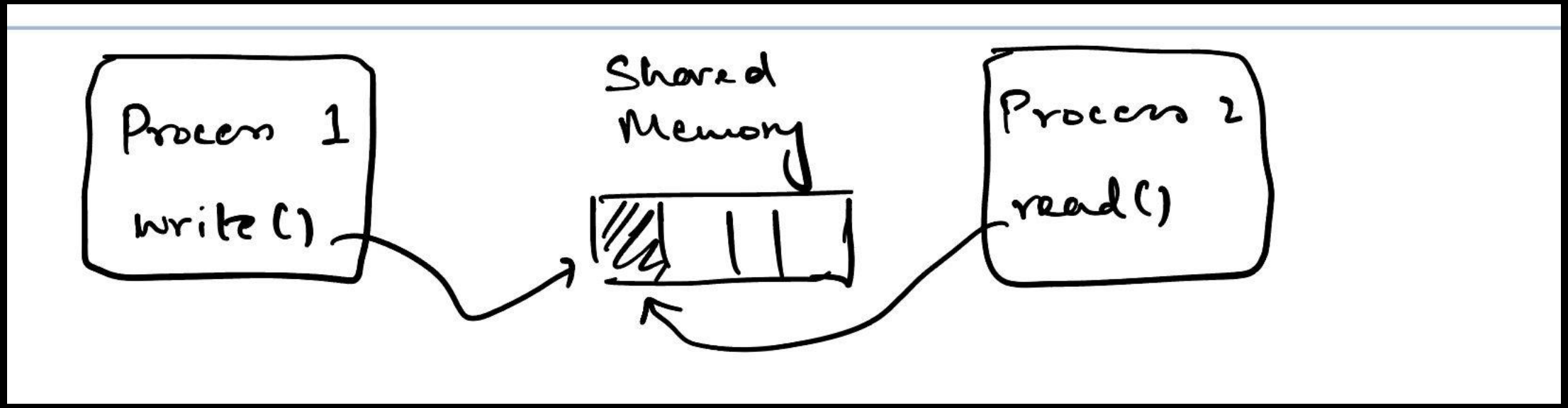
Atomic

привычный способ работы с разделяемой памятью - mutex



атомарные типы данных

```
#include <atomic>

std::atomic<bool> flag(false);

int main() {
    flag.store(true);
    assert(flag.load() == true);
    return 0;
}
```

- Вы можете сохранить в них некоторое значение с помощью метода store(). **Это операция записи.**
- Вы можете загрузить из них какое-либо значение с помощью метода load(). **Это операция чтения.**
- С ними можно выполнять сравнение и набор (CAS) с помощью метода compare_exchange_weak() или compare_exchange_strong(). Это операция чтения-модификации-записи (RMW).

memory ordering

```
// не важен порядок операций
int x = 10;
int y = 5;
```

```
// важен порядок операций
int x = 10;
int y = x  + 5;
```

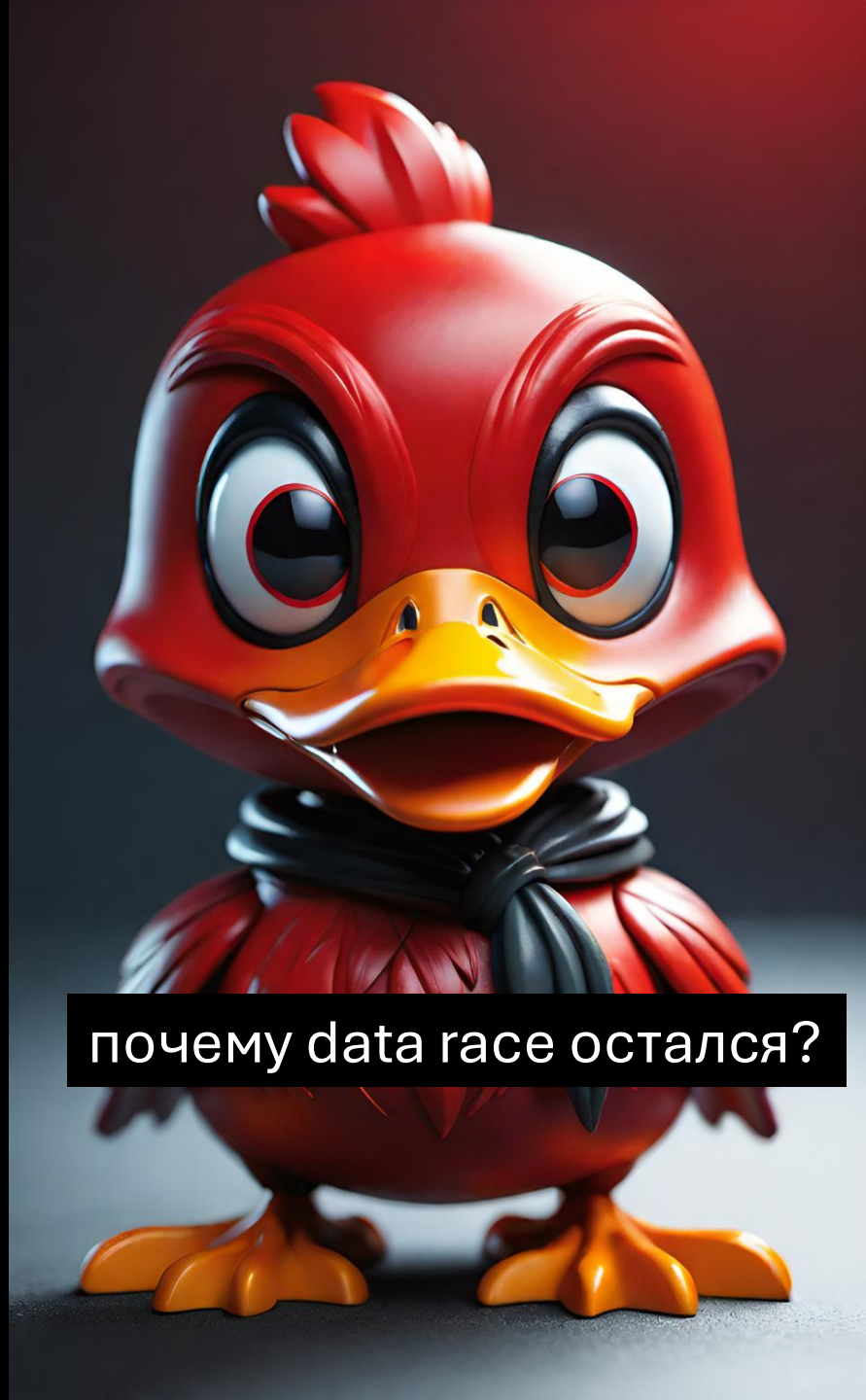
пример

```
#include <cassert>
#include <thread>

int data = 0;

void producer() { data = 100;} // Write data
void consumer() { assert(data == 100); } // Read data

int main() {
        std::thread t1(producer);
        std::thread t2(consumer);
        t1.join();
        t2.join();
        return 0;
}
```

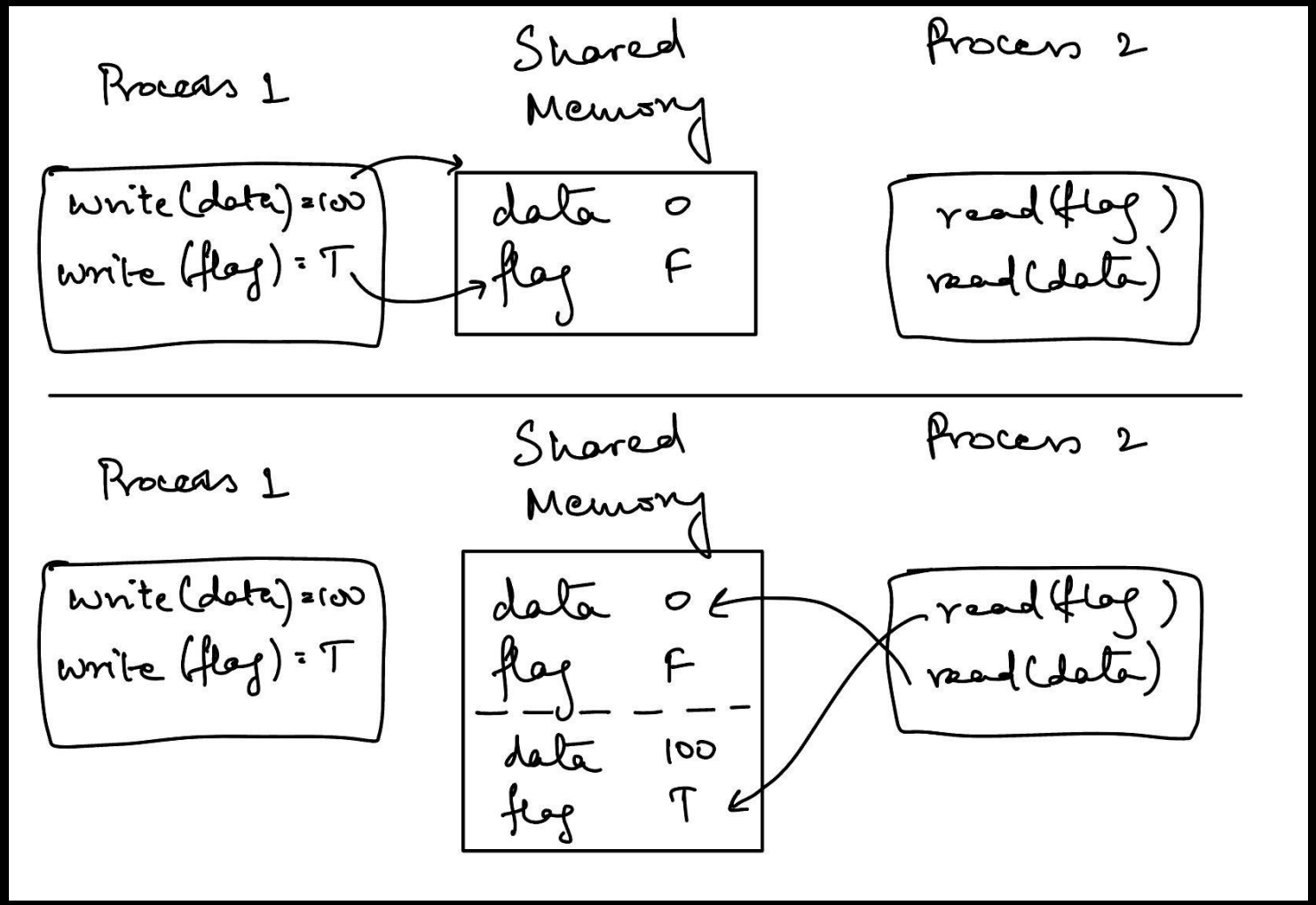
Из за многопоточности мы не знаем в каком порядке будет выполнена операция записи и чтения

попробуем atomic

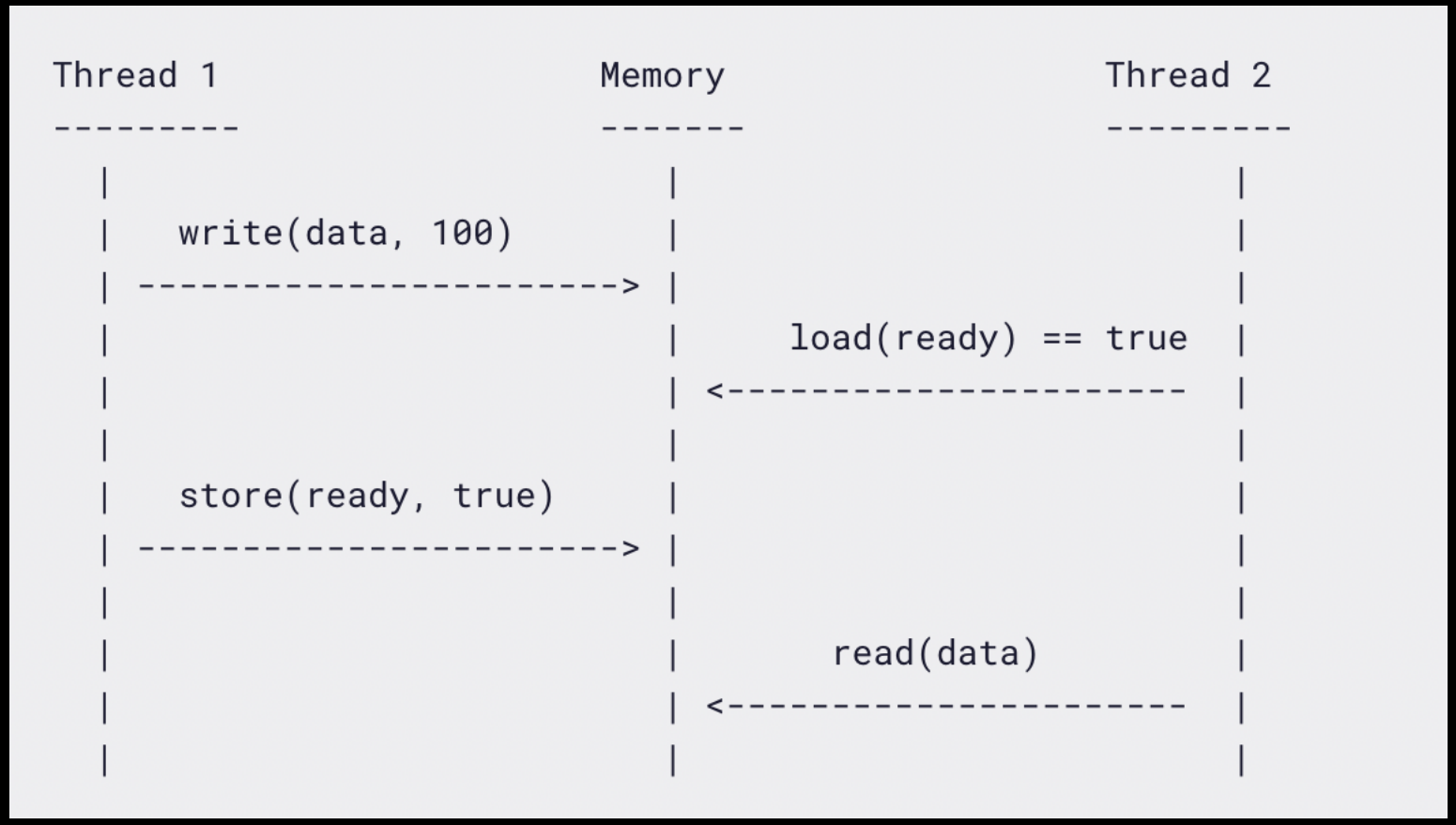
```cpp
#include <atomic>
#include <cassert>
#include <thread>

int data = 0;
std::atomic<bool> ready(false);
void producer() {
        data = 100;
        ready.store(true, std::memory_order_relaxed); // Set flag
}

void consumer() {
        while (!ready.load(std::memory_order_relaxed)) ;
        assert(data == 100);
}

int main() {
        std::thread t1(producer);
        std::thread t2(consumer);
        t1.join();
        t2.join();
        return 0;
}
```



проблема – *Relaxed memory model ordering*



Как сделать правильную последовательность?



Меняем модель памяти

- Было (по умолчанию) `std::memory_order_relax`
- Станет: `std::memory_order_seq_cst`

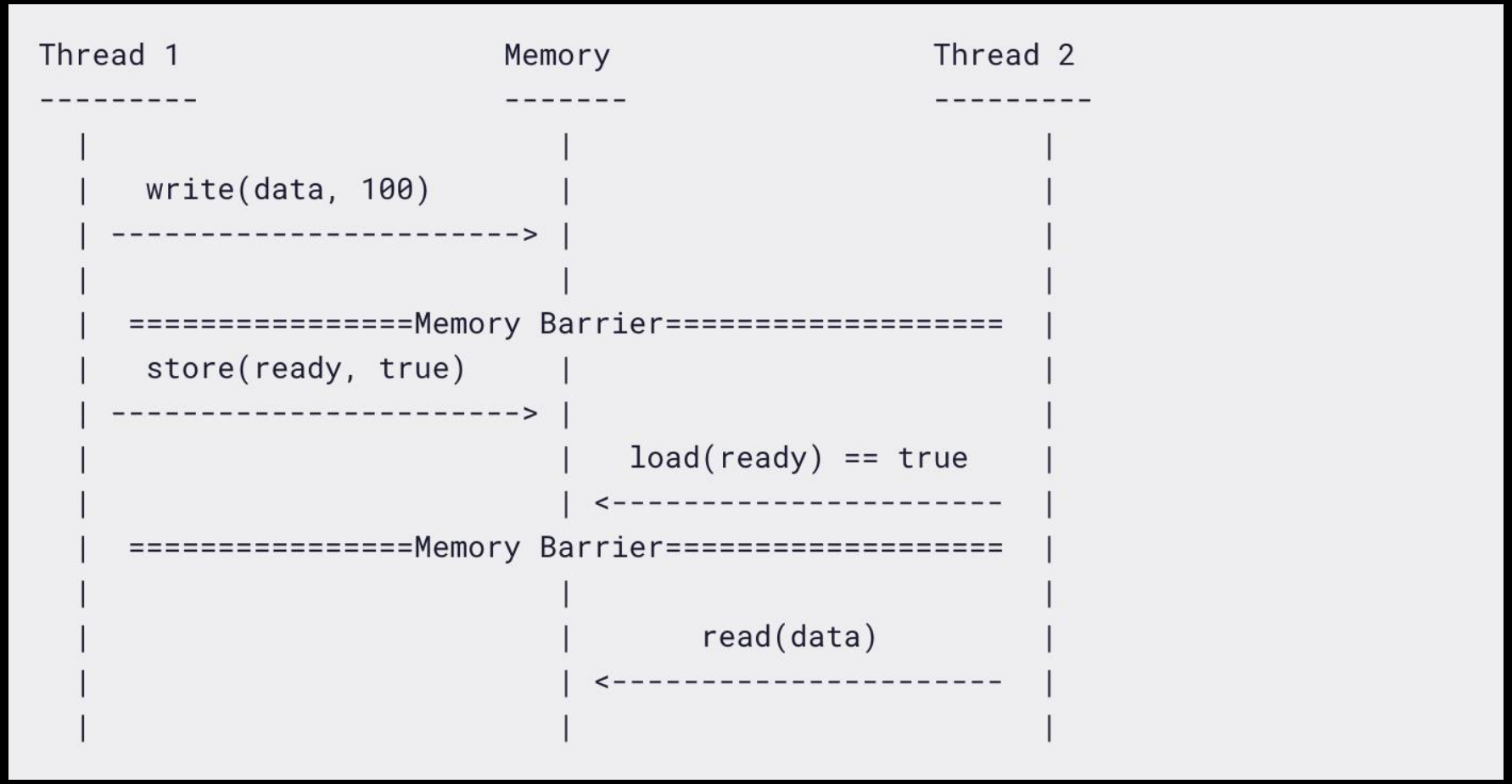
то есть:

```
ready.store(true, std::memory_order_seq_cst);
```

и

```
while (!ready.load(std::memory_order_seq_cst))
```

memory barrier



Types Of Memory Order

1. Relaxed memory model (std::memory_order_relaxed)
2. Release-acquire memory model (std::memory_order_release and std::memory_order_acquire)
3. Sequentially consistent memory order (std::memory_order_seq_cst)

Последовательное (memory_order_seq_cst):

- самое ресурсоемкое
- порядок изменения всех атомарных переменных строго определён
- зато интуитивно понятен – работает так, как написано в коде

Ослабленное

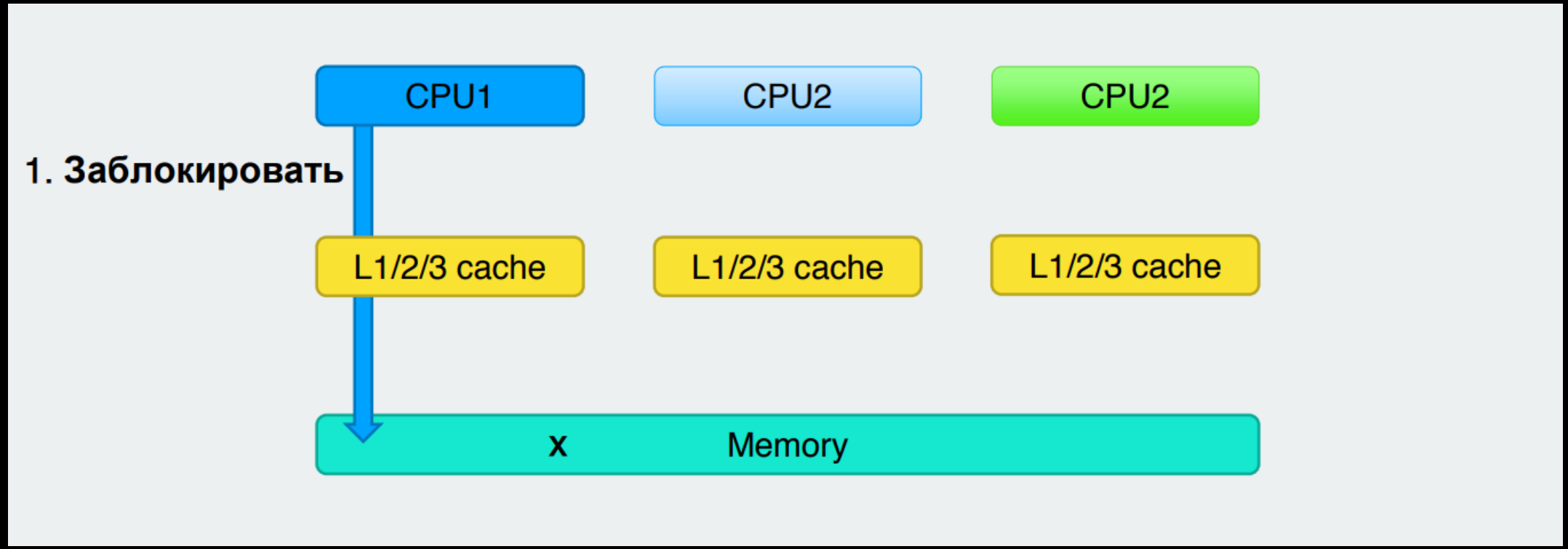
- никаких гарантий синхронизации между потоками
- но, если поток уже считал значение, в следующий раз может быть считано либо то же значение, либо записанное после
- у каждого потока свой порядок обращений

Release-Aquire

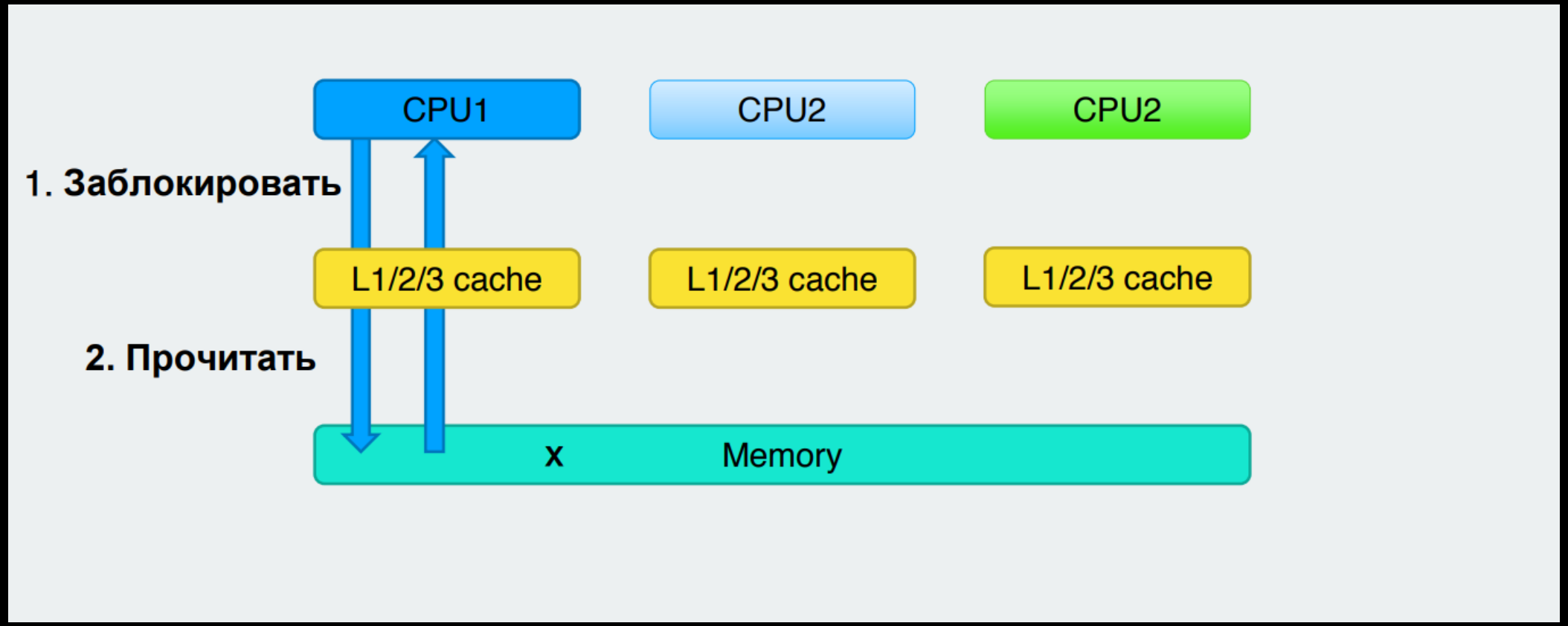
```cpp
#include <atomic>
#include <cassert>
#include <iostream>
#include <thread>

int data = 0;
std::atomic<bool> ready(false);

void producer() {
        data = 100;
        ready.store(true, std::memory_order_release); // Set flag
}
void consumer() {
        while (!ready.load(std::memory_order_acquire)) ;
        assert(data == 100);
}

int main() {
        std::thread t1(producer);
        std::thread t2(consumer);
        t1.join();
        t2.join();
        return 0;
}
```

Разница между последовательно последовательной моделью и моделью освобождения-захвата заключается в том, что первая обеспечивает глобальный порядок операций во всех потоках, а вторая - порядок только между парами операций освобождения и захвата.

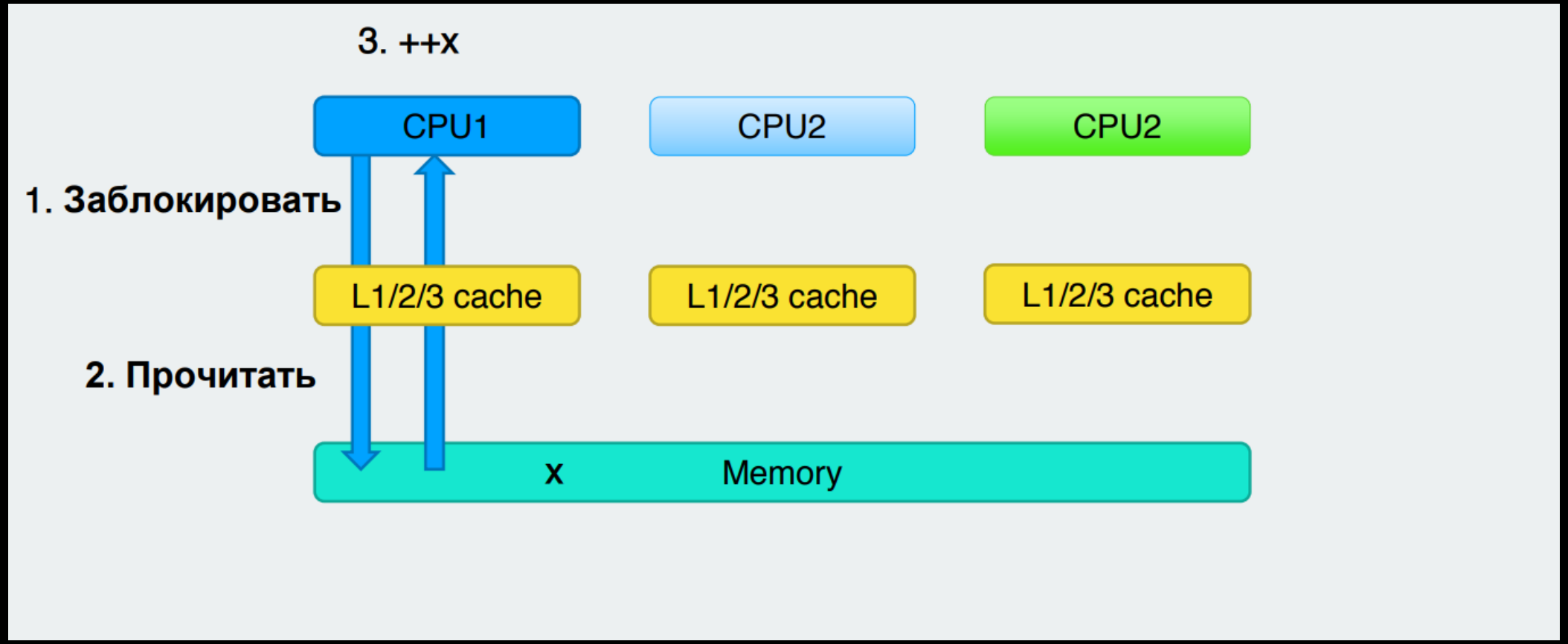
атомарные операции



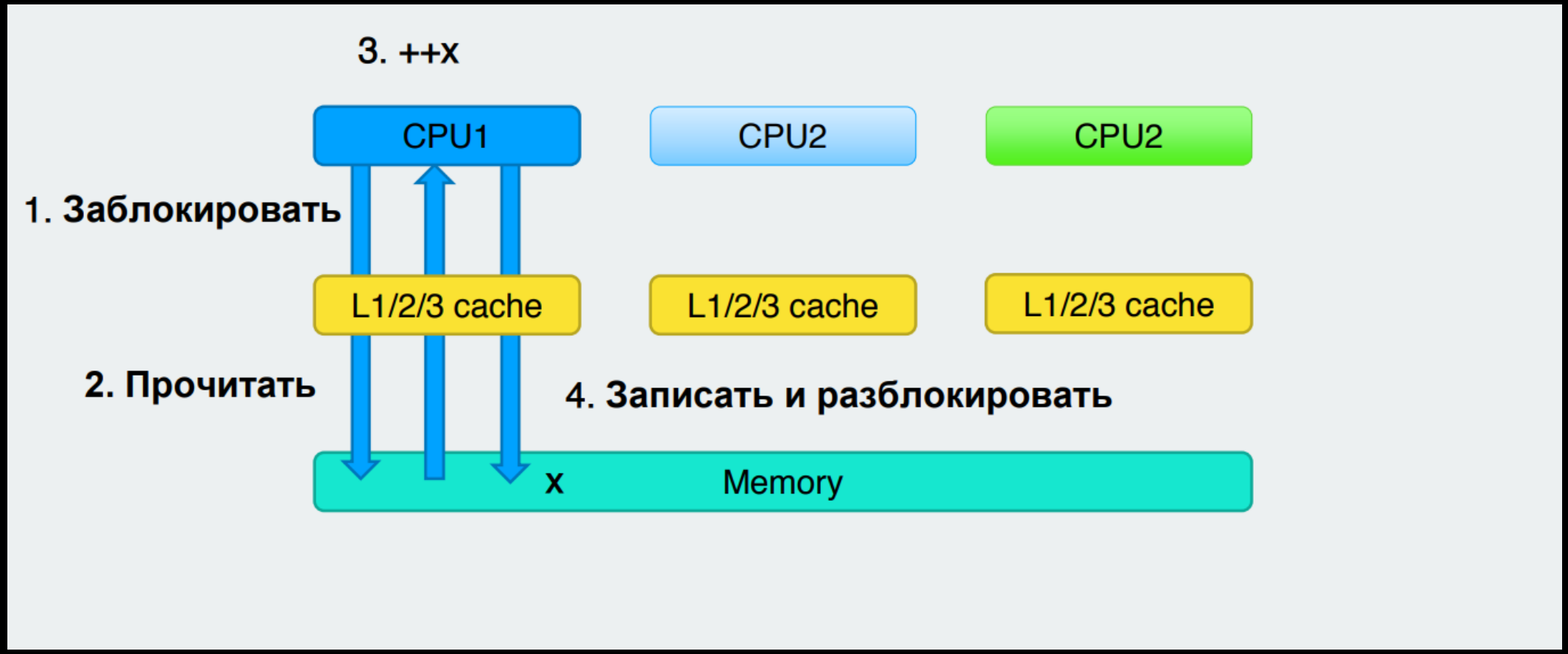
атомарные операции



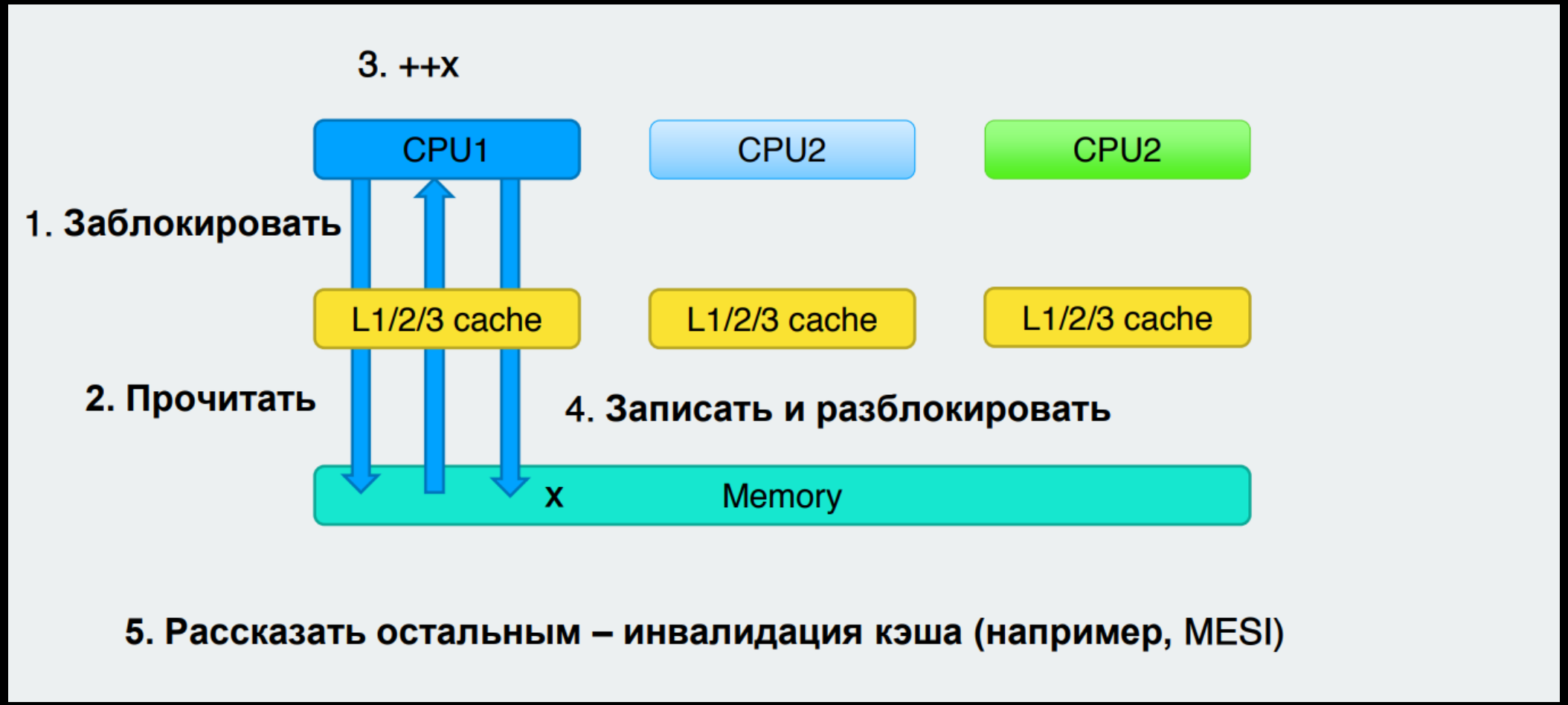
атомарные операции



атомарные операции



атомарные операции



Захват-освобождение (acquire/release)

- захват – это загрузка значения из атомарной переменной
- освобождение – запись в атомарную переменную
- освобождение синхронизируется-с захватом над одной переменной
- синхронизация между потоками

Как будем делать exchange?

Weak or strong?

Weak or Strong?

- **compare_exchange_weak** использ ьзуется, когда вам нужно повторять операцию в цикле (например, в алгоритмах, где требуется несколько попыток), и вы готовы к возможности ложных фейлов.
- **compare_exchange_strong** испо льзуется, когда вам нужна гарантия, что обмен произойдет только при совпадении значений, и вы не хотите иметь дело с ложными фейлами.

compare_exchange_weak может возвращать false (то есть не выполнять обмен), даже если ожидаемое значение совпадает с текущим. Это связано с тем, что на некоторых архитектурах (например, на процессорах с архитектурой x86) реализация compare_exchange_weak может использовать инструкцию cmpxchg, которая может "фейлиться" из-за особенностей работы процессора.

compare_exchange_strong всегда выполняет обмен, если текущее значение совпадает с ожидаемым. Он не "фейлится" ложно, то есть если значение совпадает, обмен будет выполнен

проблема ABA

В многозадачных вычислениях проблема ABA возникает при синхронизации, когда ячейка памяти читается дважды, оба раза прочитано одинаковое значение, и признак «значение одинаковое» трактуется как «ничего не менялось».

Однако, другой поток может выполниться между этими двумя чтениями, поменять значение, сделать что-нибудь ещё и восстановить старое значение. Таким образом, первый поток обманется, считая, что не поменялось ничего, хотя второй поток уже разрушил это предположение.

Решение ABA проблемы **tagged pointers**

```
template <typename T>
struct tagged_ptr {
T * ptr ;
unsigned int tag ;

tagged_ptr(): ptr(nullptr), tag(0) {}
tagged_ptr( T * p ): ptr(p), tag(0) {}
tagged_ptr( T * p, unsigned int n ): ptr(p), tag(n) {}

T * operator->() const { return ptr; }
};
```

- Вводится tag – версия объекта. При любой операции с объектом (чтение/запись) tag увеличивается.
- В CAS операциях мы сравниваем не значение, а версию указателя.

- Проблема – нам нужно атомарно менять два числа!

Безопасное удаление

```
std::shared_ptr<T> pop()
{
node* old_head=head.load(); // читаем old_head
while (old_head
                && !head.compare_exchange_weak(
                old_head,
                old_head->next)); // old head может быть удален с момента получения

return old_head ?  old_head->data : std::shared_ptr<T>();
}
```

Необходим механизм безопасного удаления

Hazard Pointers

https://www.academia.edu/23811827/Lock-Free_Data_Structures_with_Hazard_Pointers

- Прежде чем начать работать с элементом lock-free контейнера мы его помечаем как Hazard-pointer, добавляя в специальный список. У каждого потока свой массив hazard-указателей. Читать их могут все.
- При удалении, указатели не сразу удаляются, а помещаются в специальный список. Периодически из него удаляются не hazard-указатели.

braces

braces.cpp

Выражение

std::cout << *(std::cout << "1+2=",2)* << std::endl;

Выполнит левую часть кода (от запятой) , но как результат вернет правую часть

initializer_list

initializer_list.cpp

- #include <initializer_list>

- template<class T>
- void printme(std::initializer_list<T> t) {
- for (auto i : t) std::cout << i << std::endl;;
- }
- printme({10,2,25,36});

Соединяем вместе
cartesian_product.cpp

```
constexpr auto cartesian =
 [=](auto... xs) constexpr
  {
    return [=](auto f, auto cond) constexpr
       {
        (void)std::initializer_list<int>{
          ((void)call_cart(f, cond, xs, xs...), 0)...};
       };
  };
```

structured binding &
std::tie
structured_binding.cpp

- struct MyStruct {
- int i = 0;
- std::string s;
- };
- MyStruct ms;
- auto [u,v] = ms;

На этом все!

Пусть сила ООП будет с вами в Новом году!