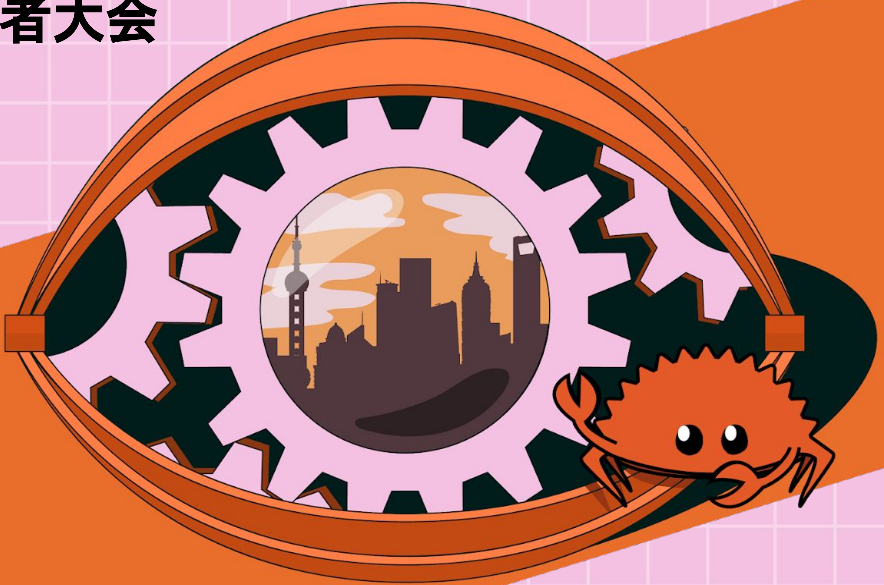
# RUST CHINA CONF 2023

第三届中国Rust开发者大会



6.17-6.18 @Shanghai

Rust原子操作高性能实践

# Rust Atomic Deep Dive

Pu Wang @ DatenLord

2023/06/17

## Outline

## What are Atomic Operations in Rust?

### Compare and Swap

```
fn compare_exchange(
    &self, // AtomicI8
    current: i8,
    new: i8,
    success: Ordering,
    failure: Ordering
) -> Result<i8, i8>
```

### Fetch and Modify

```
fn fetch_add(&self, val: i8, order: Ordering) -> i8
fn fetch_and(&self, val: bool, order: Ordering) -> bool
…
```

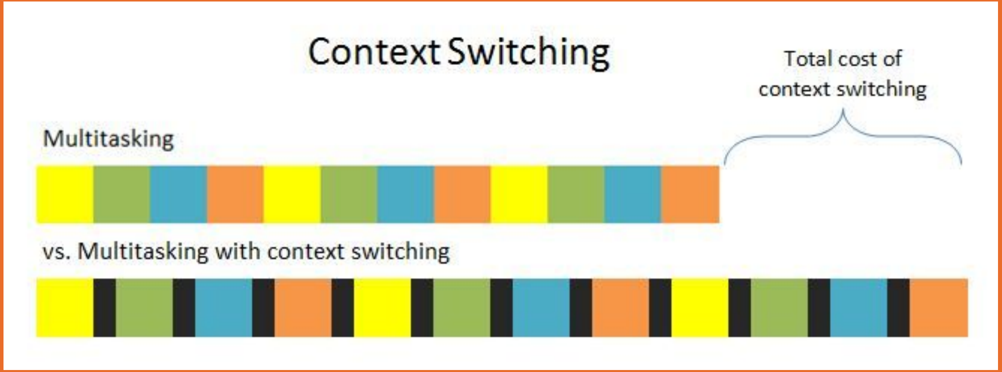
## Why Atomic Operation?

**High Performance**

Lock-free programming

**Shared Variable Access When Multi-threading**

Lock - context switch
Atomic - no context switch


Context Switching
Total cost of context switching
Multitasking
vs. Multitasking with context switching

03

## How to Understand Atomic Operations?

### Memory Model

Memory Order

The order of load/store instructions accessing memory.

### Cache Coherence

Atomic operation overhead

Atomic operations will change cache line status which might flush cache lines.

04

# Memory Model

- Program order
  - Instructions executed as the order defined in a thread
- Memory order
  - Memory access instructions
    - load & store
  - Out of order
    - Instruction reorder by compilers
    - Out of order execution in CPU
      - load v.s. load
      - load v.s. store
      - store v.s. load
      - store v.s. store

## Instruction Reorder by Compilers

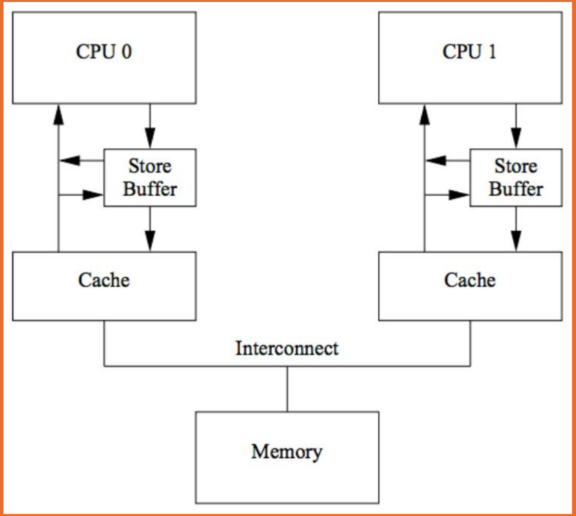
```c
int A, B;

void foo()
{
    A = B + 1;
    B = 0;
}
```

```
$ gcc -S -masm=intel foo.c
$ cat foo.s
        ...
        mov     eax, DWORD PTR _B
        add     eax, 1
        mov     DWORD PTR _A, eax
        mov     DWORD PTR _B, 0
```

```
$ gcc -O2 -S -masm=intel foo.c
$ cat foo.s
        ...
        mov     eax, DWORD PTR B
        mov     DWORD PTR B, 0
        add     eax, 1
        mov     DWORD PTR A, eax
```

04

## Out of Order Execution



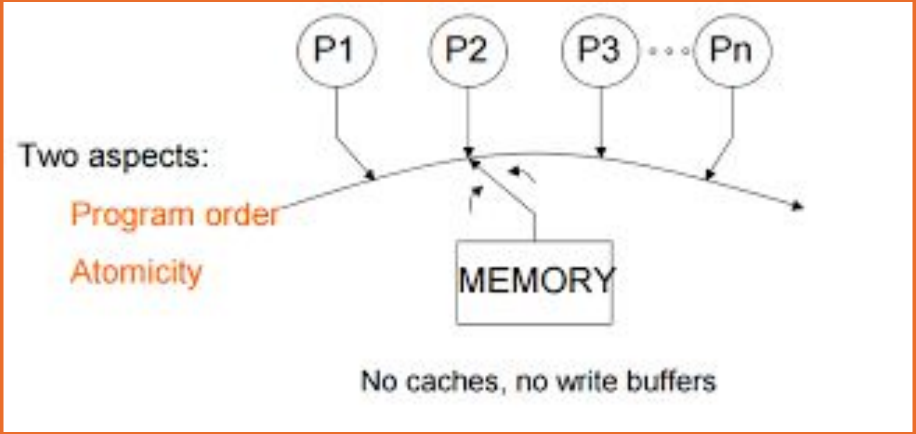
- Write buffer / store buffer
  - Delayed write
  - Store forwarding
- Total store order (TSO)
  - FIFO writer buffer
    - OoO store v.s. load
- Partial store order (PSO)
  - Non-FIFO write buffer
    - OoO store v.s. load
    - OoO store v.s. store

04

## Memory Order in C++/Rust

- Sequential Consistency
- Acquire
- Release
- AcqRel
- Relaxed
- Consume

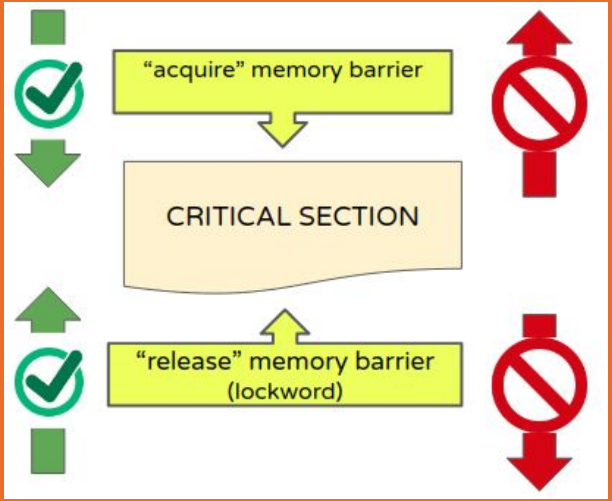
## Sequential Consistency



- Each processor issues memory operations in program order
- The switch provides the global serialization among all memory operations

04

## Acquire Release

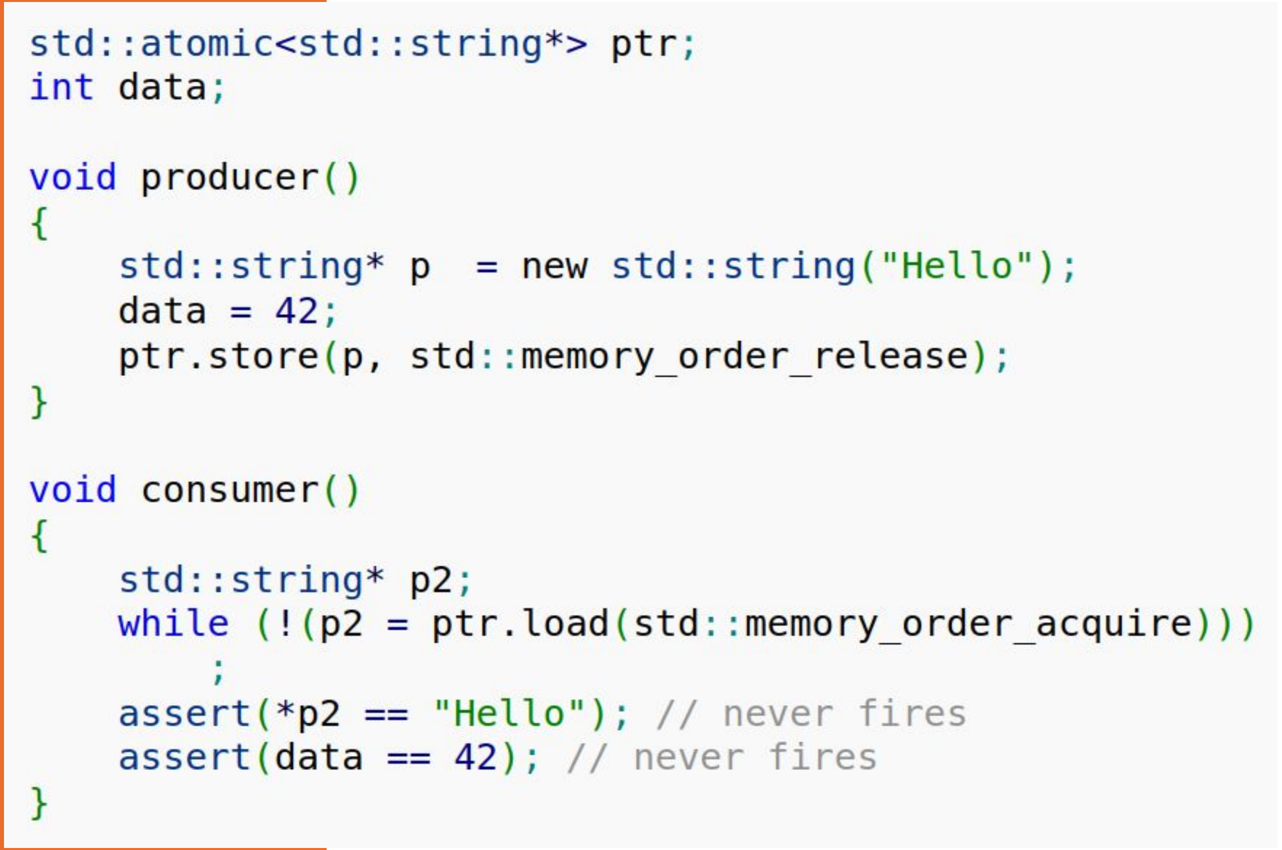


- A write-release guarantees that all preceding code completes before the releasing write

- A read-acquire guarantees that all following code starts after the acquiring read

04

# Acquire Release Example



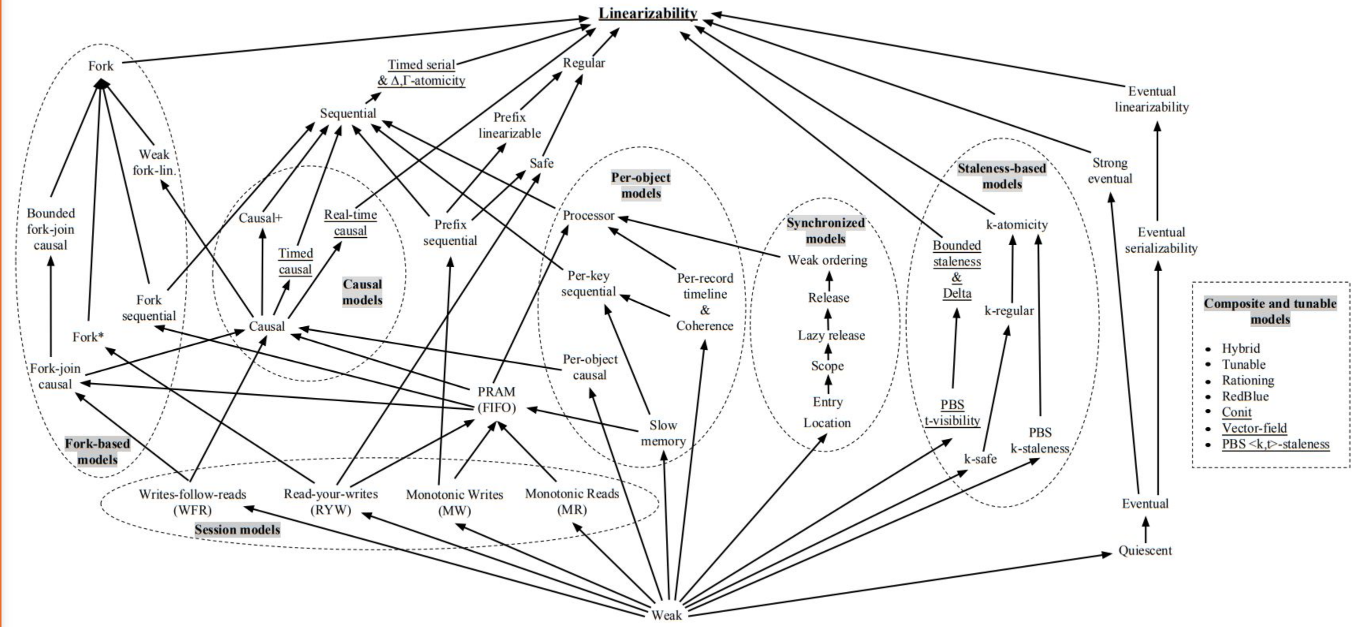
```
std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p  = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire)))
        ;
    assert(*p2 == "Hello"); // never fires
    assert(data == 42); // never fires
}
```

04

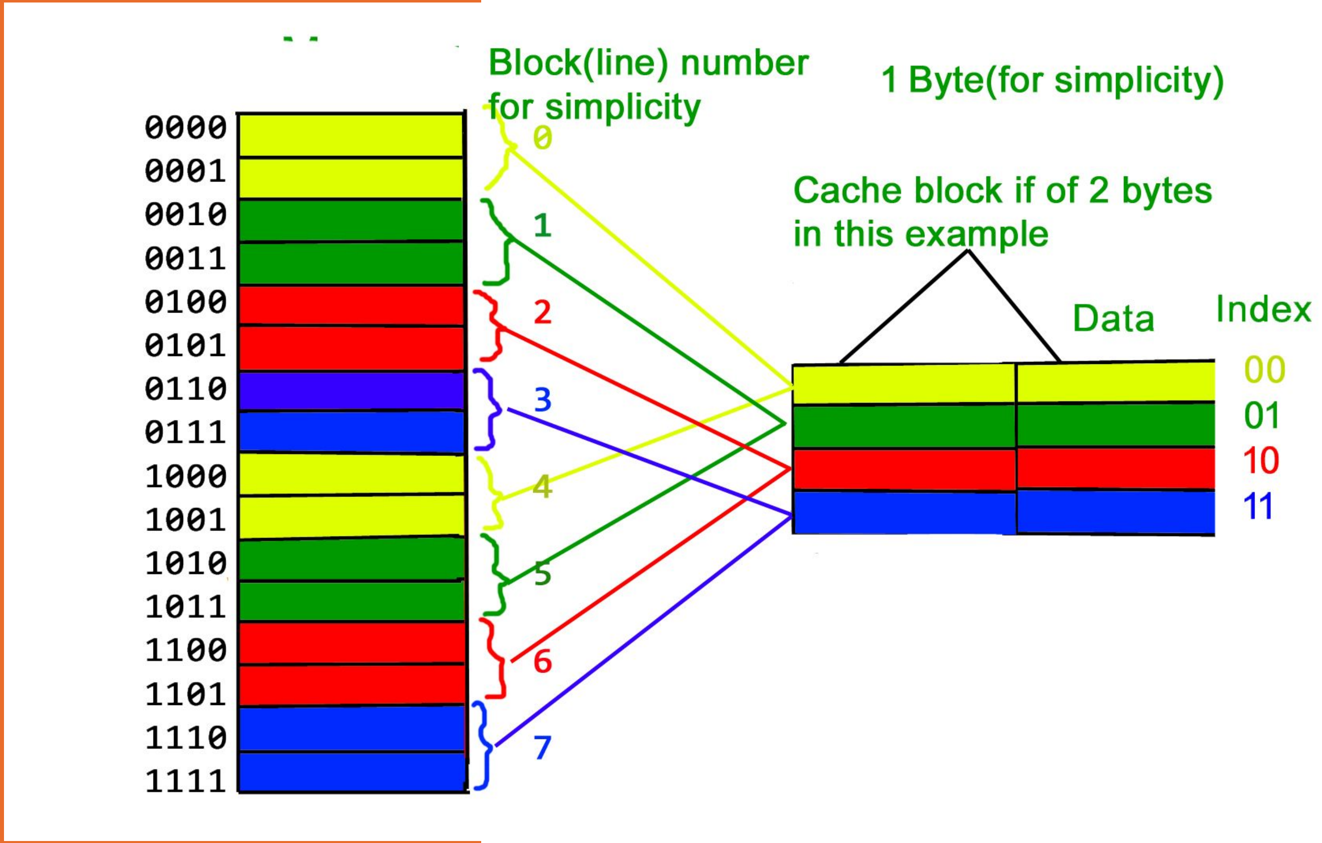
# Consistency

Consistency in Non-Transactional Distributed Storage Systems

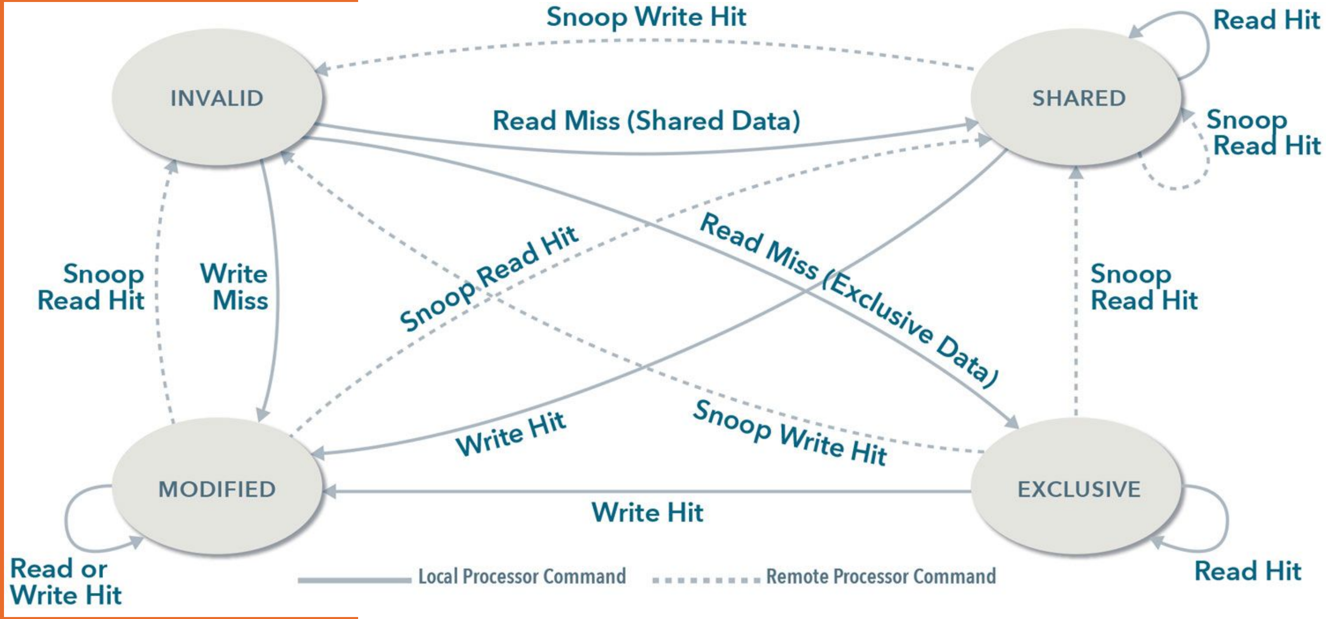
https://arxiv.org/pdf/1512.00168.pdf



## Cache

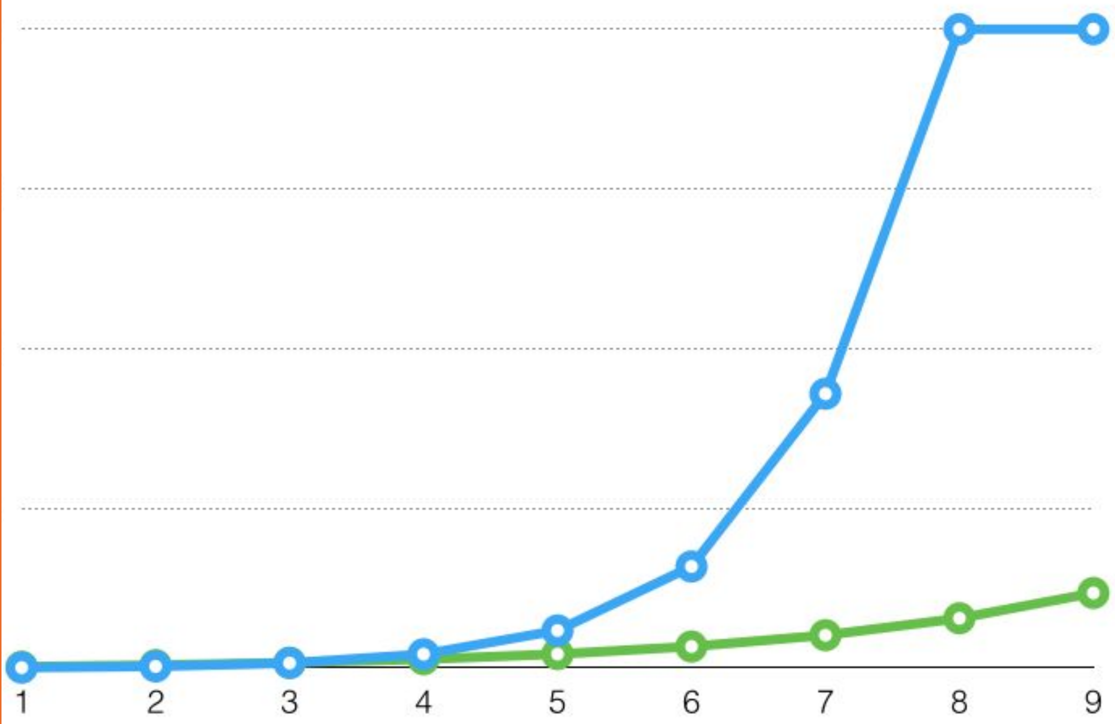

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Block(line) number for simplicity
0
1
2
3
4
5
6
7
1 Byte(for simplicity)
Cache block if of 2 bytes in this example
Data
Index
00
01
10
11

# Cache Coherence



05

## Exponential Backoff Retry



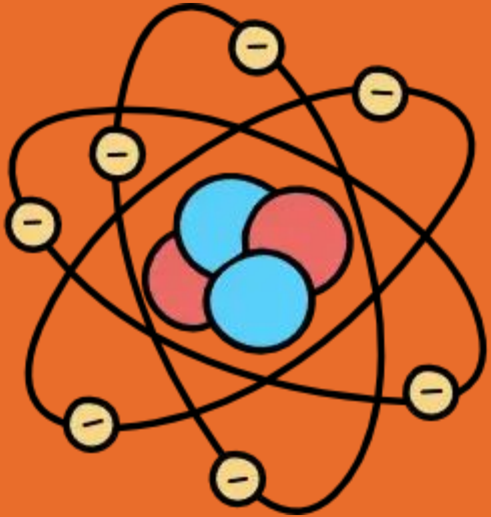
## Summary

### Memory Order

SeqCst
Acquire
Release
AcqRel

### Cache Coherence

Cache line flush overhead
Exponential backoff retry



# Thank you !