Atomics and Memory Order

Joao Barbosa

Week 2, March 2021

From last class

Single Source Shortest Path

Delta-stepping algorithm

- Use several buckets to subdivide distance
- Use a priority queue for each bucket
- Perform a parallel Djisktra (or other) for each bucket
- Put the active edge on the appropriate bucket
- When the bucket becomes empty go to next bucket

Atomics

Atomics basis of ”Lock-free” programming

”Lock-free” means ”fast”

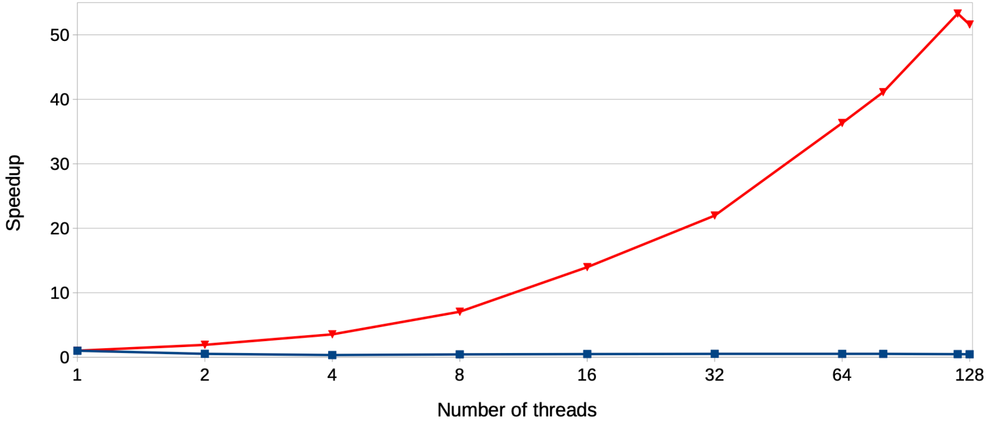
[

Performance: Measure, Measure, Measure]

- Both programs encode the same operation and get the same result
- Both programs are correct and with no ”wait-loops”
- One program uses std::mutex the other one is ”wait-free” (even better than Lock-free)

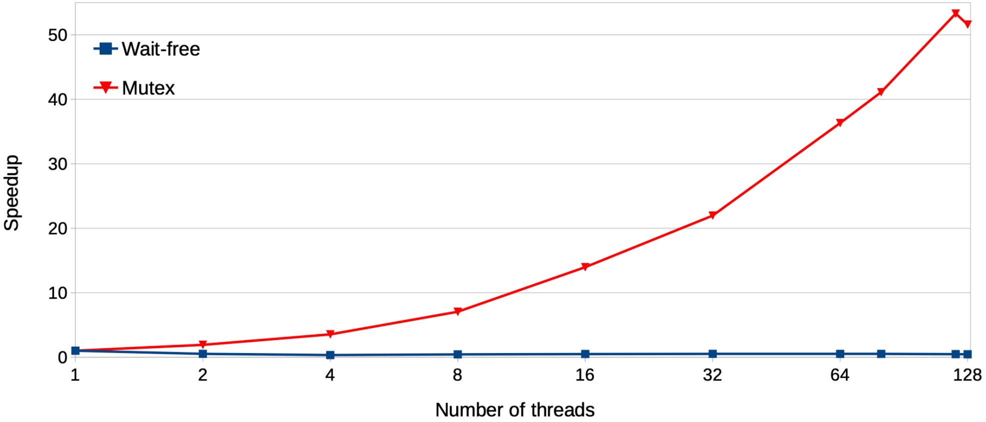
Atomics basis of ”Lock-free” programming

”Lock-free” means ”fast”



Atomics basis of ”Lock-free” programming

”Lock-free” means ”fast”



Atomics basis of ”Lock-free” programming

”Lock-free” means ”fast”

Wait-free

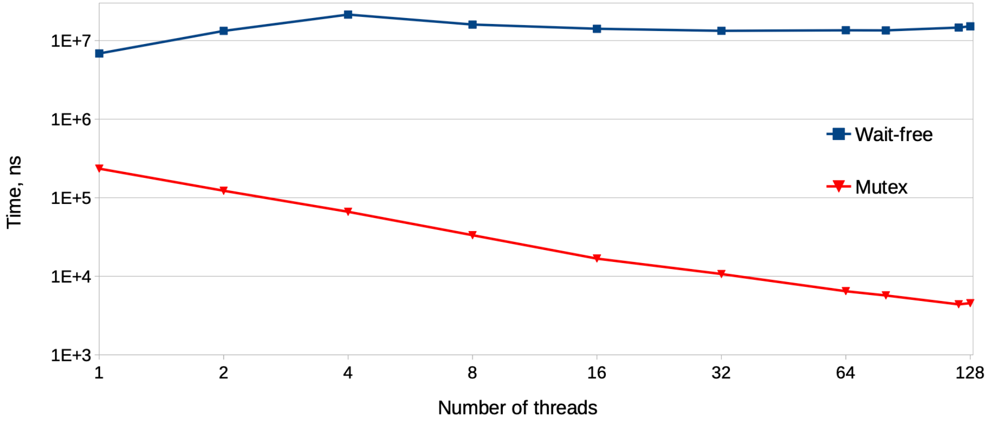
```
std::atomic<int> sum = 0;
(...)
for (int i=0; i<N; i++)
    sum += A[i]
(...)
```

Lock

```
std::mutex M;
(...)
for (int i=0; i<N; i++)
    localsum += A[i]
std::lock_guard L(M);
sum += localsum;
(...)
```

Atomics basis of ”Lock-free” programming

”Lock-free” faster?



Atomics basis of ”Lock-free” programming

”Lock-free” faster?

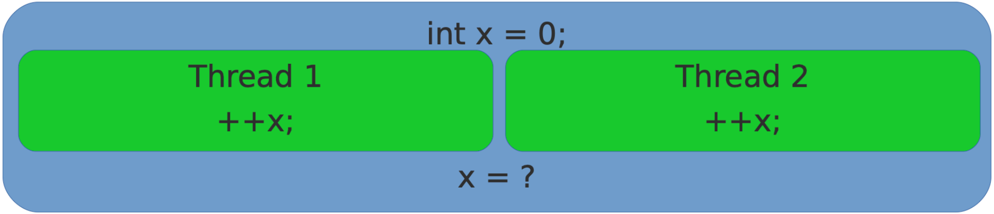
- **Algorithms rule supreme**
- ”Wait-free” has nothing to do with time
  - Refers to the number of compute ”steps”
  - Steps don’t have to be of the same duration
- Atomics **do not** guarantee good performance
- There is no substitute for understanding what you are doing

Atomic operations

What is an atomic?

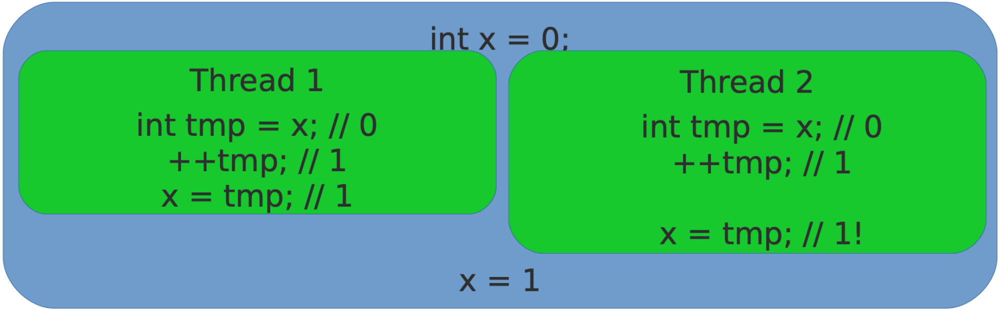
- Atomic opereations is an operation that is guaranteed to execute as a single transation:
  - Other threads will see the state of the system before the operation started or after it finished, but never in the intermediate state
  - At the low level, atomic operations are special hardware instructions

Atomic operation example



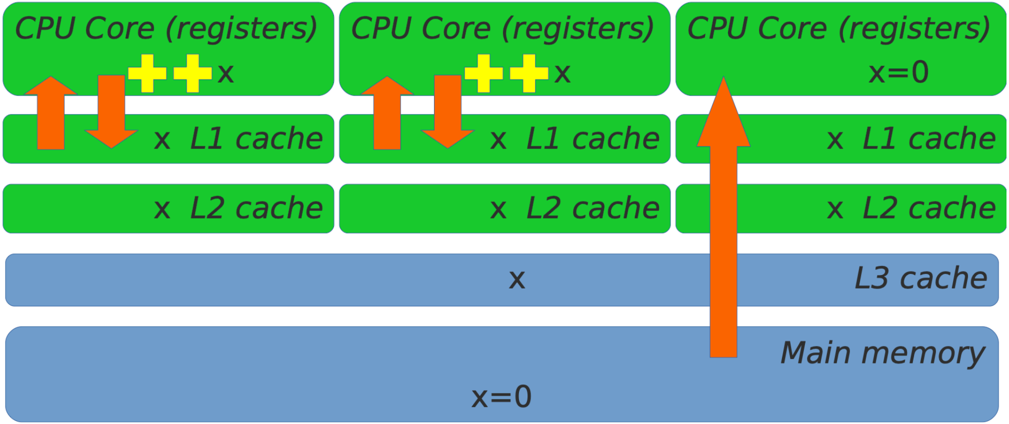
- Increment is a ”read-modify-write” operation
  - read X
  - add 1 to X
  - write new value of X

Atomic operation example



- *read-modify-write* operation is non atomic
- it is a *data race*, i.e., non defined behaviour

Atomic operation example



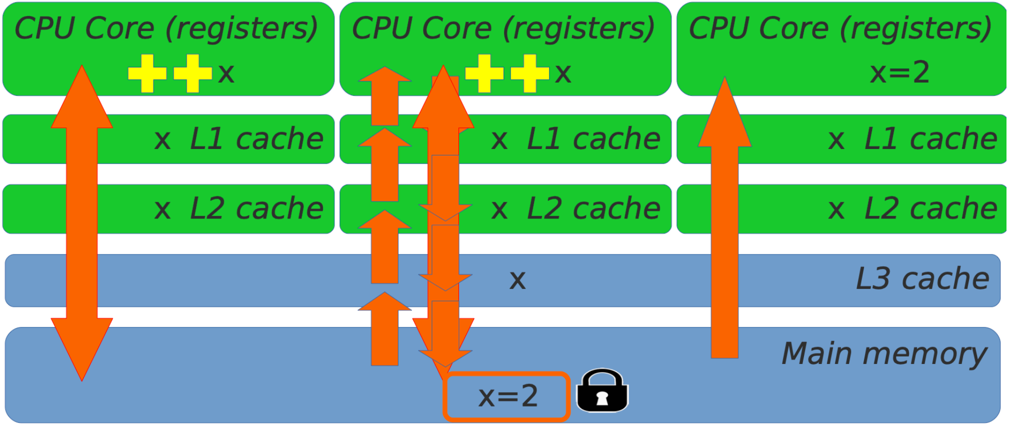
Atomic operations

- std::atomic

```
std::atomic<int> x(0)

(... Inside thread ...)
++x;
```

Atomic operation example



std::atomic

Atomic operation

- What C++ types can be made atomic?
- What operations can be done on those types?
- Are all operations on atomic types atomic?
- How fast are atomic operations?
- Is atomic the same as lock-free?
- If atomic operations avoid locks, there is no wait, right?

Atomic operation

- Any trivially copyable type can be made atomic
- What is trivially copyable?
  - Continuous chunk of memory
  - Copying the object means copying all bits
  - No virtual functions
- Examples
  - std::atomic<int>
  - std::atomic<double>
  - struct S long x; long y;; std::atomic<S>

What operations can be done with std::atomic<T>

- Assignment reads and writes
- Special atomic operations
- Other atomic operations depends on <T>

What operations can be done with std::atomic<T>

- Assignment reads and writes
- Special atomic operations
- Other atomic operations depends on <T>

What operations can be done with std::atomic<T>

```
std::atomic<int> x{0};

++x;
x++;
x += 1;
x |= 2;
x *= 2;
int y = x * 2;
x = y + 1;
x = x + 1;
x = x * 2;
```

What operations can be done with std::atomic<T>

```
std::atomic<int> x{0};

++x;               // Atomic pre-increment
x++;               // Atomic post-increment
x += 1;            // Atomic increment
x |= 2;            // Atomic bit set
x *= 2;            // No atomic multiplication
int y = x * 2;     // Atomic read x
x = y + 1;         // Atomic write of x
x = x + 1;         // Atomic read followed by atomic write
x = x * 2;         // Atomic read followed by atomic write
```

std::atomic<T> and overloaded operators

- std::atomic provides overload operators only for atomics
  - False (it just will not compile)
- any expression with atomics will not be atomic
  - Easy to make mistakes

$++x \equiv x \mathrel{+}= 1 \equiv x = x + 1,$

if x is not atomic

std::atomic<T> operation for type

- Assignment and copy for all types
- Increment and decrement of raw pointers
- Addition, subtraction, and bit logic operations for integers
- T=bool is valid, no special operations
- T=double is valid, no special operations

std::atomic<T> operation for type

- Explicit reads and writes

```
std::atomic<int> x;

auto a = x.load();
(...)
x.store(a);
```

- Atomic exchange

```
auto z = x.exchange(a); // z = x and x = y
```

std::atomic<T> operation for type

- Compare and swap

```
bool success = x.compare_exchange_strong(y, z);
// If x==y, make x=z and return true
// Otherwise, set y=x and return false
```

- Compare-and-swap is the basis for lock-free algorithms

std::atomic<T> operation for type

- Compare and swap increment

```
std::atomic<int> x{0};
int x0 = x;
while ( !x.compare_exchange_strong(x0, x0+1) ) {}
```

- Compare-and-swap multiplication

```
std::atomic<int> x{2};
int x0 = x;
while ( !x.compare_exchange_strong(x0, x0*2) ) {}
```

std::atomic<T> operation for type

- For integers only

```
std::atomic<int> x; x.fetch_add(y);
int z = x.fetch_add(y);
```

- Same for **fetch_sun(), fetch_and(), fetch_or(), fetch_xor()**
  - Less error prone than overload operators

Is std::atomic<T> lock-free?

- std::atomic hides a secret

```
long x;

struct A { long x; }

struct B { long x; long y; };

struct C { long x; long y; long z; };
```

Is std::atomic<T> lock-free?

- std::atomic is not always lock-free
- std::atomic::is_lock_free()

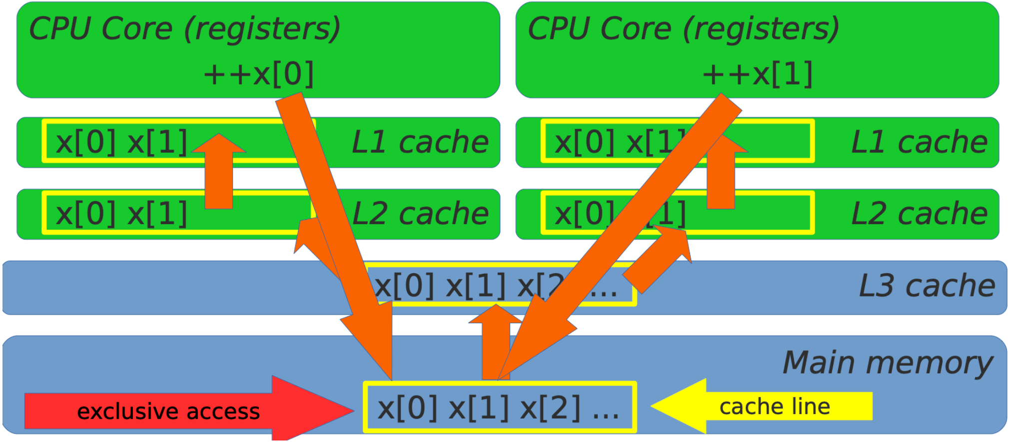
```
long x;                                   // Lock-free
struct A { long x; }                      // Lock-free
struct B { long x; long y; };
struct C { long x; long y; long z; };     // Not Lock-free
```

- Results are runtime and platform dependent
  - Why not compile time? - Alignment
- C++ 1 add a constexpr is$_{a}lways_{l}ock_{f}ree()$

Is std::atomic<T> lock-free? X86 Example

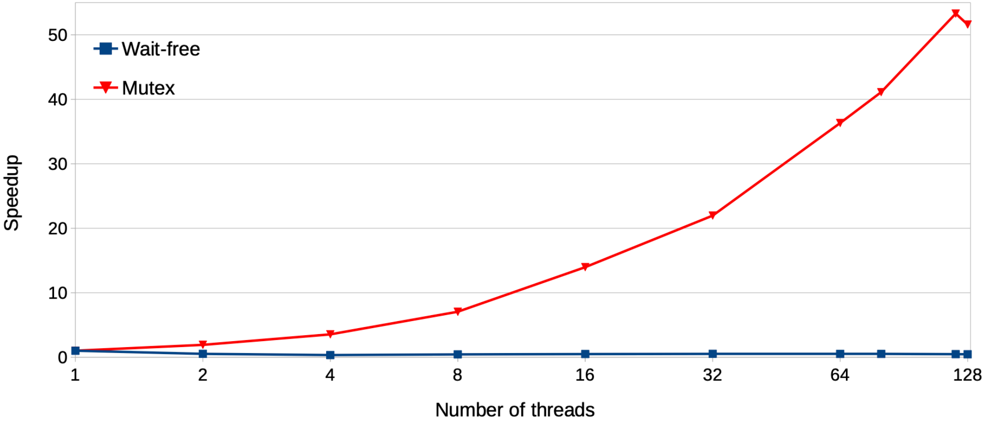
```
long x;                                    // Lock-free - atomic move %mmx
struct A { long x; }                       // Lock-free - atomic move %mmx
struct B { long x; long y; };              // Lock-free - atomic move %mmx
struct C { long x; int y; };               // Not Lock-free 12 bytes
struct E { long x; long y; long z; };        // Not Lock-free >16 bytes
```

Is std::atomic<T> lock-free? Cache alignment



Do atomic operations wait on each other?

”Lock-free” means ”fast”



Do atomic operations wait on each other?

”Lock-free” faster?

- **Algorithms rule supreme**
- ”Wait-free” has nothing to do with time
  - Refers to the number of compute ”steps”
  - Steps don’t have to be of the same duration
- Atomic operations do wait on each other
  - In particular, write operations do
  - Read-only operations can scale near-perfectly

Do atomic operations wait on each other?

”Lock-free” faster?

- Atomic operations have to wait for cache line access
  - Price of data sharing without races
  - Accessing different locations in the same cache line still incurs run-time penalty (false sharing)
  - Avoid false sharing by aligning per-thread data to separate cache lines
  - On NUMA machines, may be even separate pages

Strong and weak compare-and-swap

- C++ provides two versions of CAS – weak and strong

```
x.compare_exchange_strong(old_x, new_x) // if (x == old_x)
                                        //      { x = new_x; return true; }
                                        // else { old_x = x; return false; }
```

- x.compare_exchange_weak(old_x, new_x): same thing but can "spuriously fail" and return false even if x==old_x
- What is the value of old_x if this happens?

Strong and weak compare-and-swap

- C++ provides two versions of CAS – weak and strong

```
x.compare_exchange_strong(old_x, new_x) // if (x == old_x)
                                        //       { x = new_x; return true; }
                                        // else { old_x = x; return false; }
```

- x.compare_exchange_weak(old_x, new_x): same thing but can "spuriously fail" and return false even if x==old_x
- What is the value of old_x if this happens? Must be old_x!
- If weak CAS correctly returns x == old_x, why would it fail?

Strong and weak compare-and-swap

```
x.compare_exchange_strong(old_x, new_x) // if (x == old_x)
                                        //      { x = new_x; return true; }
                                        // else { old_x = x; return false; }
```

- x.compare_exchange_weak(old_x, new_x): same thing but can “spuriously fail” and return false even if x==old_x
- What is the value of old_x if this happens? Must be old_x!
- If weak CAS correctly returns x == old_x, why would it fail?

Strong and weak compare-and-swap

```
bool compare_exchange_strong(T& old_v, T new_v) {
    Lock L;           // Get exclusive access
    T tmp = value;    // Current value of the atomic
    if (tmp != old_v) { old_v = tmp; return false; }
    value = new_v;
    return true;
}
```

- Lock is not a real mutex but some form of exclusive access implemented in hardware

Strong and weak compare-and-swap

```
bool compare_exchange_strong(T& old_v, T new_v) {
    T tmp = value; // Current value of the atomic
    if (tmp != old_v) { old_v = tmp; return false; }
    Lock L; // Get exclusive access
    tmp = value; // value could have changed!
    if (tmp != olv_v) { old_v = tmp; return false; }
    value = new_v;
    return true;
}
```

- Double-checked locking pattern is back!

Strong and weak compare-and-swap

```
bool compare_exchange_weak(T& old_v, T new_v) {
    T tmp = value; // Current value of the atomic
    if (tmp != old_v) { old_v = tmp; return false; }
    TimedLock L; // Get exclusive access or fail
    if (!L.locked()) return false; // old_v is correct
    tmp = value; // value could have changed!
    if (tmp != olv_v) { old_v = tmp; return false; }
    value = new_v;
    return true;
}
```

- Double-checked locking pattern is back!

Atomics memory order

```
int q[N];
std::atomic<size_t> front;
void push(int x) {
    size_t my_slot = front.fetch_add(1);
    q[my_slot] = x;
}
```

- Atomic variable is an index to (non-atomic) memory

Memory order

```
struct node { int value; node* next; };

std::atomic<node*> head;
void push_front(int x) {
    node* new_n = new node;→
    new_nvalue = x;
    node* old_h = head;
    do { →new_nnext = old_h; }
    while (!head.compare_exchange_strong(old_h,new_n);
}
```

- Atomic variable is a pointer to (non-atomic) memory

Memory order

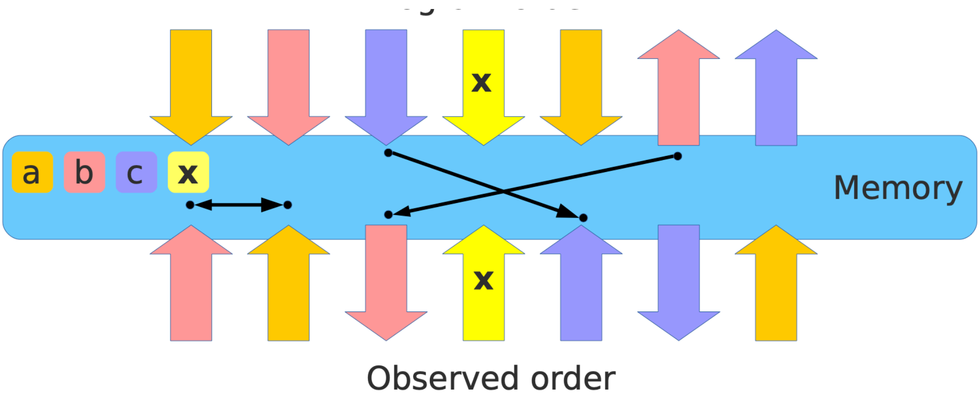
- Atomics are used to get exclusive access to memory or to reveal memory to other threads
- But most memory is not atomic!
- What guarantees that other threads see this memory in the desired state
  - For **acquiring** exclusive access: data may be prepared by other threads, must be completed
  - For **releasing** into shared access: data is prepared by the owner thread, must become visible to everyone

Memory order

- C++03 as no portable memory barriers C++11 provides standard memory barriers
- Memory barriers are closely related to “memory order” – they are what ensures the memory order
- C++ memory barriers are modifiers on atomic operations
- Actual implementation may vary

```
std::atomic<int> x;
x.store(1, std::memory_order_release);
```

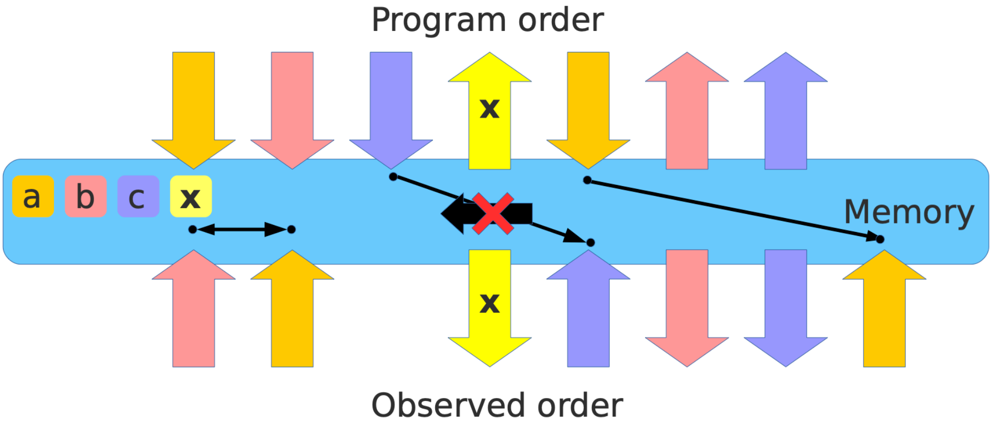
Memory order : std::memory_order_relaxed



Memory order : std::memory_order_acquire

- Acquire barrier guarantees that all memory operations scheduled after the barrier in the program order become visible after the barrier
  - “All operations” not “all reads” or “all writes”, i.e. both reads and writes
  - “All operations” not just operations on the same variable that the barrier was on
- Reads and writes cannot be reordered from after to before the barrier
  - Only for the thread that issued the barrier!

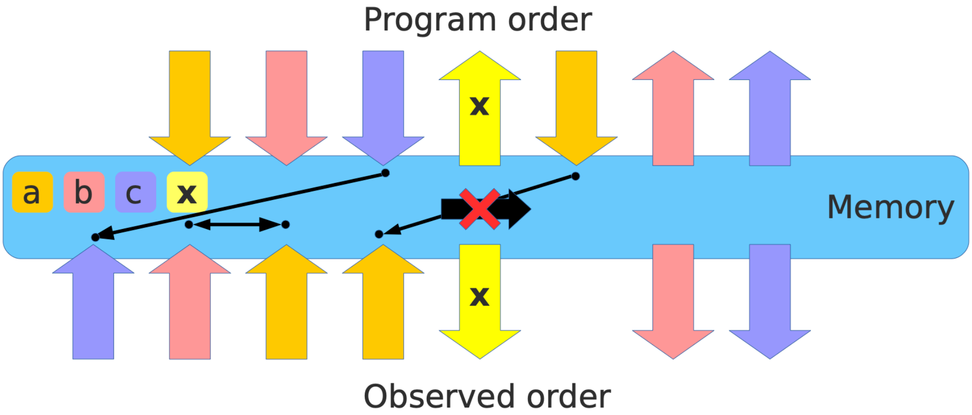
Memory order : std::memory_order_acquire



Memory order : std::memory_order_release

- Release barrier guarantees that all memory operations scheduled before the barrier in the program order become visible before the barrier
- Reads and writes cannot be reordered from before to after the barrier
  - Only for the thread that issued the barrier!

Memory order : std::memory_order_release



Memory order : Acquire / Release protocol

- Acquire and release barriers are often used together:
- Thread 1 writes atomic variable **x** with release barrier
- Thread 2 reads atomic variable **x** with acquire barrier
- All memory writes that happen in thread 1 before the barrier (in program order) become visible in thread 2 after the barrier
- Thread 1 prepares data (does some writes) then **releases** (publishes) it by updating atomic variable x
- Thread 2 **acquires** atomic variable x and the data is guaranteed to be visible

Memory order : Acquire / Release memory barrier and SEQ consisten

- Acquire-Release (std::memory_order_acq_rel) combines acquire and release barriers – no operation can move across the barrier
  - But only if both threads use the same atomic variable!
- Sequential consistency (std::memory_order_seq_cst) removes that requirement and establishes single total modification order of atomic variables