconnected...

                backoff.spin_light();
                head = self.head.load(Ordering::Relaxed);
            } else {
                // Snooze because we need to wait for the stamp to get updated.
                backoff.spin_heavy();
                head = self.head.load(Ordering::Relaxed);
            }
        }
    }
```

# Channel: read

**Goal**: loads the msg that can be returned to the Reader

**AtomicOps**

1. Publish the stamp.store, with the associated token.array.stamp using Release Ordering
2. This publishes frees up the slot allowing later start_send to Acquire slot.stamp.store and reserve a slot

```rust
pub unsafe fn read(&self, token: &mut Token) -> Result<T, ()> {
    if token.array.slot.is_null() {
        // The channel is disconnected.
        return Err(());
    }

    let slot: &Slot<T> = &*(token.array.slot as *const
Slot<T>);
    // Read the message from the slot and update the stamp.
    let msg = slot.msg.get().read().assume_init();
    slot.stamp.store(token.array.stamp, Ordering::Release);

    // Wake a sleeping sender.
    self.senders.notify();
    Ok(msg)
}
```

https://doc.rust-lang.org/std/sync/mpsc/index.html

# Channel: send()

**Goal**: blocking (or timed) send that repeats the (start_send → write) sequence or parks the thread when full.

```rust
pub unsafe fn write(&self, token: &mut Token, msg: T) -> Result<(), T>
{
        // If there is no slot, the channel is disconnected.
        if token.array.slot.is_null() {
            return Err(msg);
        }

        let slot: &Slot<T> = &*(token.array.slot as *const Slot<T>);

        // Write the message into the slot and update the stamp.
        slot.msg.get().write(MaybeUninit::new(msg));
        slot.stamp.store(token.array.stamp, Ordering::Release);

        // Wake a sleeping receiver.
        self.receivers.notify();
        Ok(())
    }

```

1. If start_send reserves a slot, we obtain the token back and call write to write the msg
2. We then enter the blocking path using thread-local Context.
3. 
4. We then enter the blocking path using thread-local Context:
   a. .
   b. Store the prepared token.array.stamp into slot.stamp with **Release**.
   c. Notify a receiver to awaken

**Use of Release Semantics**

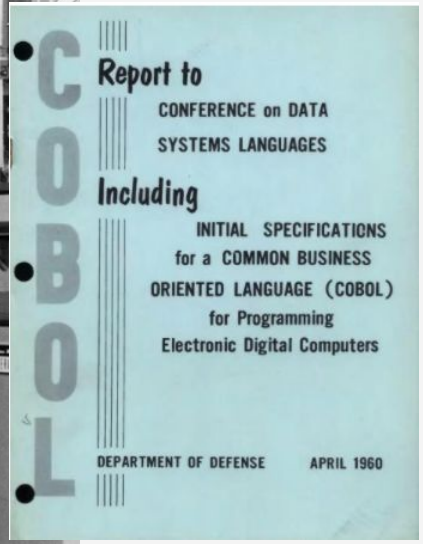
-Ensures that the consumer that later reads slot.stamp with Acquire will see the written slot.msg contents before it proceeds.

https://doc.rust-lang.org/std/sync/mpsc/index.html

# Thanks!

LinkedIn/Github: **@martinomburajr**

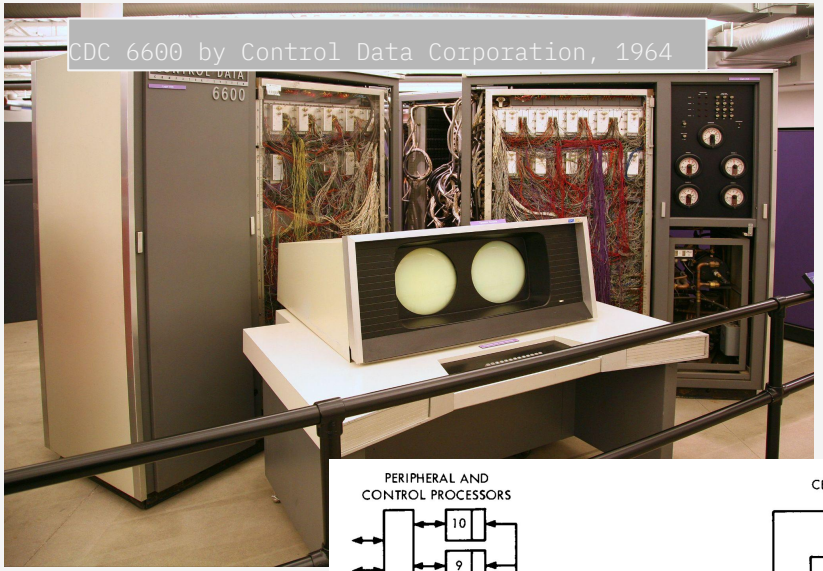
# 1950s-Bare Metal Era


COBOL statements have prose syntax such as `MOVE x TO y`,

- 1st Compilers (A-0 & FlowMatic) by **Grace Hopper**, first linker and loaders also developed *(1951)*
- Introduction of English "words" into Programming
- **COBOL** by CODASYL, paves way for using English to represent Business Logic partly based on Hopper's Flow-Matic language *(1959)*.
- As of 2014 - 90% of financial transactions still processed by COBOL

https://www.historyofinformation.com/detail.php?id=778
https://en.wikipedia.org/wiki/COBOL


CDC 6600 by Control Data Corporation, 1964

# 1960s CPU Advancements

- Introduction of CPU Pipelining and Out of Order Execution.
- Introduction of pipelining to exploit ILP e.g. CDC6600 *(1964)*
- Introduction of algorithms that added dynamic scheduling. **Decoupling actual execution order from program order** (Tomasulo, 1967).

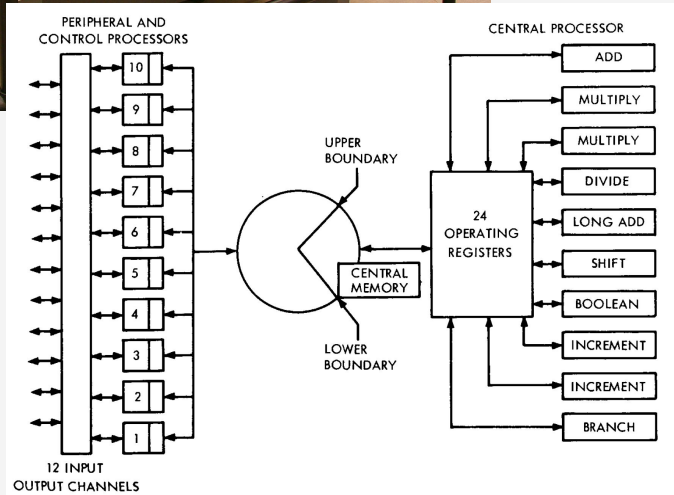


Figure 3. Block Diagram of 6600.

https://courses.cs.washington.edu/courses/cse549/14sp/Thornton.1964.AFIPS.pdf

# 1970s C, Unix

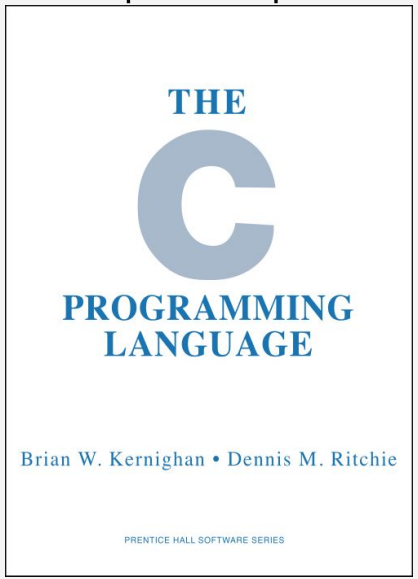
Kernighan and Ritchie working on a PDP-11

```
$ simh-pdp11 boot.ini

PDP-11 simulator V3.10-0
Disabling XQ
#0=unix

UNIX/3.0.1: unixhptm
real mem = 262144 bytes
avail mem = 195776 bytes
unix
single-user
# init 2
# process accounting started
errdemon started
cron started
multi-user
type ctrl-d

login: root
UNIX Release 3.0
# uname -a
unix unix 3.0.1 hptm
#
```

Unix System III running on a PDP-11 simulator

UNIX®

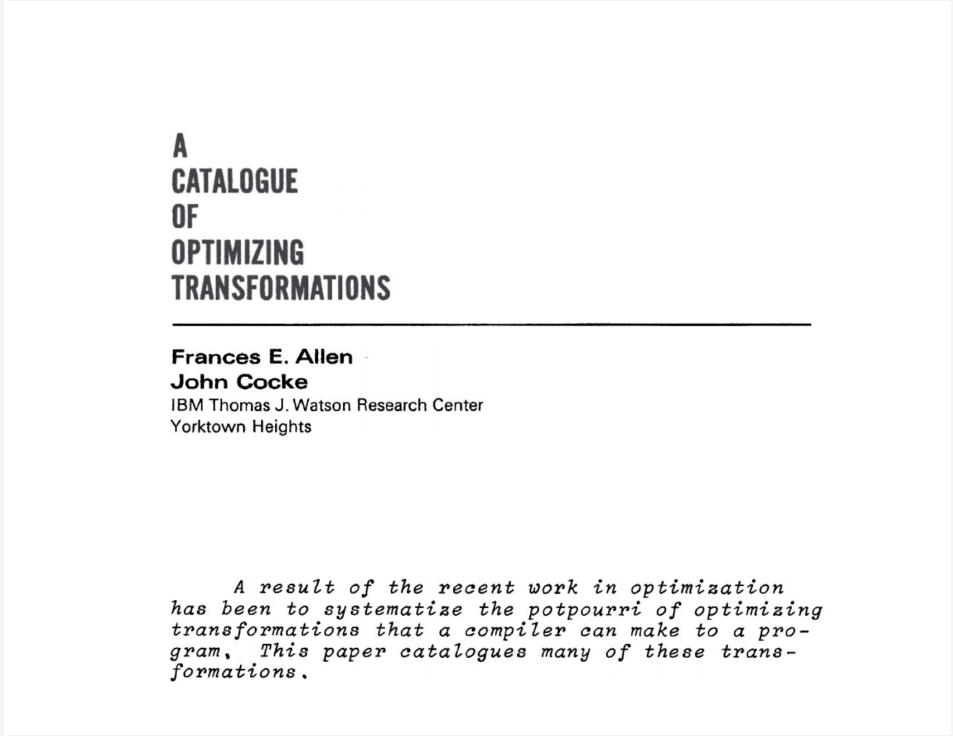
An Open Group Standard




- Creation of C and its use in Unix showing that HL languages can be portable to target many languages (1973)
- Portability and a thin abstraction over hardware let compilers take on more responsibility without hiding performance.

https://www.historyofinformation.com/detail.php?id=872

# 1970s Compiler Optimizations



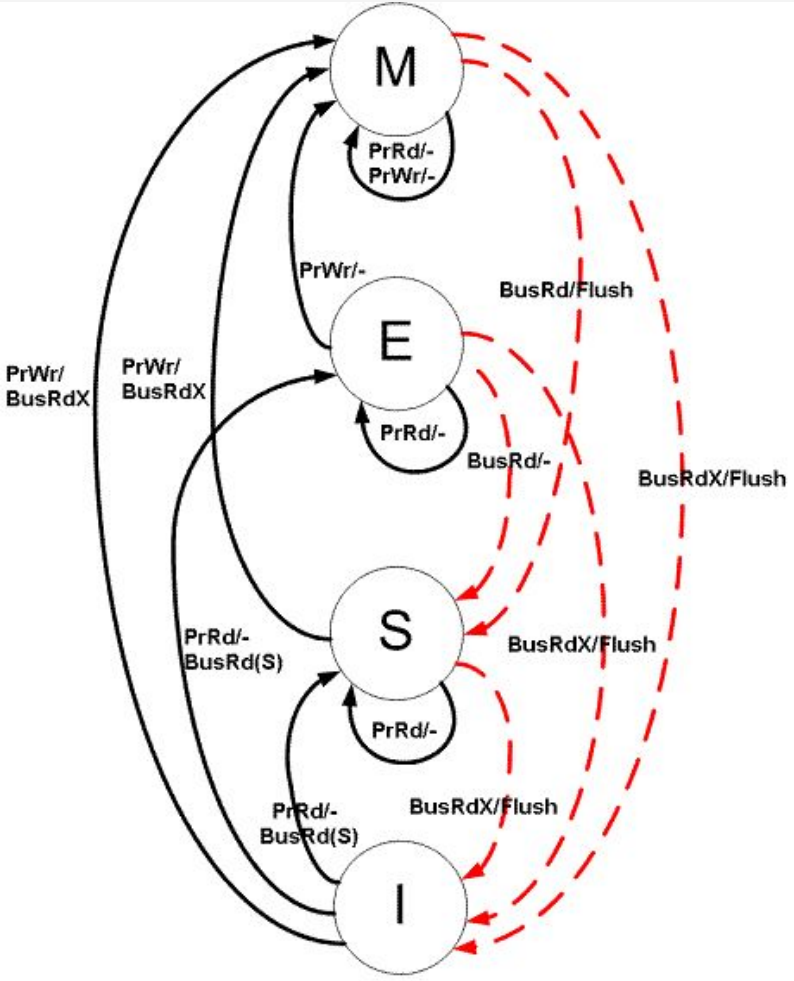
A
CATALOGUE
OF
OPTIMIZING
TRANSFORMATIONS

Frances E. Allen
John Cocke
IBM Thomas J. Watson Research Center
Yorktown Heights

*A result of the recent work in optimization has been to systematize the potpourri of optimizing transformations that a compiler can make to a program. This paper catalogues many of these transformations.*

- Work on systematized program transformations e.g. (common subexpression elimination, loop optimizations, dead-code elimination, etc.) (1971)
- The compiler's freedom to reorder and transform code for speed sets up tension with the programmer's intuitive order of operations.

https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf



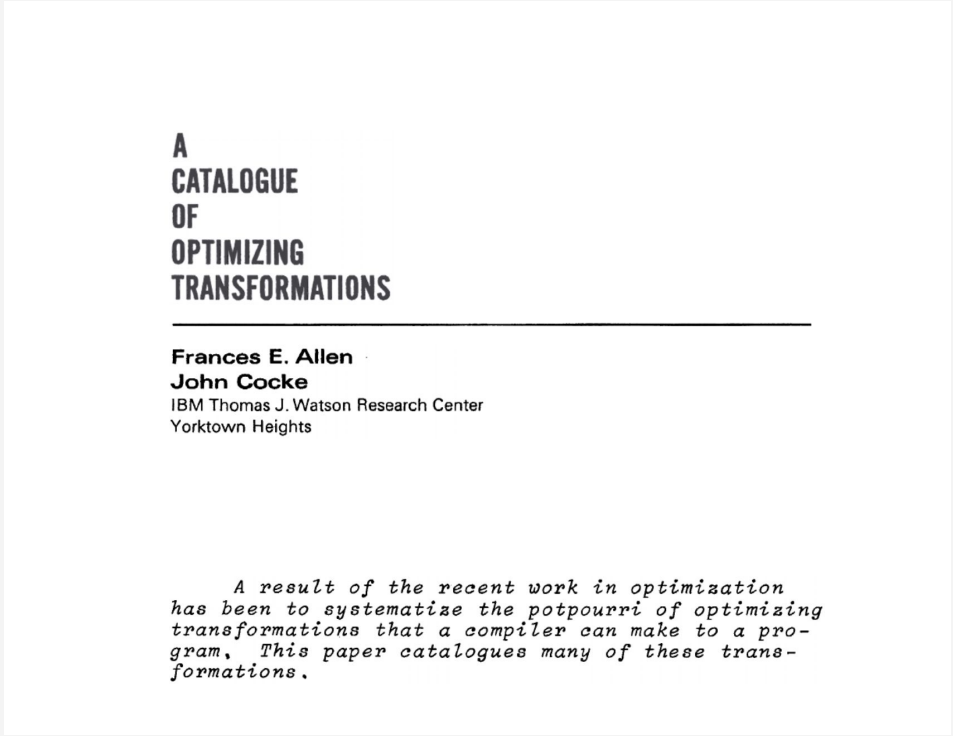

# 1980s Cache Coherence

- Improved private cache performance - creating the Coherence problem
- Multibus Framework - Cache Snooping (Goodman, 1983) ideas take shape
- MESI protocol (1984) introduces invalidate-based snooping techniques

***Coherence protocols ensure a consistent per-location order of writes (coherence), but not a single global order of all memory events.***

https://ftp.cs.wisc.edu/galileo/papers/retro-goodman.pdf
https://www.cs.cmu.edu/afs/cs/academic/class/15740-s18/www/lectures/08-09-consistency.pdf

# 1970s-2010s - Memory Model Formalization



A
CATALOGUE
OF
OPTIMIZING
TRANSFORMATIONS

Frances E. Allen
John Cocke
IBM Thomas J. Watson Research Center
Yorktown Heights

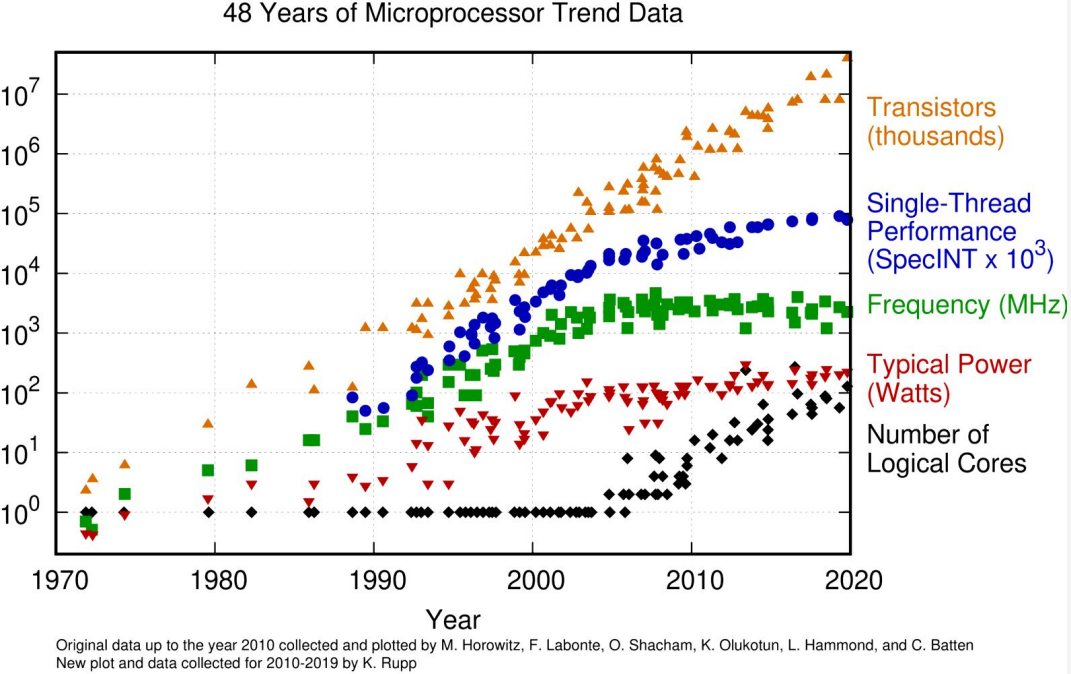
*A result of the recent work in optimization has been to systematize the potpourri of optimizing transformations that a compiler can make to a program. This paper catalogues many of these transformations.*

- Sequential consistency (SC)—a simple mental model where operations appear in one interleaving consistent with each thread's program order (Lamport, 1979)
- The 1990s more explicit work on weak/relaxed models and the synchronizations that recover SC where needed.

***Why this matters: Hardware gives performance by reordering; the memory model specifies what reorderings are allowed and how synchronization constrains them***

lamport.azurewebsites.net
https://www.cl.cam.ac.uk/~pes20/weakmemory/cacm.pdf

# 2000s+ Multicore Adoption




- Focus shifts from from “powerful” cores to more cores (Sutter, 2004)
- 1990s Java Memory Model proved inadequate (2004)
- C++11/C11 added foundations for atomic operations and a formal model (2008)

***The language became the contract between compilers and programmers, mapping source-level synchronization to constraints on compiler/hardware reorderings.***

https://fiveable.me/advanced-computer-architecture/unit-10/multicore-processor-design-principles/study-guide/bfOCvUhIgHtpMPtm

# Key Definitions: Partial Order

A partial order on a set is an arrangement such that:

1. For certain elements the following properties hold:

**Properties:**

1. **Reflexive**: a ≤ a
2. **Antisymmetric**: if a ≤ b and b ≤ a, then a = b
3. **Transitive**: if a ≤ b and b ≤ c, then a ≤ c
4. **But no requirement that every pair is comparable**

- There may exist pairs for which neither element precedes the other.
- s where any two elements a,b (binary relation ≤) where not every pair is comparable[1].
-
- Partial orders thus generalize total orders, in which every pair is comparable.

**Note:** ≤ is a binary relation on some set X, whose elements a,b and c satisfy the above set of properties

**Concurrency Examples:**

1. **Program order** inside a thread is a partial order: some instructions are ordered (sequenced-before), but instructions in different threads are not directly ordered unless synchronization links them.
2. **Happens-before** is a partial order, it does not compare all actions, only those causally linked.

[^1]:https://en.wikipedia.org/wiki/Total_order

# Key Definitions: Total Order

- Total Order (Linear Order) **is a Partial Order** where any two elements a,b are comparable and satisfy the reflexive, antisymmetric and transitive properties of partial orders[1]
- Partial orders thus generalize total orders, in which every pair is comparable[1].
- This allows for totality (comparability): for every a, b, either a ≤ b or b ≤ a.
- Every event has a clear “before/after” relationship

**Total Order:**
A ----> B ----> C ----> D
(All comparable - Every event has a clear “before/after” relationship.)

**Total Order:**
A ----> B ----> C ----> D
(All comparable)

[^1]:https://en.wikipedia.org/wiki/Total_order

# Key Definitions: Single Total Order

- A special case of Total Order where all relevant operations are covered globally, not just locally.
  - In Rust/C++: SeqCst has a single global total order on all SeqCst operations
  - Threads observe results consistent with this single order
-

**Total Order:**
AW1 ----> B ----> C ----> D
(All comparable)

[^1]:https://en.wikipedia.org/wiki/Total_order

# Key Definitions: Single Total Order

- A special case of Total Order where all relevant operations are covered globally, not just locally.
  - In Rust/C++: SeqCst has a single global total order on all SeqCst operations
  - Threads observe results consistent with this single order
-

**Total Order:**
AW1 ----> B ----> C ----> D
(All comparable)

[^1]:https://en.wikipedia.org/wiki/Total_order

# Key Definitions: Global Order

- “A universally agreed-upon ordering of operations visible to all observers, not just within one thread.”
  - Differs from Total Order as it could include multiple Total Order’s
  - E.g. Modification order for a single atomic is a global order on it’s writes

**Global Order:**
Atomic X:  W1 -> W2 -> W3
Atomic Y:  W4 -> W5

[^1]:https://en.wikipedia.org/wiki/Total_order

# Key Definitions Summary

1. **Partial Order:** A binary relation over a set of elements where not every pair a,b is comparable and satisfy the reflexivity, antisymmetry, transitivity.
2. **Total Order:**
   a. Builds on Partial Order but enables full comparability across objects
   b. Every event has a clear “before/after” relationship
3. **Single Total Order:** A special case for Total Order where all relevant operations are covered globally, not just locally.
   a. In Rust/C++ - SeqCst has a single global total order on all SeqCst operations
   b. Threads observe results consistent with this single order
4. **Global Order:** “A universally agreed-upon ordering of operations visible to all observers, not just within one thread.”
   a. Differs from Total Order as it could include multiple Total Order’s
   b. E.g. Modification order for a single atomic is a global order on it’s writes

**Partial Order (Happens-Before):**

```
Thread1:   A ----> B
Thread2:   C ----> D
(No order between A and C)
```

**Total Order:**

```
A ----> B ----> C ----> D
(All comparable)
```

**Single Total Order:**
SeqCst ops across threads form ONE linear timeline.

**Global Order e.g. MO:**

```
X.atomic:   W1 -> W2 -> W3
Y.atomic:   W4 -> W5
(All threads agree on each atomic’s order, but no cross-atomic order)
```

[^1]-https://www.cs.cmu.edu/afs/cs/academic/class/15740-s18/www/lectures/08-09-consistency.pdf
[^2]- (Lamport, 1979)