Designing an SPSC Lock-free queue

Back to the basics

Quasar Chunawala

quasar-chunawala
http://quantdev.blog
quasar-chunawala

A brief refresher

- Processes vs Threads:
  - A *process* is a running instance of a piece of code with its own private set of resources - including memory, file descriptors and execution context. It is an OS-managed entity.
  - A *thread* is a light-weight, efficient construct to execute multiple tasks within a single process. Threads share the same memory address space as the parent, including file descriptors, heap memory, and any other global data structures allocated by the process. OS-managed.
- Since all threads within a process share the same memory space, they can directly access common variables without the need for complex IPC mechanisms.
- Sharing the same memory space introduces the challenge of managing access to shared resources.
- To prevent data corruption and ensure the integrity of shared data, threads must employ synchronization mechanisms :
  - Locks, Semphores, Mutexes, condition variables.
- Preemptive multitasking involves the use of an hardware timer interrupt mechanism which suspends the currently executing process and invokes the OS scheduler to determine which process should execute next.
- Coroutines are a relatively new feature in C++. Coroutines can be defined as functions that can be paused and resumed at specific points, allowing for cooperative multitasking.
- Coroutines are cooperative, which means they must explicitly yield control to the caller in order to switch execution context.

Creating and managing threads

```cpp
#include <print>
#include <thread>

// t1 using function pointer
void func(){
    std::println("using function pointer");
}

// t2 using lambda function
auto lambda_func = [](){
    std::println("using lambda function");
};

// t4 using function object
class FunctionObjectClass{
    public:
    void operator()(){
        std::println("using function object class");
    }
};

// t5 using non-static member function
class Foo{
    public:
    void bar(){
        std::println("Using a non-static member
        ↪ function");
    }
};

int main(){
    std::thread t1(func);
    std::thread t2(lambda_func);

    // t3 using an embdedded lambda
    std::thread t3([](){
        std::println("using embedded lambda
        ↪ function");
    });

    std::thread t4{ FunctionObjectClass{} };

    Foo foo;
    std::thread t5{&Foo::bar, &foo};

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();
    return 0;
}
```

join() and detach()

- Once a `std::thread` is created, it must be either joined or detached.
- The `join()` function blocks the current thread while waiting for the completion of the specified thread identified by the `thread` object whose `join` function is invoked.
- A thread is considered *joinable* and active if the `join()` function has not been called on that thread. This is true, even if the thread has finished executing, but still has not been joined on.
- A default constructed thread or a thread that has already been joined is not *joinable*.
- To check whether a thread is joinable, we can use `std::thread::joinable()` function.

```cpp
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void func() {
    std::this_thread::sleep_for(100ms);
}

int main() {
    std::thread t1;
    std::cout << "Is t1 joinable? " << t1.joinable()
    ↪  << std::endl;

    std::thread t2(func);
    t1.swap(t2);
    std::cout << "Is t1 joinable? " << t1.joinable()
    ↪  << std::endl;
    std::cout << "Is t2 joinable? " << t2.joinable()
    ↪  << std::endl;

    t1.join();
    std::cout << "Is t1 joinable? " << t1.joinable()
    ↪  << std::endl;

    return 0;
}
```

join() and detach()

- If we want a thread to continue running in the background as a *daemon* thread, but finish the execution of the current thread, we can use the `std::thread::detach()` function.
- A daemon thread is a thread that performs some tasks in the background, that do not need to run to completion e.g. garbage collection.
- After calling `detach()` the detached thread cannot be controlled or joined using the `std::thread` object, since this object no longer represents the detached thread.

```cpp
#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

namespace {
int timeout = 3;
}

void daemonThread() {
    sync_cout << "Daemon thread starting...\n";
    while (timeout-- > 0) {
        sync_cout << "Daemon thread is running...\n";
        std::this_thread::sleep_for(1s);
    }
    sync_cout << "Daemon thread exiting...\n";
}

int main() {
    std::thread t(daemonThread);
    t.detach();

    std::this_thread::sleep_for(std::chrono::seconds(timeout +
    ↪  1));

    sync_cout << "Main thread exiting...\n";
    return 0;
}
```

`std::jthread` class

- `std::jthread x{Callable,Args...}` is a wrapper over `std::thread` that creates a new thread running `Callable(Args...)`.
- `std::jthread` destructor calls `join()` on the underlying thread and blocks until the owned thread completes. It implements the RAII idiom. They avoid pitfalls when we forget to use `join` on a thread.

example_04.cpp

```cpp
#include <chrono>
#include <iostream>
#include <thread>
#include <syncstream>

using namespace std::chrono_literals;
#define sync_cout std::osyncstream(std::cout)

void func(){
    std::this_thread::sleep_for(1s);
}

int main()
{
    std::jthread t(func);
    sync_cout << "Thread Id : " << t.get_id() << "\n";
}
```

Yielding thread execution

- A thread can also decide to pause fits execution, let the implementation reschedule the execution of threads.
- The `std::this_thread::yield` method provides a hint to the OS to reschedule another thread.
- The OS will suspend the current thread and move it back to a queue of threads to schedule all the threads with the same priority.
- The example on the right shows 2 threads `t1` and `t2`, executing the same work package. They randomly choose to either do some work (e.g. locking a mutex) or yield the execution to another thread.

example_05.cpp

```cpp
#include <chrono>
#include <iostream>
#include <thread>
#include <syncstream>
#include <mutex>
#include <random>

using namespace std::chrono_literals;
using namespace std::chrono;
#define sync_cout std::osyncstream(std::cout)

namespace{
    int val = 0;
    std::mutex mtx;
}

int main()
{
    auto work = [&](const std::string& name){
        while(true){
            bool work_to_do = rand() % 2;
            if(work_to_do)
            {
                sync_cout << name << ": working" << "\n";
                std::lock_guard<std::mutex> lock(mtx);
                for(auto start = steady_clock::now(),
                         now = start;
                         now < start + 3s;
                         now = steady_clock::now())
                         {}

            }else{
                sync_cout << name << ": yielding" << "\n";
                std::this_thread::yield();
            }
        }
    };

    std::jthread t1(work, "t1");
    std::jthread t2(work, "t2");
    return 0;
}
```

Data Race Condition

- Most multi-threaded programs need to share state between threads.
- Unsynchronized access to a memory location from more than one thread, where at least one thread is writing.
- We see two main issues here: first, the value of `counter` is incorrect; second, every execution of the program ends with a different value of the `counter`. The results are non-deterministic and incorrect.
- `t1` and `t2` run concurrently and modify the same variable. Looking closer, let's study the following line carefully:
- Increment is a read-modify-write operation:
  - The contents of the memory address where the `counter` variable is stored are loaded into a CPU register.
  - The value in the register is incremented by 1.
  - The value in the register is stored in the `counter` variable memory address.
- Consider the below possible scenario:

Thread 1	Thread 2
[1] Load `counter` value into CPU register	[3] Load `counter` value into CPU register
[2] Increment register value	[5] Increment register value
[4] Store register in `counter`	[6] Store register in `counter`

- The `counter` variable has been incremented just once, when it should have been incremented twice. This is a **race condition**.

example_06.cpp

```cpp
#include <iostream>
#include <thread>

int counter=0;

int main(){
    auto func = [](){
        for(int i{0};i < 1000000; ++i){
            counter++;
        }
    };

    std::thread t1(func);
    std::thread t2(func);

    t1.join();
    t2.join();

    std::cout << counter << "\n";
    return 0;
}
```

Output:

example_06

```
1659295
1217311
1167474
```

Mutual Exclusion and `std::mutex`

- **Mutual Exclusion** is the fundamental concept in concurrent programming that ensures that multiple threads or processes do not simultaneously access a shared resource such as a shared variable, a critical section of code, a network socket or a file.
- `std::mutex` offers exclusive ownership semantics.
  - A calling thread must not own the mutex before calling `lock()` or `try_lock()`
  - When a thread owns a mutex, all other threads will block (when calling `lock`. This is the exclusive ownership semantics property of `std::mutex`.
- The `std::mutex` class has three methods:
  - `lock()` : Calling `lock()` acquires the mutex. If the mutex is already locked, then the calling thread is blocked until the mutex is unlocked.
  - `try_lock()` : When called, this function returns either `true`, indicating the mutex has been successfully locked, or `false` in the event of the mutex already being locked.
  - `unlock()` : Calling `unlock()` releases the mutex.

example_07.cpp

```cpp
#include <iostream>
#include <mutex>
#include <thread>
#include <print>

std::mutex mtx;
int counter{0};

int main()
{
    auto func_without_locks = []{
        for(int i{0}; i <1'000'000; ++i){
            ++counter;
        }
    };

    auto func_with_locks = []{
        for(int i{0}; i <1'000'000; ++i){
            mtx.lock();
            ++counter;
            mtx.unlock();
        }
    };
    {
        counter = 0;
        std::thread t1(func_without_locks);
        std::thread t2(func_without_locks);
        t1.join();
        t2.join();

        std::println("Counter without using locks : {}", counter);
    }
    {
        counter = 0;
        std::thread t1(func_with_locks);
        std::thread t2(func_with_locks);
        t1.join();
        t2.join();

        std::println("Counter using locks : {}", counter);
    }
    return 0;
}
```

std::shared_mutex

- There are cases, when we may need to let several threads simultaneously read protected data and give just one thread exclusive access.
- The main difference between `std::shared_mutex` and other mutex types is that it has two access levels:
  - Shared. Several threads can share the ownership of the same mutex. Shared ownership is acquired/released by calling `lock_shared()`, `try_lock_shared()`/`unlock_share`
    While atleast one thread has acquired shared access to the lock, no other thread can get exclusive access to it.
  - Exclusive. Only one thread can own the mutex. Exclusive ownership is acquired/released by calling `lock()`, `try_lock()`/`unlock()`.

example_08.cpp

```cpp
#include <algorithm>
#include <chrono>
#include <iostream>
#include <shared_mutex>
#include <thread>
#include <print>
#include <vector>

int counter{0};

int main()
{
    using namespace std::chrono_literals;
    std::shared_mutex mutex;
    auto reader = [&]{
        for(int i{0}; i<10; ++ i)
        {
            mutex.lock_shared();
            // Read the counter and do something
            mutex.unlock_shared();
        }
    };

    auto writer = [&]{
        for(int i{0}; i<10; ++ i)
        {
            mutex.lock();
            // Update the counter
            ++counter;
            std::println("Thread id = {}, Counter = {}",
std::this_thread::get_id(), counter);
            mutex.unlock();
            std::this_thread::sleep_for(1ms);
        }
    };

    std::vector<std::thread> threads;
    threads.emplace_back(reader);
    threads.emplace_back(reader);
    threads.emplace_back(writer);
    threads.emplace_back(reader);
    threads.emplace_back(reader);
    threads.emplace_back(writer);
    for(auto& t : threads)
        t.join();
}
```

Problems when using locks

- Suppose a thread needs to access 2 resources to complete a task. Each resource is synchronized with a different `std::mutex` object.
- A thread must therefore acquire the first resource mutex, then acquire the second resource mutex, process the resources and finally release both mutexes.
- Consider the following scenario: *thread 1* holds the first mutex and *thread 2* holds the second mutex. *Thread 1* will be blocked forever waiting for the second mutex to be available and *thread 2* will be blocked forever waiting for the first mutex to be available.
- This is called **deadlock** because both threads will blocked forever waiting for each other to release the required mutex.
- One possible solution for deadlock could be the following : when a thread tries to acquire the lock, it blocks just for a limited time, if unsuccessful, it will release any lock it may have acquired.
- For example, *thread 1* acquires the first lock and *thread 2* acquires the second lock. Then, *thread 1* proceeds to acquire the second lock. If after a certain time, *thread 1* still has not acquired the second lock, it releases the first one.
- This solution may work sometimes, but it is not right. Imagine the scenario: *thread 1* has acquired the first lock and *thread 2* has acquired the second lock. After sometime, both threads release their already acquired locks, and then they acquire the same locks again. Then, the threads release their locks and re-acquire them and so forth.
- The threads are unable to do any meaningful work. This is called a **livelock**, because the threads are not blocking forever waiting on a lock, but they are unable to proceed with the work package.
- The most common solution for both deadlocks and livelocks is acquiring the locks in a consistent order. For example, if a thread needs to acquire 2 locks, it will always acquire the first lock first, and then it will acquire the second lock.

Generic lock management

- The C++ standard library provides different wrapper classes for managing mutexes.

Mutex Manager Class	Supported Mutex Types	Mutexes managed
`std::lock_guard`	All	1
`std::scoped_lock`	All	Zero or more
`std::unique_lock`	All	1
`std::shared_lock`	`std::shared_mutex`	1

- The `std::lock_guard` class is an RAII(Resource Acquisition is Initialization) wrapper that makes it easier to use mutexes and guarantees that a mutex will be released when the `lock_guard` destructor will be called.
- This makes exception handling easier, when a lock is already acquired.

example_09.cpp

```cpp
#include <format>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mtx;
uint32_t counter{};

void function_throws(){
    throw std::runtime_error("Error");
}

int main(){
    auto worker = []{
        for(int i{0}; i<1'000'000; ++i){
            mtx.lock();
            counter++;
            mtx.unlock();
        }
    };

    auto worker_exceptions = []{
        for(int i{0}; i<1'000'000; ++i){
            try{
                std::lock_guard<std::mutex> lock(mtx);
                counter++;
                function_throws();
            }catch(std::system_error& e){
                std::cout << e.what() << std::endl;
                return;
            }catch(...){
                return;
            }
        }
    };

    std::thread t1(worker_exceptions);
    std::thread t2(worker);

    t1.join();
    t2.join();

    std::cout << "Final counter value : " << counter << "\n";
    return 0;
}
```

Condition variables

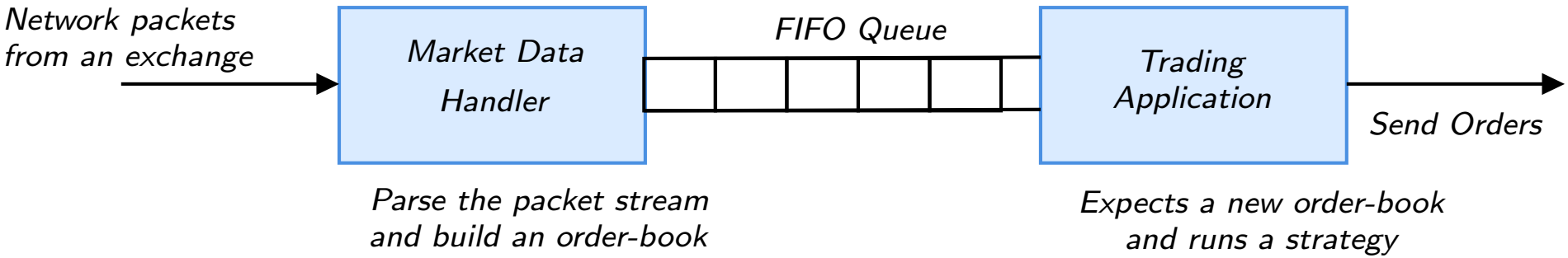
- **Condition variables** are a synchronization primitive provided by the C++ standard library. They allow for several threads to wait for a notification from the other thread.
- Condition variables are always associated with a mutex.
- There are 2 ways to wait for a certain condition: one is waiting in a loop and using a mutex as a synchronization mechanism. This is implemented in the `wait_for_counter_non_zero_mtx`. The function acquires the lock, reads the value in `counter`, and releases the lock.
- Condition variables help us to simplify the previous code. The `wait_for_counter_10_cv` function waits until `counter` is equal to 10. Upon calling `wait()`, the condition variable locks the mutex and waits until the condition is `true`. If the condition is not `true`, the condition variable remains in a *waiting state* until it is signaled and releases the mutex.

example_10.cpp

```cpp
#include <chrono>
#include <condition_variable>
#include <print>
#include <mutex>
#include <thread>
#include <vector>

int counter{0};

int main()
{
    using namespace std::chrono_literals;

    std::mutex mtx;
    std::mutex cout_mtx;
    std::condition_variable cv;

    auto increment_counter = [&]{
        for(int i{0}; i < 20; ++i){
            std::this_thread::sleep_for(100ms);
            mtx.lock();
            ++counter;
            mtx.unlock();
            cv.notify_one();
        }
    };

    auto wait_for_counter_non_zero_mtx = [&]{
        mtx.lock();
        while(counter == 0){
            mtx.unlock();
            std::this_thread::sleep_for(10ms);
            mtx.lock();
        }

        mtx.unlock();
        std::lock_guard<std::mutex> cout_lck(cout_mtx);
        std::println("Counter is non-zero");
    };

    auto wait_for_counter_10_cv = [&]{
        std::unique_lock<std::mutex> lck(mtx);
        cv.wait(lck, []{ return counter == 10; });

        std::lock_guard<std::mutex> cout_lck(cout_mtx);
        std::println("Counter is: {}", counter);
    };

    std::thread t1(wait_for_counter_non_zero_mtx);
    std::thread t2(wait_for_counter_10_cv);
    std::thread t3(increment_counter);

    t1.join();
```

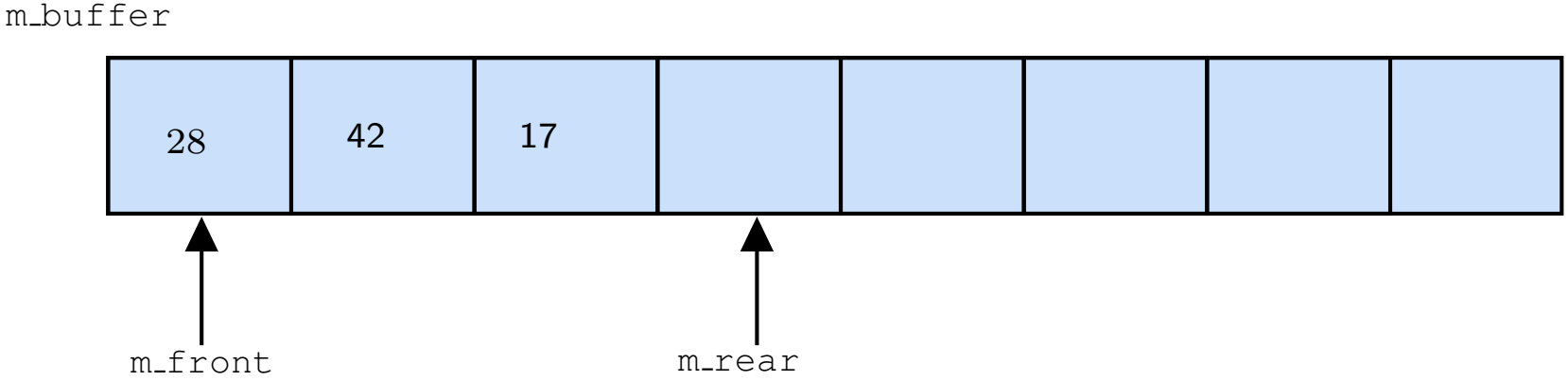
Implementing a basic thread safe queue

- FIFO queues are a standard way of communication between threads. One of the use-cases in the trading industry is as follows :



std::deque is not thread-safe

- std::deque is not thread-safe!
- Suppose the queue has 2 elements. The current value of the index m_front = 0.



- Consider 2 threads $t1$ and $t2$ that simultaneously invoke pop_front().
- Consider the below pseudo-code for pop_front():

```
T pop_front()
{
    T item = m_buffer[m_front];              // [1] Read the element at the front index
    m_front = (m_front + 1) % m_capacity;    // [2] Increment front pointer
}
```

- Imagine the following scenario. First, thread $t1$ executes [1], that is reads the element at the front index(28) from the internal deque storage into the variable item. Next, it computes the result of adding 1 to the current value in the variable m_front, and stores this in a temporary. This temporary holds the value 1. At this point, the OS preempts $t1$ and schedules $t2$. Thread $t2$ executes [1]. $t1$ is still pending a write of the temporary value 1 to m_front. So, $t2$ sees the value of m_front as 0. $t2$ also reads the element at the front of the queue as 42 and stores the result in the variable item. $t2$ then executes [2] and finishes off. The OS resumes $t1$, which also completes running [2] and finishes off. Thus, both pop threads end up popping the same item twice. We are left with 1 item in the queue.

`std::deque` is not thread-safe

- The `std::deque` interface also presents a problem. Consider the following client-code:

Sample code snip

```
// Possible library implementation
namespace std{
    class deque{
        public:
        /* ... */
        [[nodiscard]] size_t front() const{
            return m_front;
        }

        [[nodiscard]] bool is_empty() const{
            return m_front == m_rear;
        }


    };
}
// Client code
void process_next_queue_item(std::deque& queue)
{
    if(!queue.is_empty())              // [1] Check if the queue is empty
    {
        T item = queue.front();     // [2] Read the element at the front index
        queue.pop_front();          // [3] Pop the queue at front-end
        /* Process item ... */
    }
}
```

- With 1 element left in the queue, two worker threads processing items in the queue, can cause a race condition, if a thread is in the middle of popping an element off the queue, while a second thread performs the `!queue.is_empty()` check.

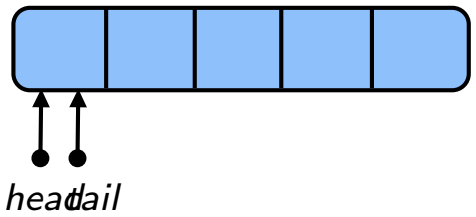
`std::deque` is not thread-safe

- This queue suffers from yet another sort of race condition called the *producer-consumer* problem.
- Suppose the `push` thread waits for packets on a network and puts them on a queue. There can be periods of silence or bursts of data over a network. If the `push` thread produces items at a rate consistently smaller than the rate at which the `pop` thread consumes items, we will often end up with an empty queue and the CPU will be hogged by the `pop_front()` consumer thread and will waste precious CPU cycles waiting for data.
- We are going to code up a bounded queue, also known as a *ring buffer* - that is, a queue with a fixed capacity.

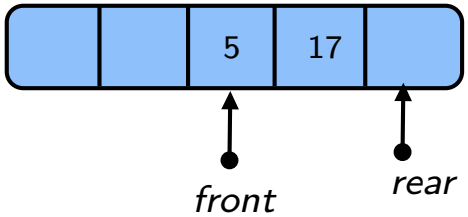
Setup

- Fixed-capacity bounded queue.
- Once we store an element at the end of the internal buffer, the next one will be stored at the beginning, if there is free space, so we *wrap around* `head` and `tail` pointers.
- There are different ways to indicate when a queue is empty and when it is full. In this implementation, I use :
  - If `tail == head`, then the queue is empty.
  - If `(tail + 1) % capacity == head`, then the queue is full.
- Because of the way we check if the queue is full, we lose one slot in the buffer, so the real capacity is `capacity - 1`.
- We will consider the queue as full, when there is just one empty slot.

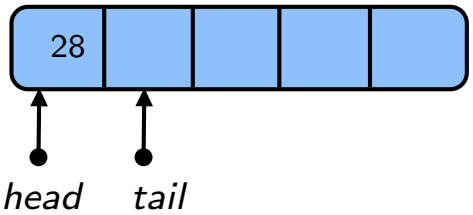
Queue empty



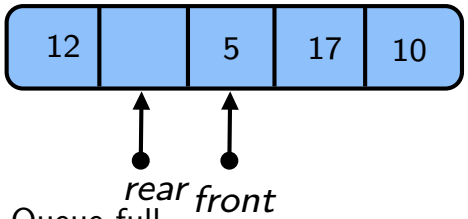
pop(); pop();



`push(28)`

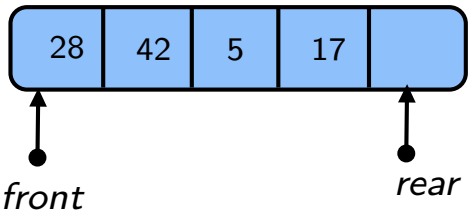


`push(10);push(12);`



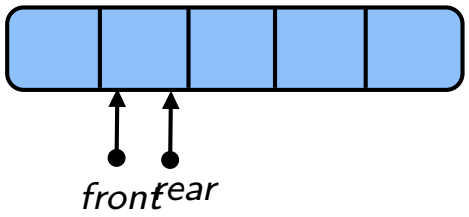
Queue full

`push(42);push(5);push(17);`



Queue full

`pop(); pop(); pop(); pop();`



Queue empty

Basic Thread-safe queue interface

Interface

```
namespace dev{
    template<typename T>
    class threadsafe_queue{
        private:
        std::vector<T> m_buffer;
        size_t m_head;
        size_t m_tail;
        size_t m_capacity;
        mutable std::shared_mutex m_mutex;
        std::condition_variable not_empty_;
        std::condition_variable not_full_;

        bool is_empty_helper();
        size_t next(size_t idx);
        bool is_full_helper();
        std::size_t size_helper();

        public:
        using value_type = T;
        using reference = T&;
        using const_reference = const T&;

        explicit threadsafe_queue(size_t capacity)
        : m_buffer{m_capacity}
        , m_head{0}
        , m_tail{0}
        , m_capacity{capacity}
        {}
        /* continued ... */
    }
}
```

Interface(contd...)

```
namespace dev{
    template<typename T>
    class threadsafe_queue{
        /* continued ... */
        threadsafe_queue(const threadsafe_queue&
        ↪  other) = delete;
        threadsafe_queue& operator=(const
        ↪  threadsafe_queue&) = delete;

        value_type front();
        value_type back();
        bool is_empty();
        bool is_full();
        std::size_t size();

        // non-blocking
        bool try_push(const_reference item);

        // blocking
        void push(const_reference item);

        // non-blocking
        bool try_pop();

        // blocking
        bool pop();
    };
}
```

Basic thread-safe queue - private helper functions

```
/* Private helper functions */
    bool is_empty_helper(){
        return m_head == m_tail;
    }

    size_t next(size_t idx){
        return (idx + 1) % m_capacity;
    }

    bool is_full_helper(){
        return next(m_tail) == m_head;
    }

    std::size_t size_helper(){
        return m_tail < m_head ? (m_tail + m_capacity - m_head) : m_tail - m_head;
    }
```

Basic thread-safe queue - getters

```
/* Getter functions */
```

```
value_type front(){
    std::shared_lock<std::shared_mutex> lock_shared{ m_mutex };
    return m_buffer[m_head];
}

value_type back(){
    std::shared_lock<std::shared_mutex> lock_shared{ m_mutex };
    return m_buffer[m_tail - 1];
}

bool is_empty(){
    std::shared_lock<std::shared_mutex> lock_shared{ m_mutex };
    return is_empty_helper();
}

bool is_full(){
    std::shared_lock<std::shared_mutex> lock_shared{ m_mutex };
    return is_full_helper();
}

std::size_t size(){
    std::shared_lock<std::shared_mutex> lock_shared{ m_mutex };
    return size_helper();
}
```

Basic thread-safe queue - push, pop

/* push/pop functions */

```
// blocking
void push(const_reference item){
    std::unique_lock<std::mutex> unique_lck(m_mutex);
    not_full_.wait(unique_lck, [this]{ return !is_full_helper(); });

    m_buffer[m_tail] = item;
    m_tail = next (m_tail);

    unique_lck.unlock();
    not_empty_.notify_one();
}

// blocking
value_type pop(){
    std::unique_lock<std::mutex> unique_lck(m_mutex);
    not_empty_.wait(unique_lck, [this](){ return !is_empty_helper(); });

    m_head = next(m_head);

    unique_lck.unlock();
    not_full_.notify_one();

    return true;
}
```

Basic thread-safe queue - try_push, try_pop

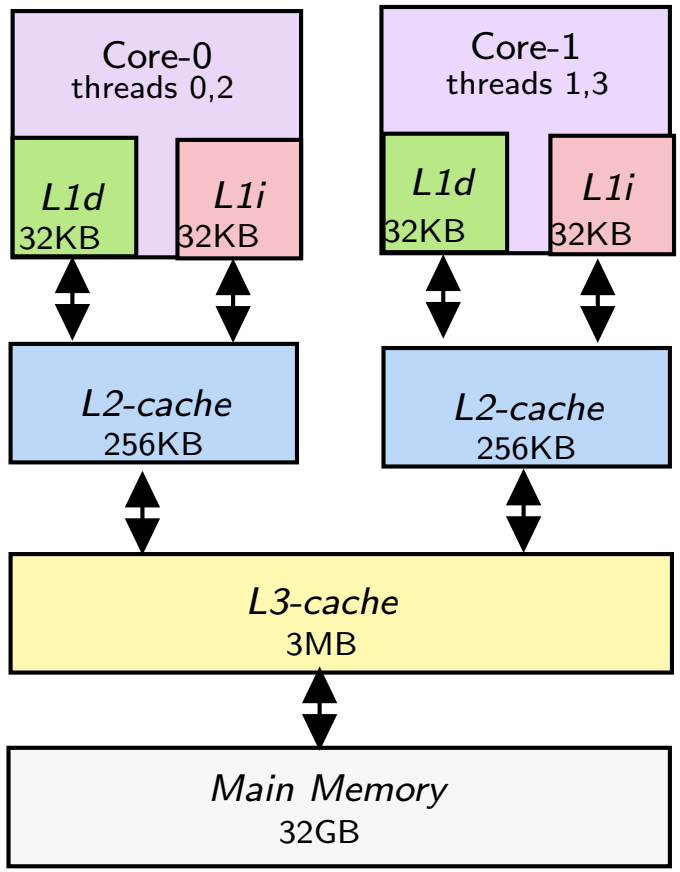
/* try_push and try_pop functions */

```
// non-blocking
bool try_push(const_reference item){
    std::unique_lock<std::mutex> unique_lck(m_mutex, std::try_to_lock);
    if(!unique_lck || is_full_helper())
        return false;

    m_buffer[m_tail] = item;
    m_tail = next (m_tail);

    unique_lck.unlock();
    not_empty_.notify_one();
    return true;
}

// non-blocking
bool try_pop()
{
    std::unique_lock<std::mutex> unique_lck(m_mutex, std::try_to_lock);
    if(!unique_lck || is_empty_helper())
        return false;

    value_type item = m_buffer[m_head];
    m_head = next(m_head);

    unique_lck.unlock();
    not_full_.notify_one();
    return true;
}
```

Compiler Explorer : https://compiler-explorer.com/z/Gz1vMGcno

Fun with Cache lines

- Modern systems make use of multi-level caching for faster memory access. There are usually 3 layers of cache - $L1$, $L2$ and $L3$.
- Each cache is exponentially larger than the other ($L3 > L2 > L1$). It leads to an exponentially larger memory access time.
- On most systems, each core has its own $L1$ data and $L1$ instruction cache. Some systems like the one below may have an $L2$ cache per core, while an $L3$ cache is almost always shared amongst the cores.
- Whenever a memory address is requested, the CPU checks the $L1$ cache first. If the data is found, it uses that (ignoring modification by other threads for now). If the data is not found, it looks for it in the deeper caches and eventually in the main memory.
- The CPU then copies over the data to the $L1$ cache. Interestingly, the CPU usually just does not copy the data requested, but instead copies the nearby data as well expecting the user to request that data in future. This is called **spatial locality**.



Fun with Cache lines

- Perform a simple experiment. We have a function that accesses every Nth element. We execute `BM_access_Nth_element` and microbenchmark it for step-sizes $1, 2, 4, 8, 16, 32, 64, \ldots$.
- We find that accessing every element roughly takes the same number of CPU cycles as access every 16th element.
- Thus, memory comes in cache lines. We don't just access an individual `int`, if we want something from memory, we bring the whole cache line in, into $L1$ cache. If you touch one `int` on the cache line, you get all the other `int`s on that cache line for free.

/* Cache line size */

```
#include <benchmark/benchmark.h>
#include <vector>
#include <algorithm>

static void BM_access_Nth_element(benchmark::State& state) {
    for (auto _ : state) {
        constexpr size_t size = 2 << 23;    // 32 MB
        std::vector<int> v(size);
        std::size_t step{state.range(0)};
        std::generate(v.begin(), v.end(), std::rand);
        size_t num_elements{0};
        for(int i{0}; num_elements < 1000; i+= step)
        {
            v[i]++;
            ++num_elements;
        }

        benchmark::DoNotOptimize(v);
    }
}
BENCHMARK(BM_access_Nth_element)->RangeMultiplier(2)->Range(1,
↪  1<<8);
BENCHMARK_MAIN();
```

```
-------------------------------------------------------------
Benchmark                             Time             CPU
-------------------------------------------------------------
BM_access_Nth_element/1   1111128330 ns    610593436 ns
BM_access_Nth_element/2   1105773736 ns    605356946 ns
BM_access_Nth_element/4   1094496578 ns    590871063 ns
BM_access_Nth_element/8   1099342032 ns    596099320 ns
BM_access_Nth_element/16  1086154354 ns    592554463 ns
BM_access_Nth_element/32  1613422626 ns    611174008 ns
BM_access_Nth_element/64  1103676115 ns    604367120 ns
BM_access_Nth_element/128 1058715276 ns    560294304 ns
BM_access_Nth_element/256  992869876 ns    492471902 ns
```

Compiler Explorer : https://compiler-explorer.com/z/YxWdYeGca

C++ atomics

- Atomic operations are indivisible. An atomic operation is any operation that is **guaranteed to execute as a single transaction**.
- At a low-level, atomic operations are special hardware instructions.
- Consider, for instance a shared variable `counter` that is initialized to `0`. Consider the assembly instructions corresponding to the increment operation `count++`.

```cpp
// Incrementing a counter
int counter {0};

int main(){
    counter++;
    return 0;
}
```

- Generated Assembly :

```
counter:
        .zero   4
main:
        push    rbp
        mov     rbp, rsp
        mov     eax, DWORD PTR counter[rip]
        add     eax, 1
        mov     DWORD PTR counter[rip], eax
        mov     eax, 0
        pop     rbp
        ret
```

C++ atomics

- The following code does the same: it increments a global counter. This time, though, we use an atomic type and operations.

```
#include <atomic>
std::atomic<int> counter {0};

int main(){
    counter++;
    return 0;
}
```

- Generated Assembly :

```
lock add         DWORD PTR counter[rip], 1
```

- Atomic operations allow threads to read, modify and write indivisibly and can also be used as synchronization primitives. Atomic operations must be provided by the CPU (as in the `lock add` instruction).

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

- Explicit reads and writes:

```
/* Atomic reads and writes */
std::atomic<T> x;
T y = x.load();      // Same as T y = x;
x.store(y);          // Same as x = y;
```

- Atomic exchange:

```
/* Atomic exchange */
T z = x.exchange(y);     // Atomically: z = x; x = y;
```

- exchange is an atomic swap. It's a read-modify-write done atomically. It reads the old value, replaces it with the new value and guarantees that nobody can get in there in between.
- Compare-and-swap (conditional exchange):

```
/* Atomic CAS */
bool success = x.compare_exchange_strong(y,z);      // T& y
// var.compare_exchange_strong(expected,desired);
// If x == y, x = z and return true
// Otherwise, set y = x and return false
```

Why is CAS so special?

- Compare-and-swap(CAS) is used in most lock-free algorithms.
- Example : atomic increment with CAS:

```
/* CAS Loop */
    std::atomic<int> x{0};
    int x0 = x.load();      // [1]
    while(!x.compare_exchange_swap( x0, x0 + 1 )){}  // [2]
```

- Pretty much, every lock-free algorithm is centered around a loop like this. So, we want to increment `x`. First of all, at [1], I will read the atomic value and store it in a local `x0`. I hope nobody got to `x` before me, `x` hasn't changed. If that's `true`, I am going to change it atomically to the desired value (that could be an increment, decrement, multiplication by 2 etc). If nobody else changed `x`, I did my increment atomically. CAS returns `true`. `while(<pred>)` predicate is false and the loop ends.
- If somebody did change `x`, CAS fails and returns `false`. The changed value of `x` is updated in `x0`, so I don't have to the read again. And I go on the next iteration of the loop and keep trying, until my compare-and-swap beats everyone else' compare-and-swap and gets that increment in.

What other operations can be done on `std::atomic<T>`

- For integer `T`:

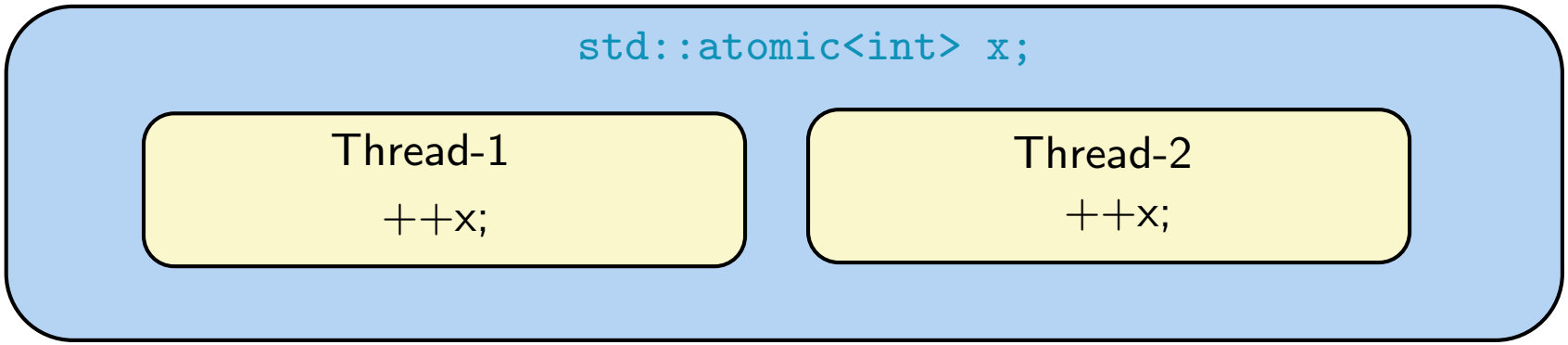
```
std::atomic<int> x;
x.fetch_add(y);      // Same as x += y;
```

- `fetch_add()` doesn't just add atomically. It increments atomically, but also returns to you the old value (which is the fetch part). So, it returns the old value and adds the increment, all of it atomically.
- Also, `fetch_sub`, `fetch_and()`, `fetch_or()` and `fetch_xor()`.
- If you have multiple of these atomic operations, its composition is not.

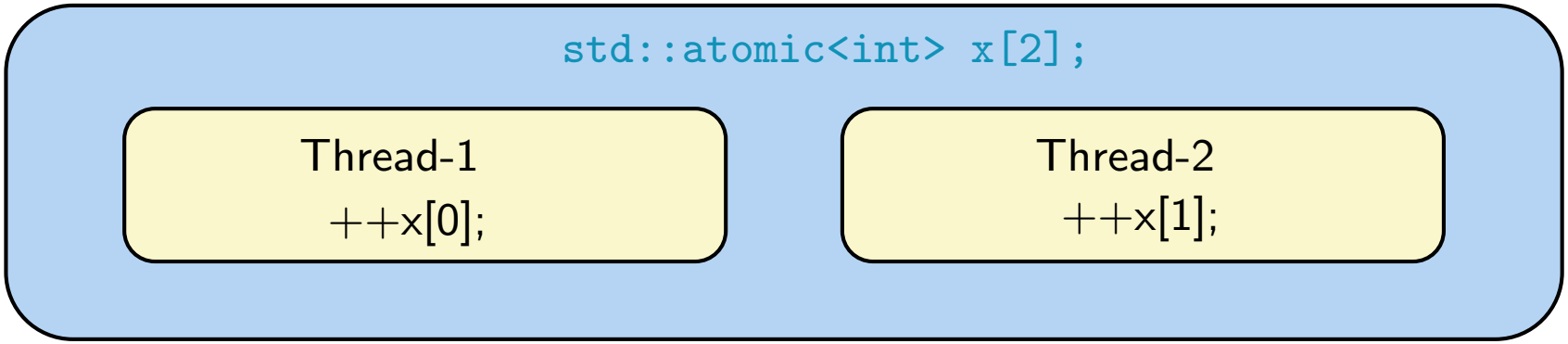
Do atomic operations wait on each other?

- Atomic operations are lock-free, maybe even wait-free. It doesn't mean they don't wait on each other.

Accessing shared variable



Accessing Non-shared variable



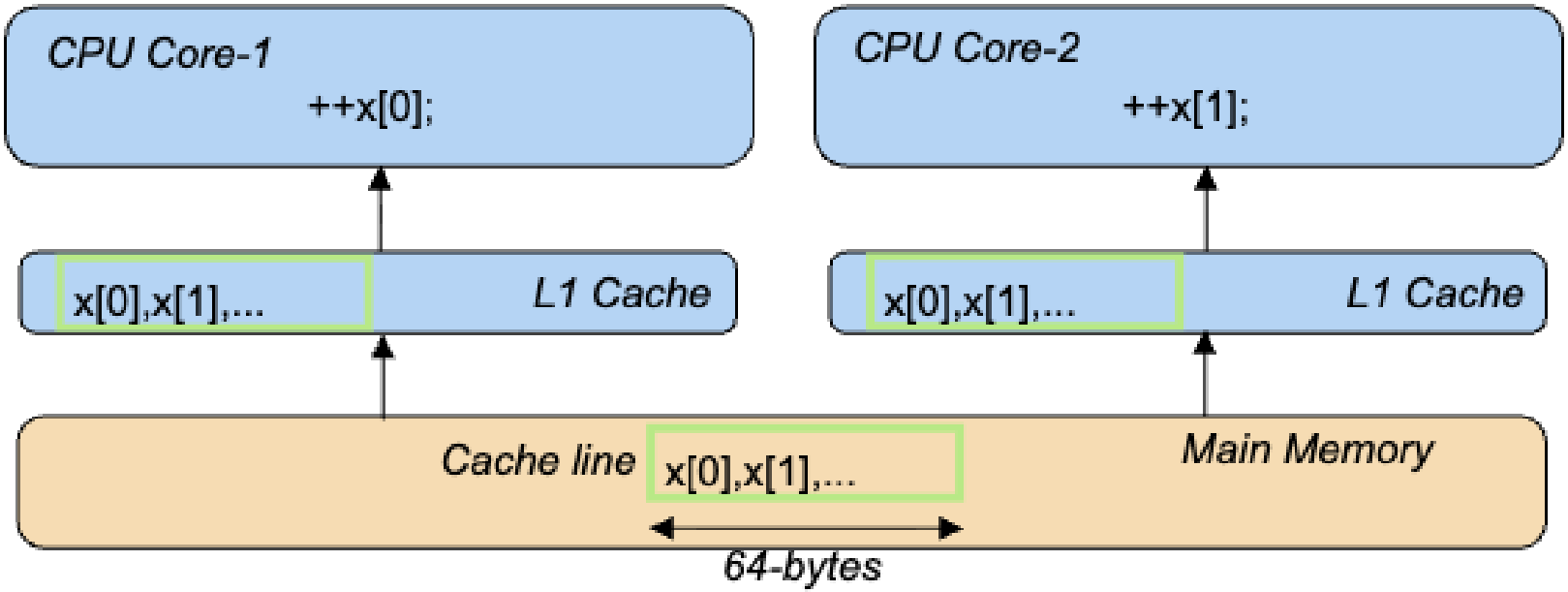
Credit. C++ atomics, from basic to advanced. What do they really do? by Fedor Pikus, CppCon 2017

Do atomic operations wait on each other?

```
[quantdev@quasar-arch atomic_bench]$ g++ main.cpp -std=c++20 -O3 -pthread -lbenchmark -o atomic_bench
[quantdev@quasar-arch atomic_bench]$ ./atomic_bench
2025-11-08T20:13:41+00:00
Running ./atomic_bench
Run on (22 X 4800 MHz CPU s)
CPU Caches:
  L1 Data 48 KiB (x11)
  L1 Instruction 64 KiB (x11)
  L2 Unified 2048 KiB (x11)
  L3 Unified 24576 KiB (x1)
Load Average: 0.07, 0.09, 0.06
***WARNING*** CPU scaling is enabled, the benchmark real time measurements may be noisy and will incur extra ove
-----------------------------------------------------------------------------------------
Benchmark                                       Time             CPU   Iterations UserCounters...
-----------------------------------------------------------------------------------------
BM_SharedAtomicIncrement/1/real_time         0.558 ms        0.015 ms         1169 items_per_second=179.297M/s
BM_SharedAtomicIncrement/2/real_time          2.56 ms        0.044 ms          282 items_per_second=78.101M/s
BM_SharedAtomicIncrement/4/real_time          5.40 ms        0.096 ms          100 items_per_second=74.1264M/s
BM_SharedAtomicIncrement/8/real_time          10.6 ms        0.218 ms           66 items_per_second=75.1328M/s
BM_SharedAtomicIncrement/16/real_time         22.9 ms        0.401 ms           30 items_per_second=69.8455M/s
BM_SharedAtomicIncrement/32/real_time         48.4 ms         1.14 ms           14 items_per_second=66.1447M/s
BM_SeparateAtomicIncrement/1/real_time       0.556 ms        0.016 ms         1205 items_per_second=179.897M/s
BM_SeparateAtomicIncrement/2/real_time        2.74 ms        0.033 ms          262 items_per_second=73.0452M/s
BM_SeparateAtomicIncrement/4/real_time        5.74 ms        0.081 ms          112 items_per_second=69.6313M/s
BM_SeparateAtomicIncrement/8/real_time        11.2 ms        0.244 ms           63 items_per_second=71.3645M/s
BM_SeparateAtomicIncrement/16/real_time       23.9 ms        0.403 ms           29 items_per_second=66.8318M/s
BM_SeparateAtomicIncrement/32/real_time       26.6 ms         1.23 ms           26 items_per_second=120.25M/s
```

Do atomic operations wait on each other?

- Can we conclude that atomic operations don't wait on each other from this? Not necessarily! What's going on really?
- The two atomic operations are in the same cache-line. On $x86$, the whole cache line trickles up and down from the main-memory to the on-board CPU cache and back.
- Even if you want one variable from the cache line - the entire 64-byte chunk will go up and down.
- And if two different CPUs want two different variables within the same cache-line, they need to wait, as if it was the same variable. You don't get a lower granularity than 64-bytes on x86.



Credit. C++ atomics, from basic to advanced by Fedor Pikus

Do atomic operations wait on each other?

- Atomic operations do wait on each other.
  - In particular, write operations do.
  - Read-only operations can scale near-perfectly.
- We run a micro-benchmarking test, this time, incrementing truly non-shared variables that are on different cache lines.

```
-------------------------------------------------------------------------------
Benchmark                             Time             CPU   Iterations UserCounters...
-------------------------------------------------------------------------------
BM_Shared/1/real_time              0.561 ms         0.015 ms         1249 items_per_second=178.278M/s
BM_Shared/2/real_time               3.25 ms         0.054 ms          210 items_per_second=61.4546M/s
BM_Shared/4/real_time               6.07 ms         0.061 ms          121 items_per_second=65.8638M/s
BM_Shared/8/real_time               10.6 ms         0.240 ms           64 items_per_second=75.1448M/s
BM_Shared/16/real_time              22.9 ms         0.384 ms           30 items_per_second=69.8626M/s
BM_Shared/32/real_time              51.4 ms         0.944 ms           13 items_per_second=62.2302M/s
BM_Shared/64/real_time               103 ms          2.67 ms            7 items_per_second=62.2315M/s
BM_FalseShared/1/real_time         0.578 ms         0.016 ms         1264 items_per_second=172.877M/s
BM_FalseShared/2/real_time          2.98 ms         0.037 ms          245 items_per_second=67.1271M/s
BM_FalseShared/4/real_time          6.18 ms         0.065 ms          123 items_per_second=64.6853M/s
BM_FalseShared/8/real_time          10.6 ms         0.234 ms           64 items_per_second=75.6088M/s
BM_FalseShared/16/real_time         13.5 ms         0.386 ms           51 items_per_second=118.33M/s
BM_FalseShared/32/real_time         26.7 ms          1.21 ms           27 items_per_second=119.802M/s
BM_FalseShared/64/real_time         37.0 ms          2.36 ms           18 items_per_second=172.999M/s
BM_NonShared/1/real_time           0.576 ms         0.015 ms         1159 items_per_second=173.597M/s
BM_NonShared/2/real_time           0.591 ms         0.028 ms         1192 items_per_second=338.258M/s
BM_NonShared/4/real_time           0.632 ms         0.051 ms         1023 items_per_second=632.886M/s
BM_NonShared/8/real_time           0.598 ms         0.102 ms         1247 items_per_second=1.33678G/s
BM_NonShared/16/real_time          1.05 ms          0.225 ms          676 items_per_second=1.51828G/s
BM_NonShared/32/real_time          1.81 ms          0.701 ms          384 items_per_second=1.76598G/s
BM_NonShared/64/real_time          3.21 ms           1.28 ms          217 items_per_second=1.995G/s
```

Do atomic operations wait on each other?

- Atomic operations have to wait for cache line access.
  - This is **price of data sharing** without race conditions.
  - Modifying different locations on the same cache line still incurs run-time penalty (false sharing).
- Avoid false sharing by aligning per-thread data to separate cache lines.

Strong and weak compare-and-swap

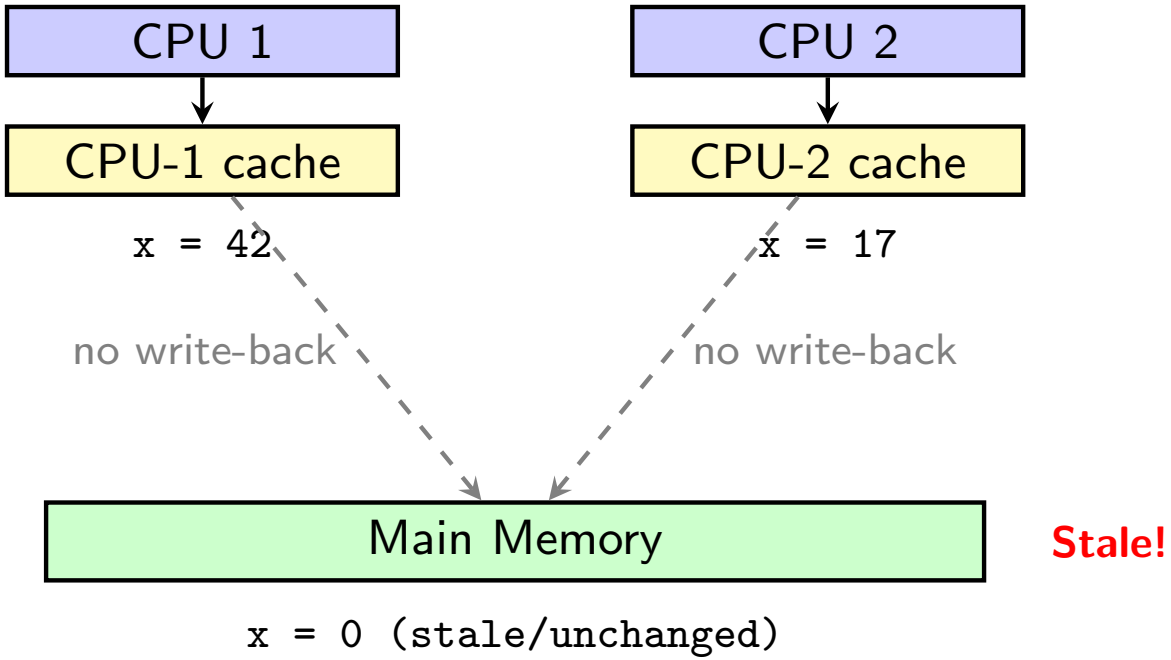
- C++ provides two versions of CAS - weak and strong.

```
/* CAS 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` is essentially the same thing, but can spuriously fail and return `false`, even if `x == old_x`.

Memory barriers - the twin of C++ atomics

- Memory barriers go hand-in-hand with C++ atomics. Memory barriers control how changes to memory made by one CPU core becomes visible to other CPU cores.
- If you don't have memory barriers, there is no guarantee of visibility whatsoever. item Imagine you have two CPUs, both modifying a variable `x` in their on-chip caches. The main memory doesn't have to change at all! There is no guarantee that anybody can see anything.



Stale!

Problem: Each CPU sees its own value. Memory is unchanged. No coherence!

Memory barriers in C++

- C++ memory barriers are modifiers on the atomic operations.
- Example:

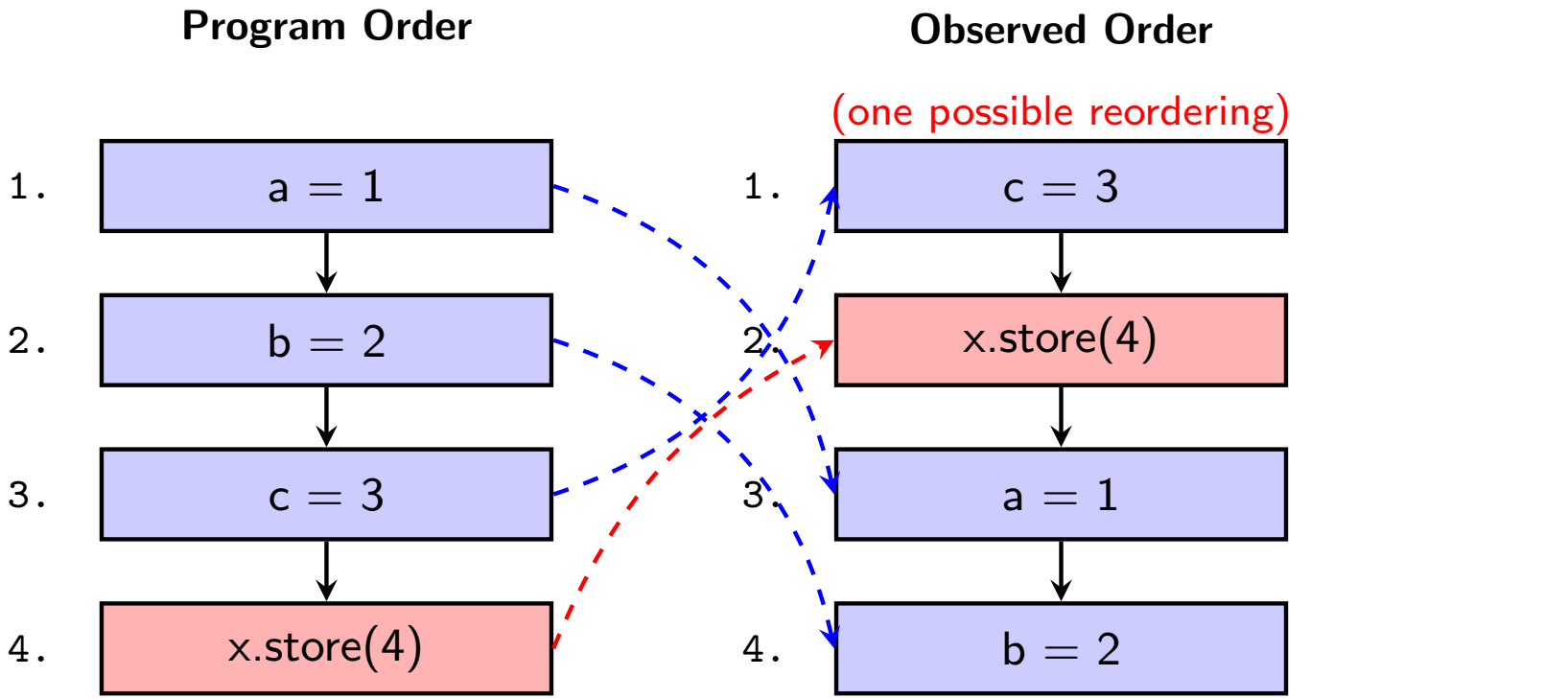
```
std::atomic<int> x;
x.store(1, std::memory_order_release);
```

- This implies that I have put a release memory barrier on that store.

No barriers - std::memory_order_relaxed

- No memory barrier means, that I can reorder reads and writes anyway I want.
- I have an atomic x variable and a, b and c are non-atomic variables. In the program order, I write to a, then I write to b, then I write to c and then I write to x. The observed order could be anything.

```
x.fetch_add(1, std::memory_order_relaxed);
```



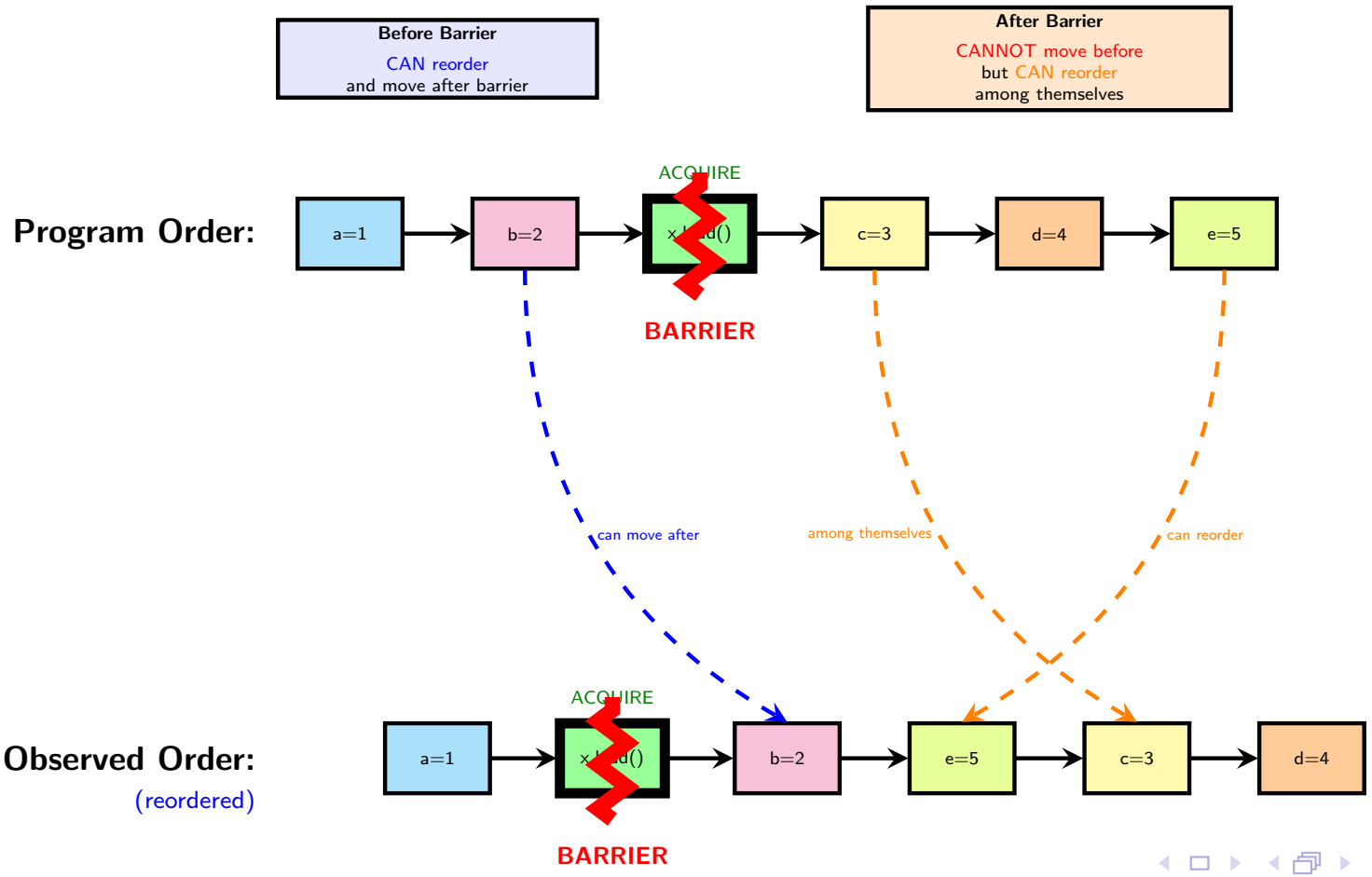
No Memory Barrier

Non-atomic variables (a, b, c) can be reordered freely.

Atomic variable (x) with memory_order_relaxed also provides no ordering guarantees.

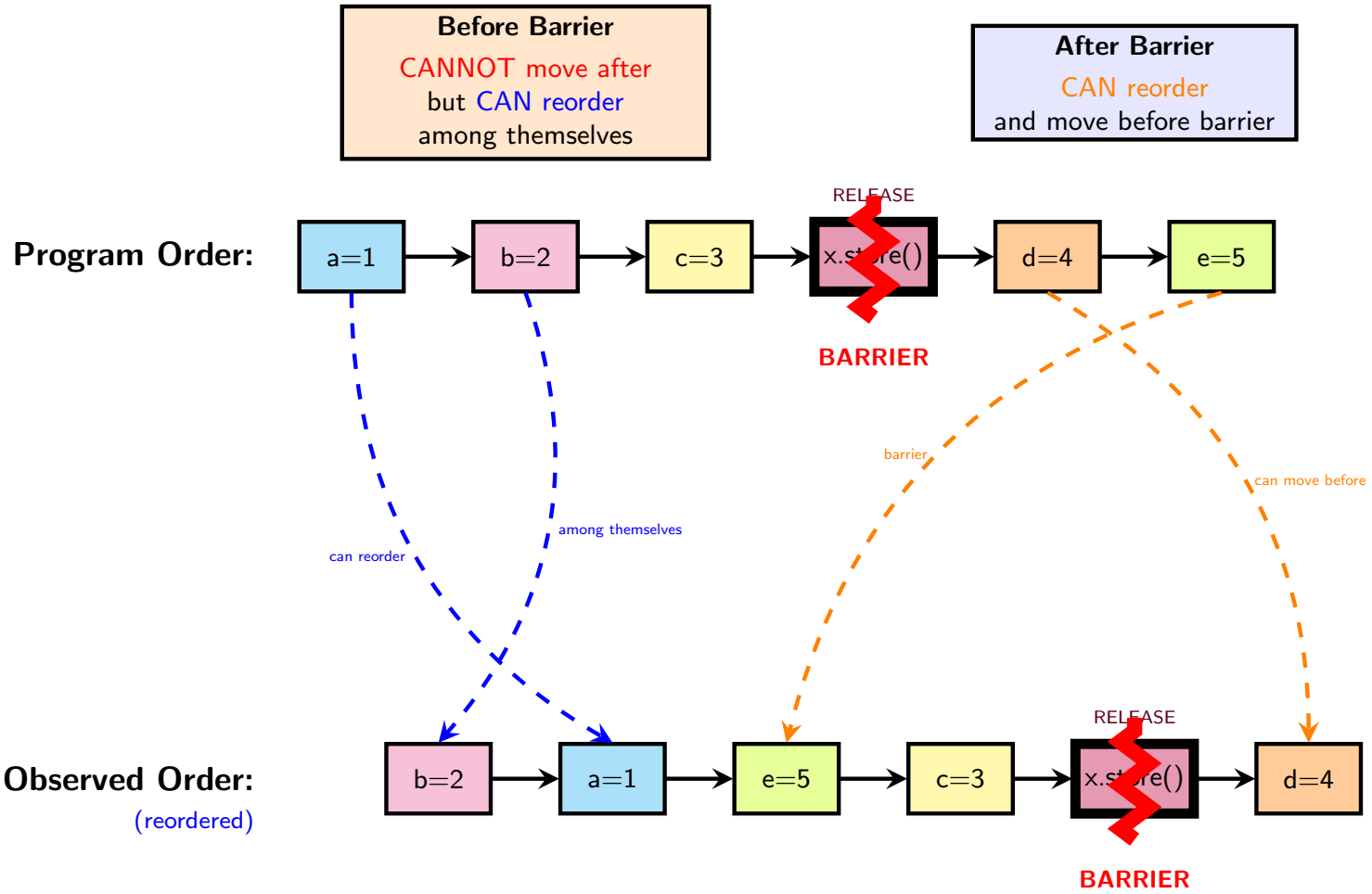
Acquire barrier

- Acquire barrier is a half-barrier. It acts as a one-way gate. Nothing that was after the `load` can move in front of it. Anything that was before can move after.
- 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*, but both reads and writes.
  - **All operations** not just operations on the atomic variable variable that the barrier was on, but literally **all operations**.
- Reads and writes cannot be reordered from after to before the barrier.



Release barrier

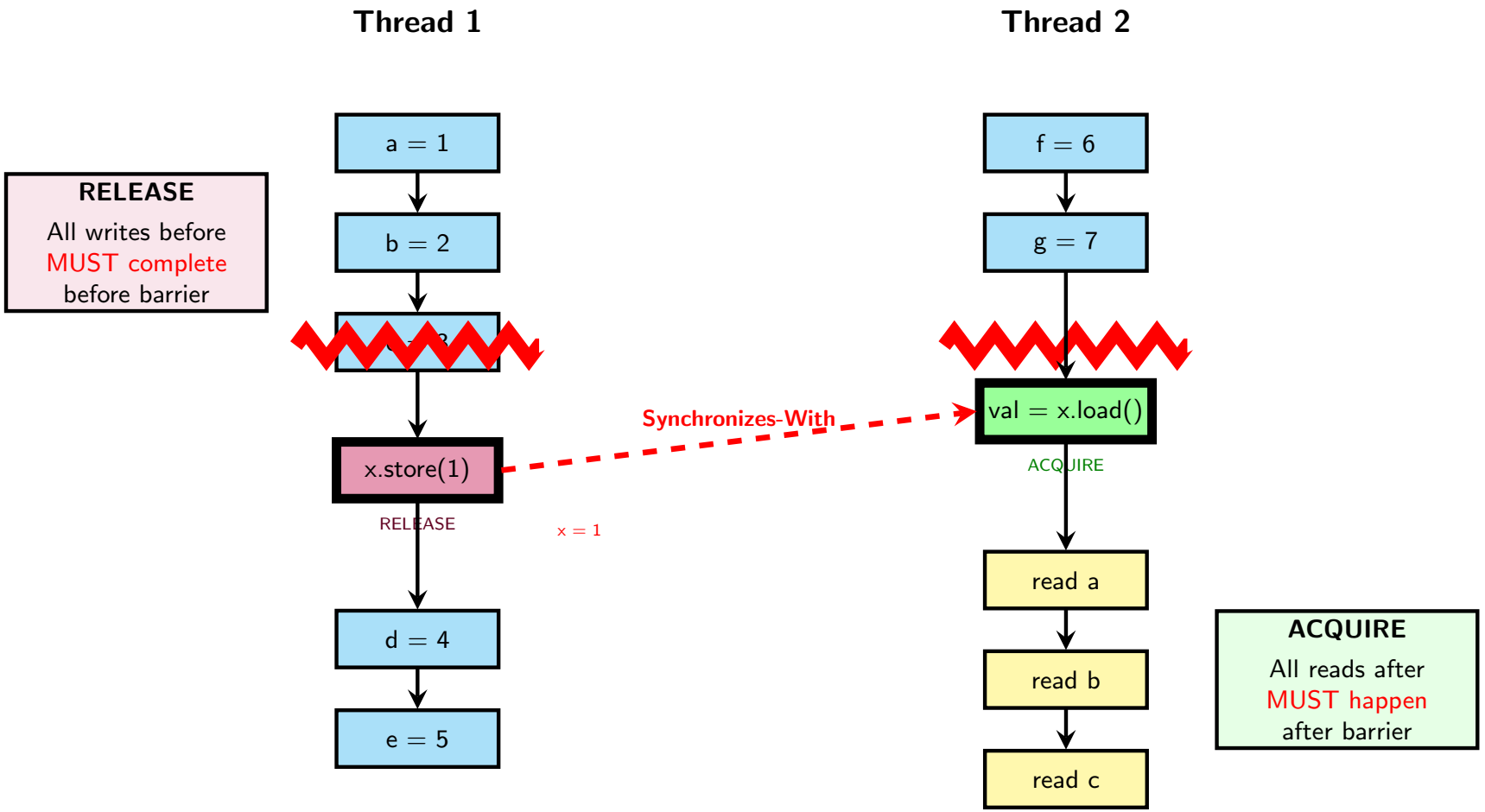
- It makes sense that the release barrier is the reverse.
- Nothing that was before the barrier can move after. Anything that is after can move in front of the `load`.



Acquire-release order

- Acquire and release barriers are often used together. That's the acquire-release protocol. Think about it!
- Thread $t1$ writes atomic variable `x` with `release` barrier.
- Thread $t2$ reads atomic variable `x` with `acquire` barrier.
- I have a guarantee on the acquire side. All the memory reads done after the `acquire` barrier in $t2$ in program order have to be done after the barrier in actual real execution order.
- I also have a guarantee on the release side. All the memory writes done before the `release` barrier in $t1$ in program order have to be done before the barrier in actual real execution order.
- All memory writes that happen in $t1$ before the barrier (in program order) become visible in thread $t2$ 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.
- **Note**. It has to be the same atomic variable `x`.

Acquire-release order



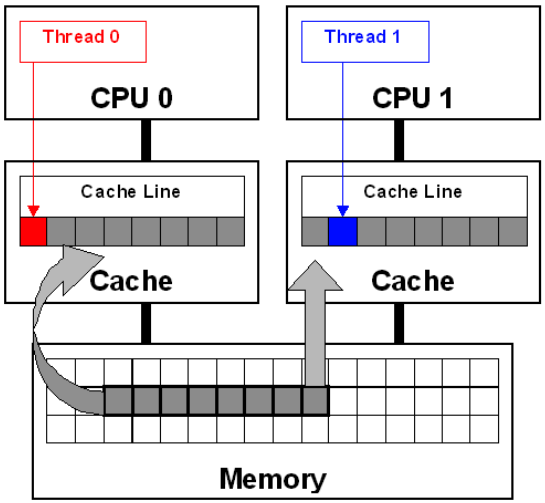
Acquire-Release Protocol Guarantee:

Thread 1: Prepares data (a, b, c), then publishes via `x.store(1, release)`
Thread 2: Acquires via `x.load(acquire)`, then safely reads data

⇒ All writes before release in T1 are visible after acquire in T2
⇒ Must be the same atomic variable `x`

Cache Coherency

- A cache-line is typically 64-bytes on most machines.
- If cache line X is in the cache of multiple cores, and one of these cores mutates its data, then the change must be reflected in all the cores owning X via a cache coherency mechanism
- For simplicity, suppose there are cores - A and B. Further assume, that core A reads and writes to a variable a, core B reads and writes to a variable b, and a and b are close to each other in main memory - they are on the same cache line.
- Core A first reads the value of a from main memory. It therefore loads the entire cache line and marks it as exclusive access as it is the only core operating on this cache line. Core B decides to read the value of b. Since, a and b are close and reside on the same cache line, Core B loads the same cache line and both cores tag their cache lines as shared access.
- Now, let's suppose core A decides to change the value of a. The core A stores this change only in its store buffer and marks its cache line as modified. It also communicates this change to core B, and this core in turn will mark its cache as invalidated.
- That's how different cores ensure their caches are coherent with each other.



Designing the SPSC lock-free queue data-structure

- Lock-free queues are generally **bounded** queues.
- We use a power of 2 buffer-size. This is because when calculating the next index for any buffer size `N`, the modulo (`%`) operator requires a division instruction, which requires several CPU cycles. When the capacity is a power of 2, we can just do the following:
  `size_t next_index = curr_index & (m_capacity - 1)`
- **False Sharing**. False-sharing occurs when threads on different processors modify different variables residing on the same cache-line.
- To avoid false-sharing, the `std::hardware_destructive_interference_size` constant defined in the thread header, is used to determine the cache-line size and is to be used with `alignas()`.

example_12.cpp

```cpp
template<Queueable T, std::size_t N>
    class spsc_queue{
        private:
        using size_type = std::size_t;
        using value_type = T;
        using reference = T&;

        static constexpr std::size_t m_capacity {1 << N};
        T m_buffer[m_capacity];
        alignas(std::hardware_destructive_interference_size) std::atomic<std::size_t> m_read_index{ 0 };
        alignas(std::hardware_destructive_interference_size) std::atomic<std::size_t> m_write_index{ 0 };
    };
```

Implementing try_push and try_pop methods

- try_push

example_12.cpp

```cpp
template<typename U>
requires std::is_convertible_v<T,U>
bool try_push(U&& element){
    const std::size_t write_index = m_write_index.load(std::memory_order_relaxed);
    const std::size_t next_write_index = (write_index + 1) & (m_capacity - 1);

    if(next_write_index != m_read_index.load(std::memory_order_acquire))
    {
        m_buffer[write_index] = std::forward<U>(element);
        m_write_index.store(next_write_index, std::memory_order_release);
        return true;
    }
    return false;
}
```

- try_pop

example_12.cpp

```cpp
std::optional<T> try_pop(){
    std::optional<T> result{std::nullopt};
    const std::size_t read_index = m_read_index.load(std::memory_order_relaxed);

    if(read_index == m_write_index.load(std::memory_order_acquire))
        return result;

    result = std::move(m_buffer[read_index]);
    m_read_index.store((read_index + 1 ) & (m_capacity - 1), std::memory_order_release);
    return result;
}
```