ВВЕДЕНИЕ В МУЛЬТИПРОГРАММИРОВАНИЕ

ЛЕКЦИЯ 15

Неблокирующие алгоритмы

1. Без препятствий (**Obstruction-Free**) — поток совершает прогресс, если не встречает препятствий со стороны других потоков
2. Без блокировок (**Lock-Free**) — гарантируется системный прогресс хотя бы одного потока
3. Без ожидания (**Wait-Free**) — каждая операция выполняется за фиксированное число шагов, не зависящее от других потоков

CPP Core Guideline

CP.100: Don’t use lock-free programming unless you absolutely have to

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Атомарные переменные

Атомарность

Атомарная операция – операция, которая либо выполняется целиком, либо не выполняется вовсе.

Операция, которая не может быть частично выполнена и частично не выполнена.

Сторонний наблюдатель может увидеть состояние системы до выполнения атомарной операции или после, но не во время выполнения операции.

Атомарные операции

Атомарность означает неделимость операции. Это значит, что ни один поток не может увидеть промежуточное состояние операции, она либо выполняется, либо нет.

Например операция «++» не является атомарной:

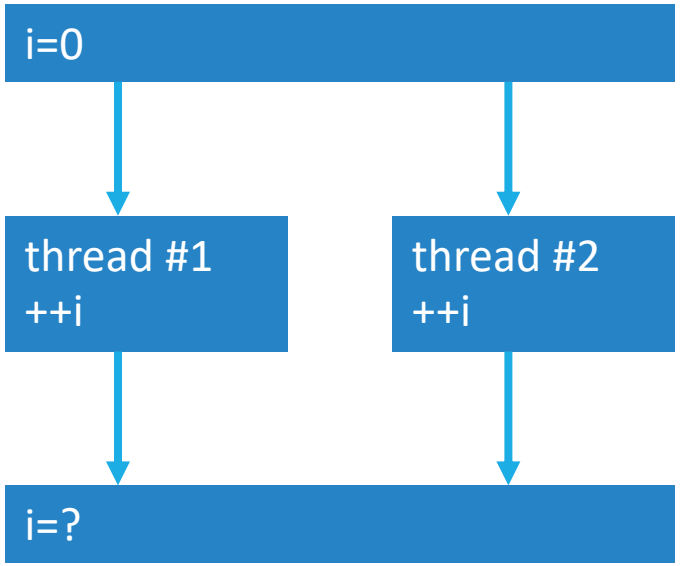
```
int x = 0;

++x;
```

Транслируется в ассемблерный код, примерно так:

```
013C5595  mov         eax,dword ptr [x]
013C5598  add         eax,1
013C559B  mov         dword ptr [x],eax
```

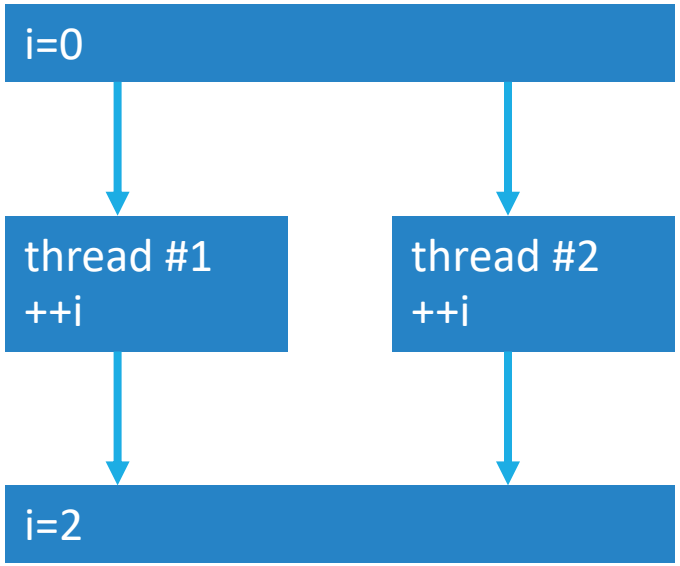
Без atomic vs atomic

без atomic



undefined behavior

atomic



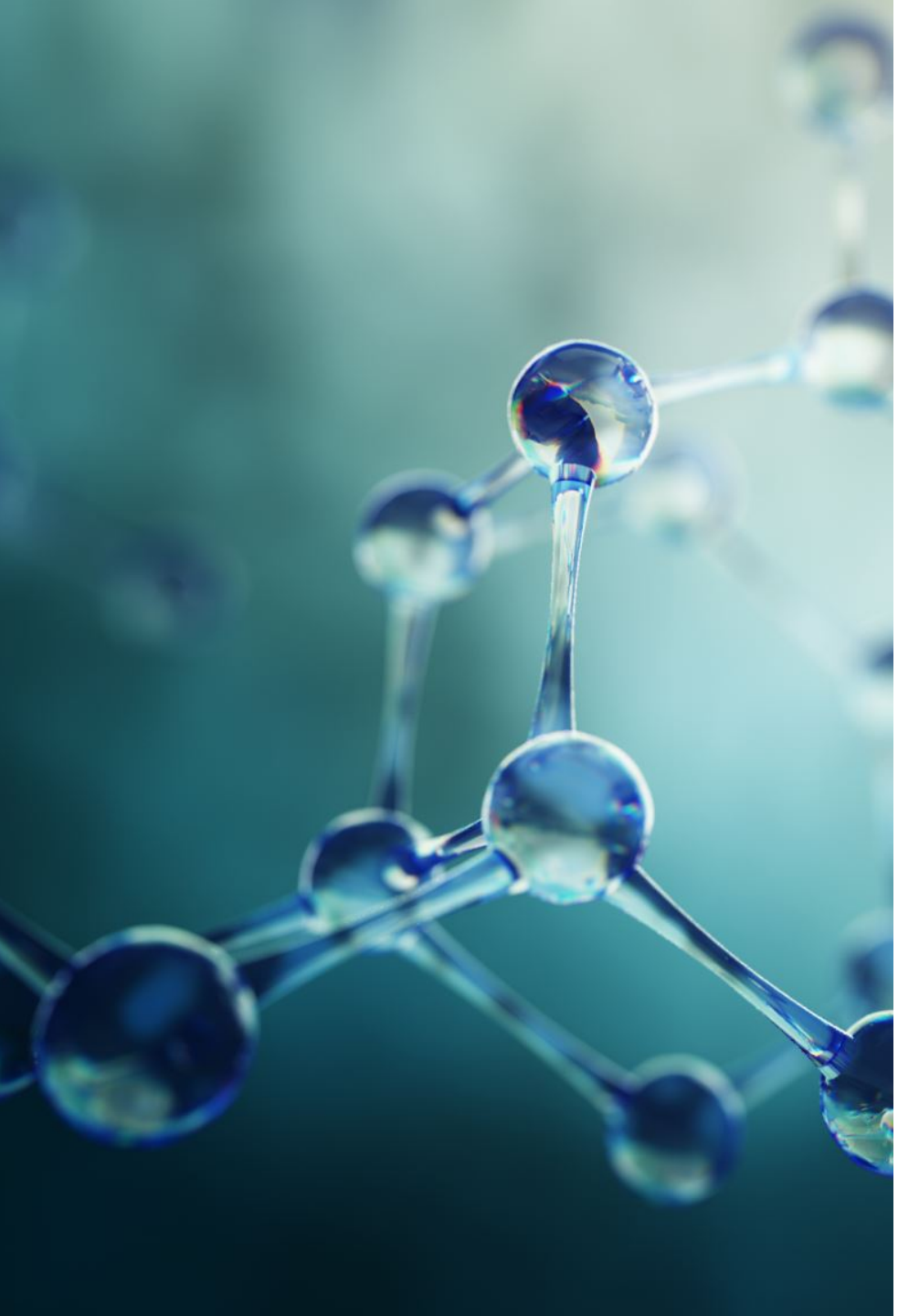
Атомарные типы #include <atomic>

std::atomic<T>

- операции гарантированно атомарные
- хоть и всё ещё происходит чтение-модификация-запись
- а реализовано это через блокировки и взаимодействии процессоров.

Атомарность

1. Есть разделяемые переменные;
2. Доступ к разделяемым переменным осуществляется без использования механизмов блокировок;
   - Переменные можно изменять/читать без появления «состояния гонки»;
   - Промежуточные состояния изменения переменных «не наблюдаемы»;
3. Используется доступ к аппаратным атомарным инструкциям (fetch-and-add, xchg, cmpxchg);



Атомарные типы C++ #include<atomic>

```
std::atomic_bool //bool
std::atomic_char //char
std::atomic_schar //signed char
std::atomic_uchar //unsigned char
std::atomic_int //int
std::atomic_uint //unsigned int
std::atomic_short //short
std::atomic_ushort //unsigned short
std::atomic_long //long
std::atomic_ulong //unsigned long
std::atomic_llong //long long
std::atomic_ullong //unsigned long long
std::atomic_char16_t //char16_t
std::atomic_char32_t //char32_t
std::atomic_wchar_t //wchar_t
std::atomic_address //void*
```

Операции

- bool is_lock_free() // свободен ли тип от блокировок
- void store(T val, std::memory_order) //положить значение
- T load(std::memory_order) // достать значение
- operator T // автоматическое преобразование типа к T
- exchange(T val) // обменять (аналог swap)
- compare_exchange_strong/weak // чуть позже

Compare and Swap

Сравнить и обменять
краеугольный камень Lock free подхода

- атомарная операция;
- заменить значение если совпадает с ожидаемым;
- повторять в цикле пока значение не начнет совпадать и не удастся сделать замену;

CAS-операции

1. CAS — compare-and-set, compare-and-swap

```
bool compare_and_set(
        int* адрес_переменной ,
        int старое значение ,
        int новое значение)
```

2. Возвращает признак успешности операции установки значения
3. Атомарна на уровне процессора

1. Является аппаратным примитивом
2. Возможность продолжения захвата примитива без обязательного перехода в режим «ожидания»
3. Меньше вероятность возникновения блокировки из-за более мелкой операции
4. Быстрая

Если значение переменной такое, как мы ожидаем – то меняем его на новое;

Основные операции

```
load() //Прочитать текущее значение

store() //Установить новое значение

exchange() //Установить новое значение и вернуть предыдущее

compare_exchange_weak() // см. следующий слайд

compare_exchange_strong() // compare_exchange_weak в цикле

fetch_add() //Аналог оператора ++

fetch_or() //Аналог оператора --

is_lock_free() //Возвращает true, если операции на данном типе
неблокирующие
```

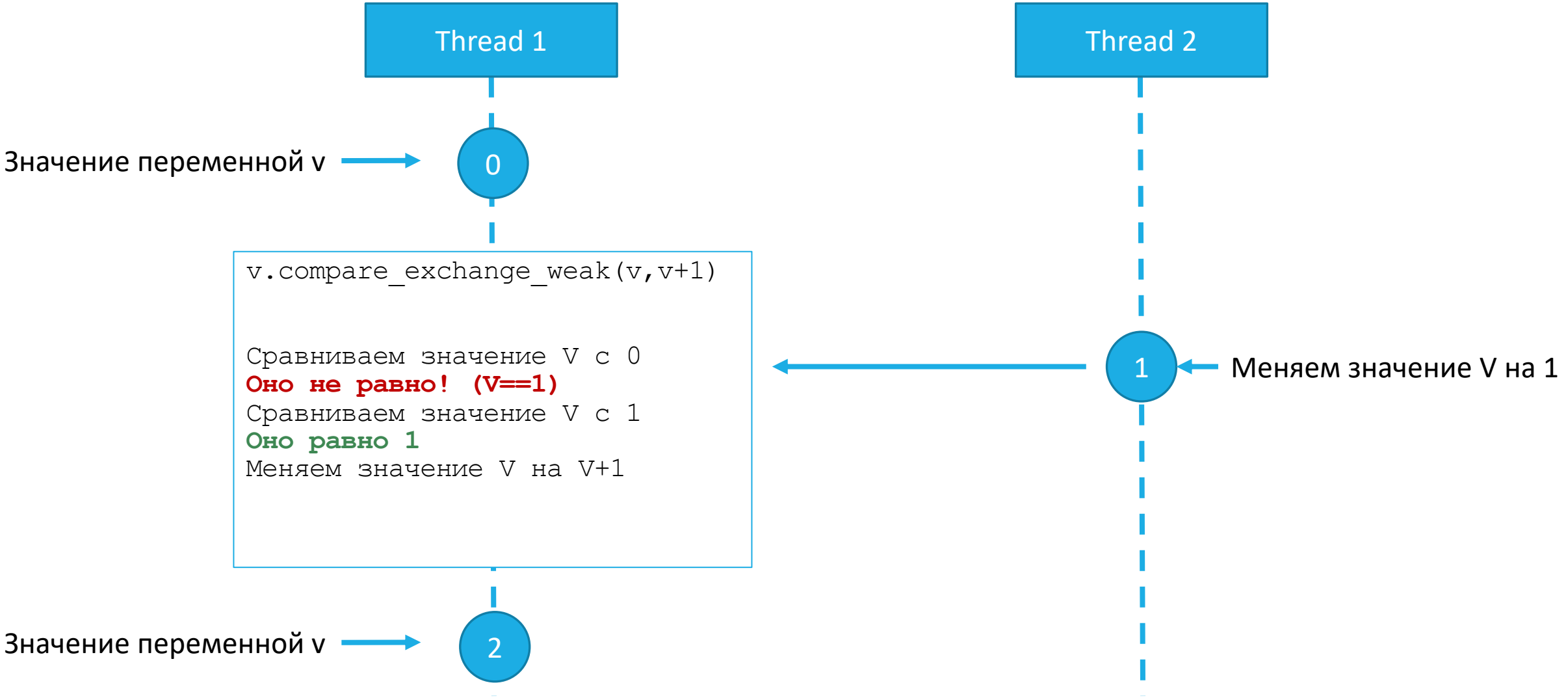
Метод atomic::compare_exchange_weak

```
bool compare_exchange_weak( Ty& OldValue, Ty NewValue)

/**

Сравнивает значения которые хранится в *this с OldValue.

  • Если значения равны то операция заменяет значение, которая
    хранится в *this на NewValue (*this= NewValue) , с помощью
    операции read-modify-write.

  • Если значения не равны, то операция использует значение,
    которая хранится в *this, чтобы заменить OldValue (OldValue
    =*this).

 */
```

CAS Loop
типичный паттерн применения

1. Прочитать значение A из переменной V;
2. Взять какое-то новое значение B для V;
3. Использовать CAS для атомарного изменения V из A в B до тех пор, пока другие потоки меняют значение V во время этого процесса;

compare_exchange_weak простой инкремент



Атомарные переменные
cas_1.cpp

```
void add_func(bool atomic) {
{
 std::unique_lock<std::mutex> lock(mtx);
 cv.wait(lock); // ждем начала работы
}
if (atomic) { // делаем 1000 атомарных добавлений
 for (int i = 0; i < 1000; i++)
  cnt.value_a.fetch_add(1);
 } else { // делаем 1000 не-атомарных добавлений
 for (int i = 0; i < 1000; i++)
  ++ cnt.value;
 }

}
```

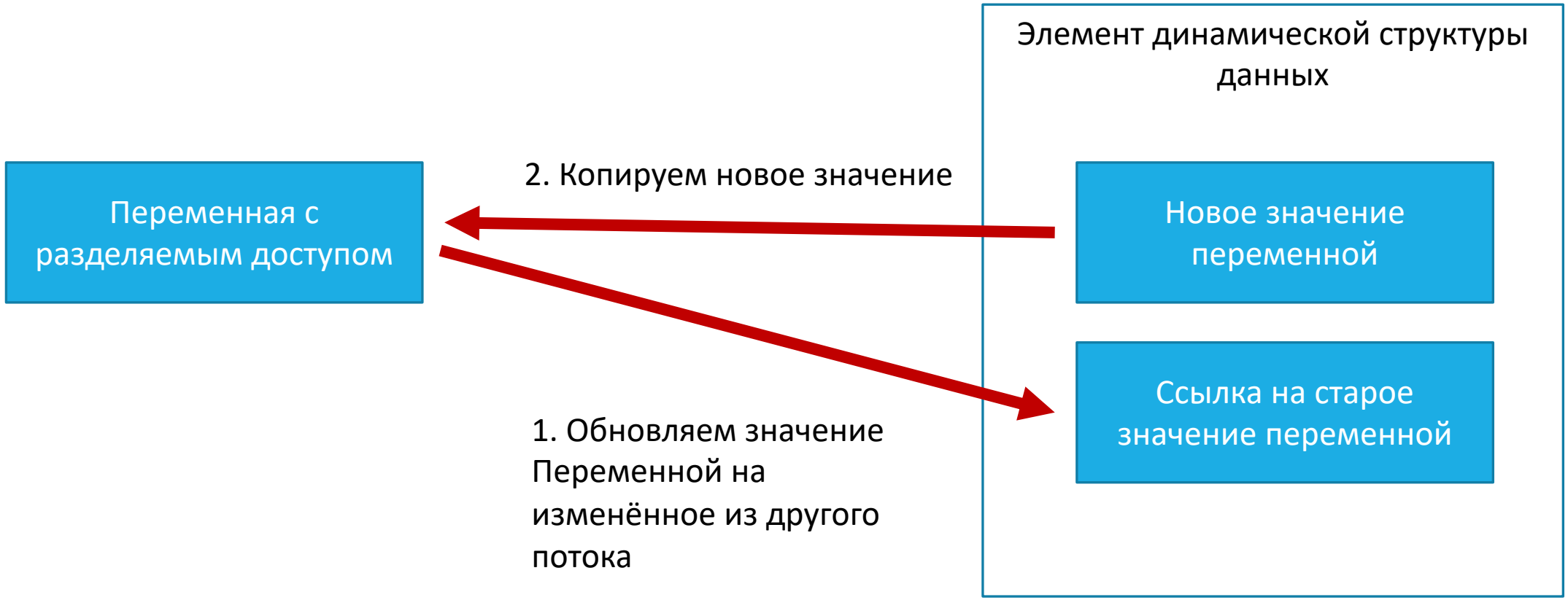
Lock Free но не Wait Free
lock_free.cpp

```
static const size_t sleep_micro_sec = 5;
std::atomic<bool> locked_flag{false};

void foo() {
 bool exp = false;
 while (!locked_flag.compare_exchange_strong(exp, true)) {
 exp = false;
 if (sleep_micro_sec == 0) {
  std::this_thread::yield();
  } else if (sleep_micro_sec != 0) {
   std::this_thread::sleep_for(std::chrono::microseconds(sleep_micro_sec));
  }
 }
}
```

spin_lock.cpp

```
class spin_lock
{
atomic<unsigned int> m_spin ;
public:
spin_lock(): m_spin(0) {}
~spin_lock() { assert( m_spin.load(memory_order_relaxed) == 0);}

void lock()
{
unsigned int nCur;
do { nCur = 0; } // устанавливаем "старое значение" в 0
while ( !m_spin.compare_exchange_weak( nCur, 1, memory_order_acquire ));
// если значение m_spin == 0 то меняем m_spin на 1 и выходим из цикла
// если значение m_spin == 1 то меняем nCur на 1 и идем на еще один цикл
}
void unlock()
{
m_spin.store( 0, memory_order_release );
}
};
```

CAS


Элемент динамической структуры данных
Переменная с разделяемым доступом
2. Копируем новое значение
Новое значение переменной
Ссылка на старое значение переменной
1. Обновляем значение Переменной на изменённое из другого потока

Потокобезопасный Stack

cas_2.cpp

```
void push(const T& data)
{
 // новый узел ссылается на конец списка
 node* new_node = new node(data, head.load());
 // если конец списка (head) равен next у нового конца списка,
 // тогда устанавливаем новый конец списка
 // иначе меняем new_node->next на head
 while (!head.compare_exchange_weak(
                          new_node->next,
                          new_node));
}
```

Недостатки CAS

1. CAS заставляет потоки, которые его вызывают, работать в условиях соревнования (contention)
2. Больше contention = больше бесполезных циклов процессора, трата процессорного времени
3. Написание корректных и быстрых алгоритмов на CAS требует специальной подготовки

Практический пример: **Singleton**

Нужно создать ровно один экземпляр объекта, который доступен из разных потоков.

Это может пригодится:

- Класс-менеджер для доступа к базе-данных. Так как нам нужно контролировать число подсоединений к базе;
- Класс-менеджер для работы с сетью. Так как нам надо контролировать число открытых портов;
- Класс-менеджер для работы с конфигурацией приложения. Что бы прочесть конфигурацию ровно один раз;

singleton_1.cpp // так делать нельзя

```
class Singleton{
private:
inline static Singleton *instance;
inline static std::mutex mtx;
Singleton(){
 sync_stream{} << "created\n";
};

public:
static Singleton *get_instance(){
 if (instance == nullptr){
  std::lock_guard<std::mutex> lock(mtx);
  if (instance == nullptr)
     instance = new Singleton(); // reordering or visibility of instance issue
  }
 return instance;
 }
};
```

singleton_2.cpp // а так уже льзя

```
class Singleton
{
private:
 Singleton(){
  sync_stream{} << "created\n";
 };

public:
 static Singleton *get_instance(){
  static Singleton instance;
  return &instance;
 }
};
```

singleton_3.cpp // spin-lock

```
lass Singleton
{
private:
 inline static std::atomic<Singleton*> instance;
 inline static std::atomic<bool> create {false};
 Singleton(){
  sync_stream{} << "created\n";
 };

public:
static Singleton *get_instance(){
  if (instance.load() == nullptr) {
  if (!create.exchange(true)) // сохраняем истину, ожидаем предыдущее значение 1
   instance.store(new Singleton()); // construct
   else while(instance.load() == nullptr) { } // spin
  }
  return instance.load();
 }
};
```

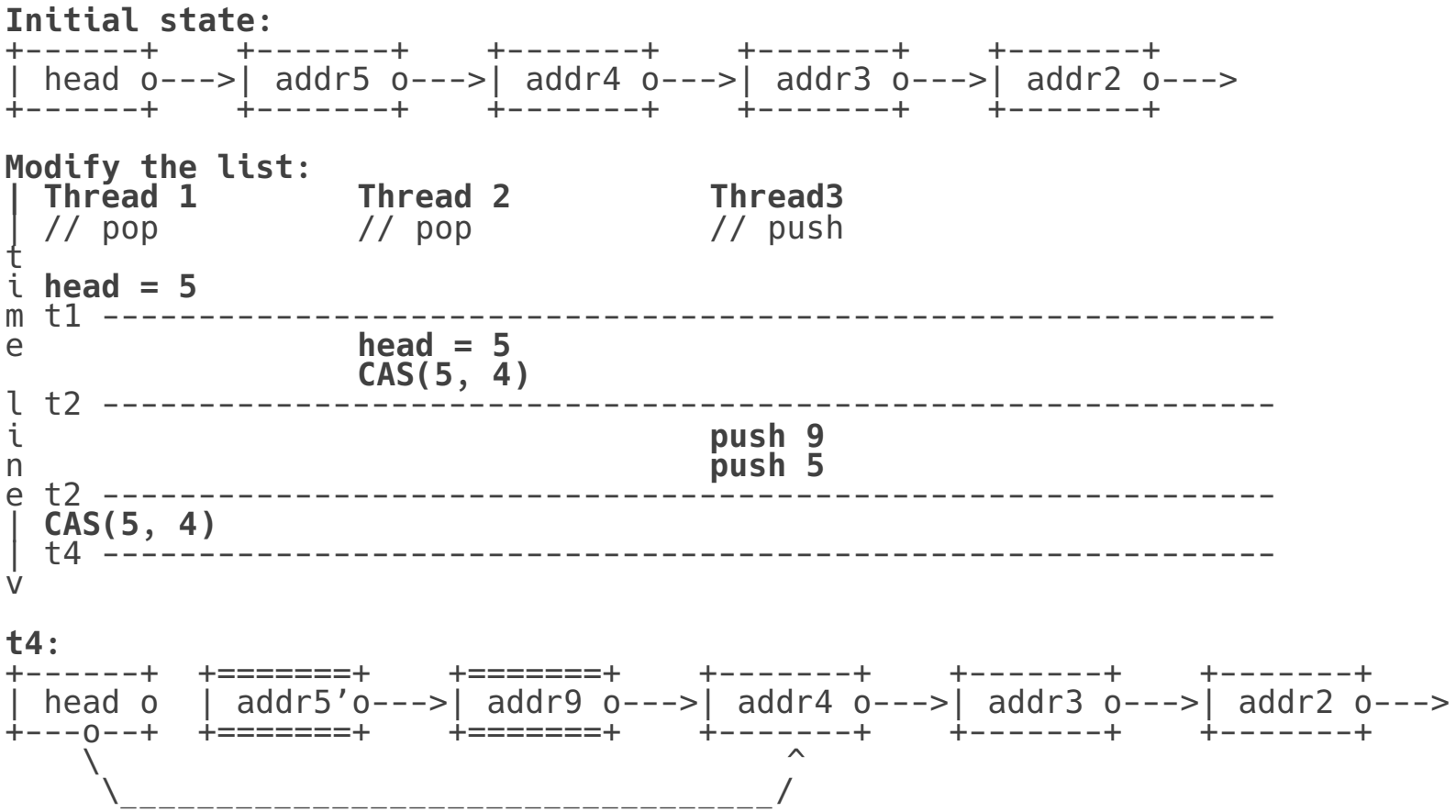
singleton_4.cpp // lazy evoluation

```
class Singleton
{
private:
inline static Singleton *instance{nullptr};
Singleton(){
 sync_stream{} << "created\n";
};

public:
 static Singleton *get_instance(){
  static std::once_flag create{};
  std::call_once(create, [] { instance = new Singleton(); });
  return instance;
 }
};
```

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.

ABA проблема в стэке



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>();
}
```

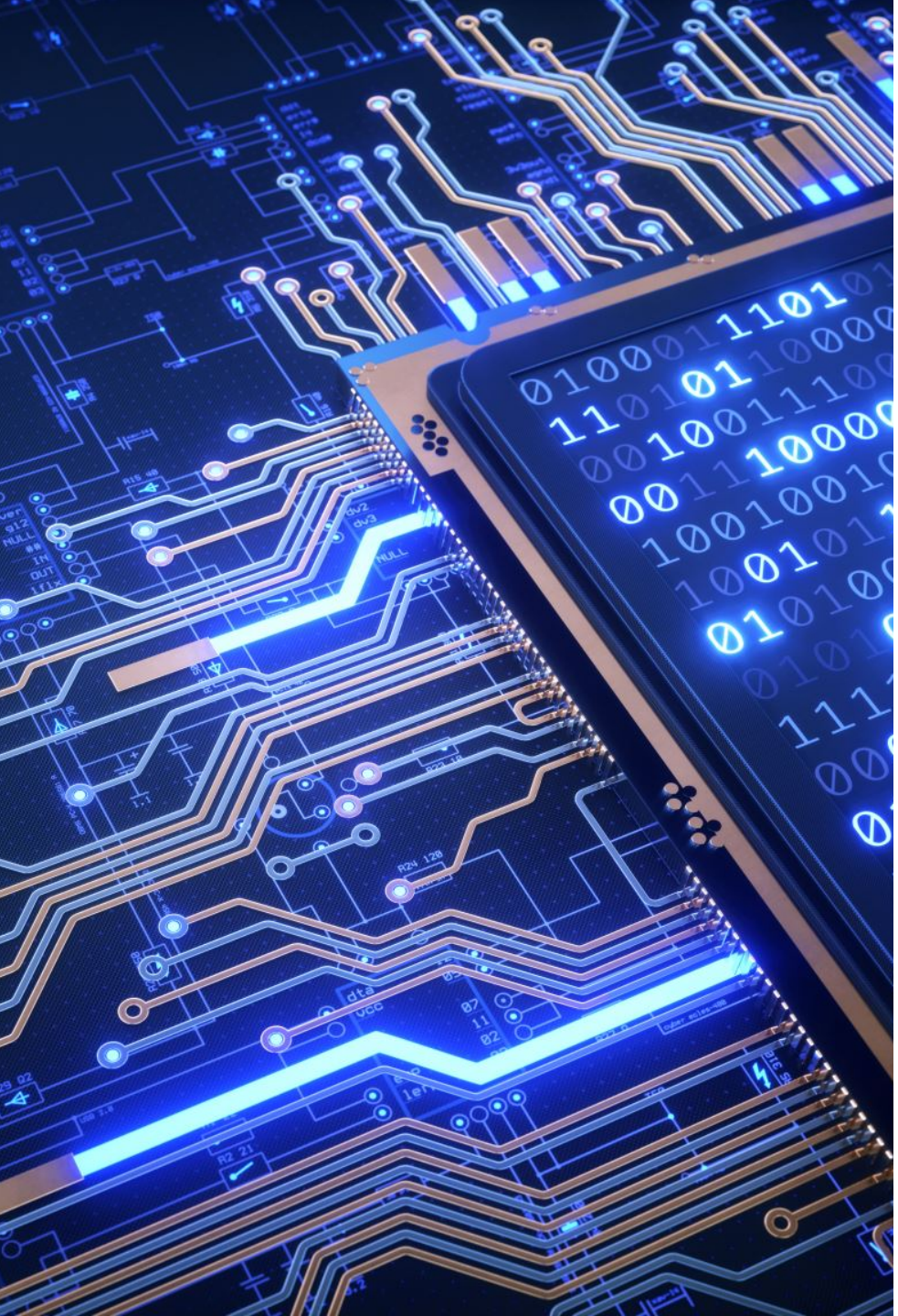
Необходим механизм безопасного удаления

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.

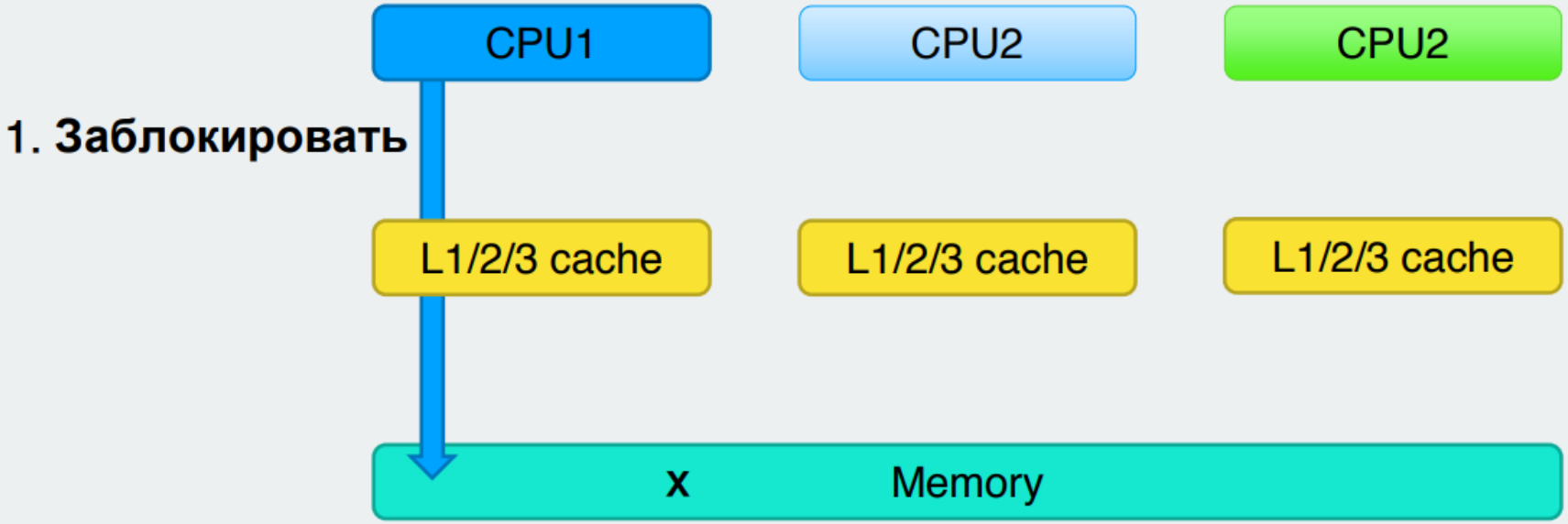
При удалении, указатели не сразу удаляются, а помещаются в специальный список. Периодически из него удаляются не hazard-указатели.



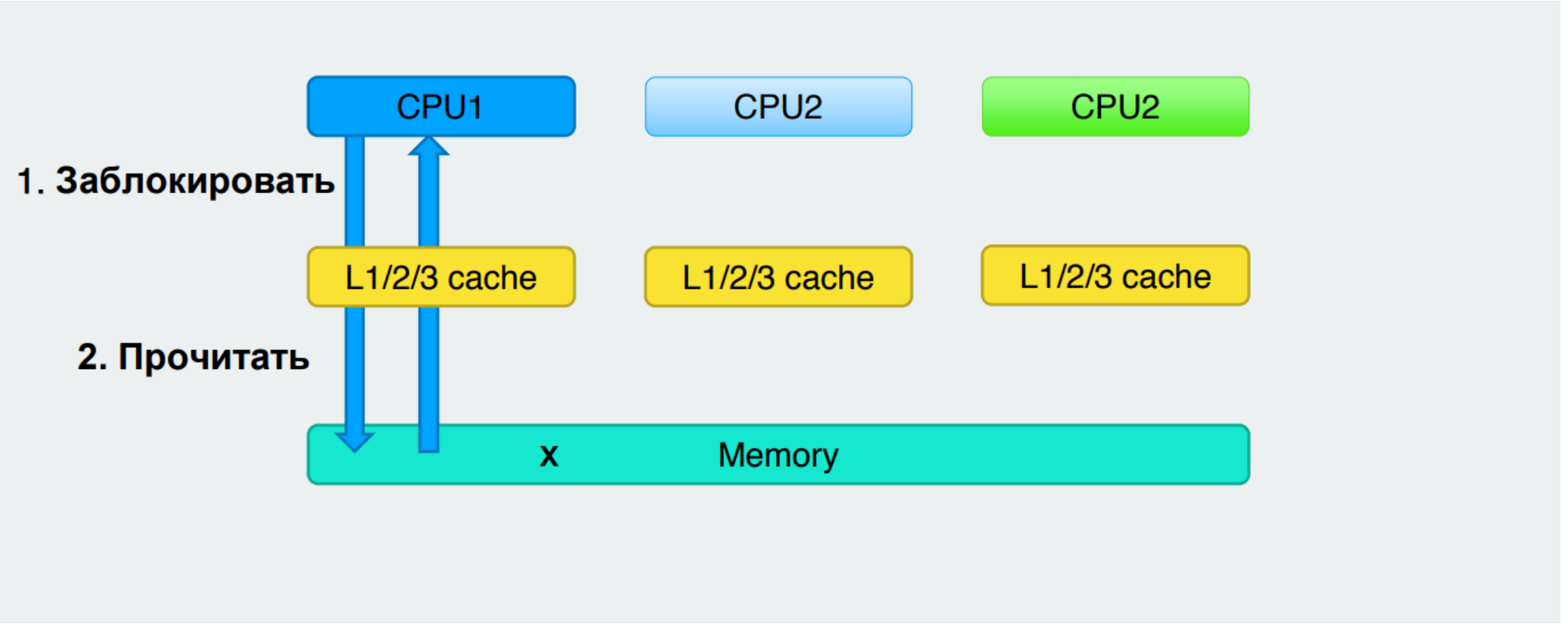
Модели памяти

1. **последовательное** (последовательно согласованное) –memory_order_seq_cst
2. **захват-освобождение** - memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel
3. **ослабленное** - memory_order_relaxed

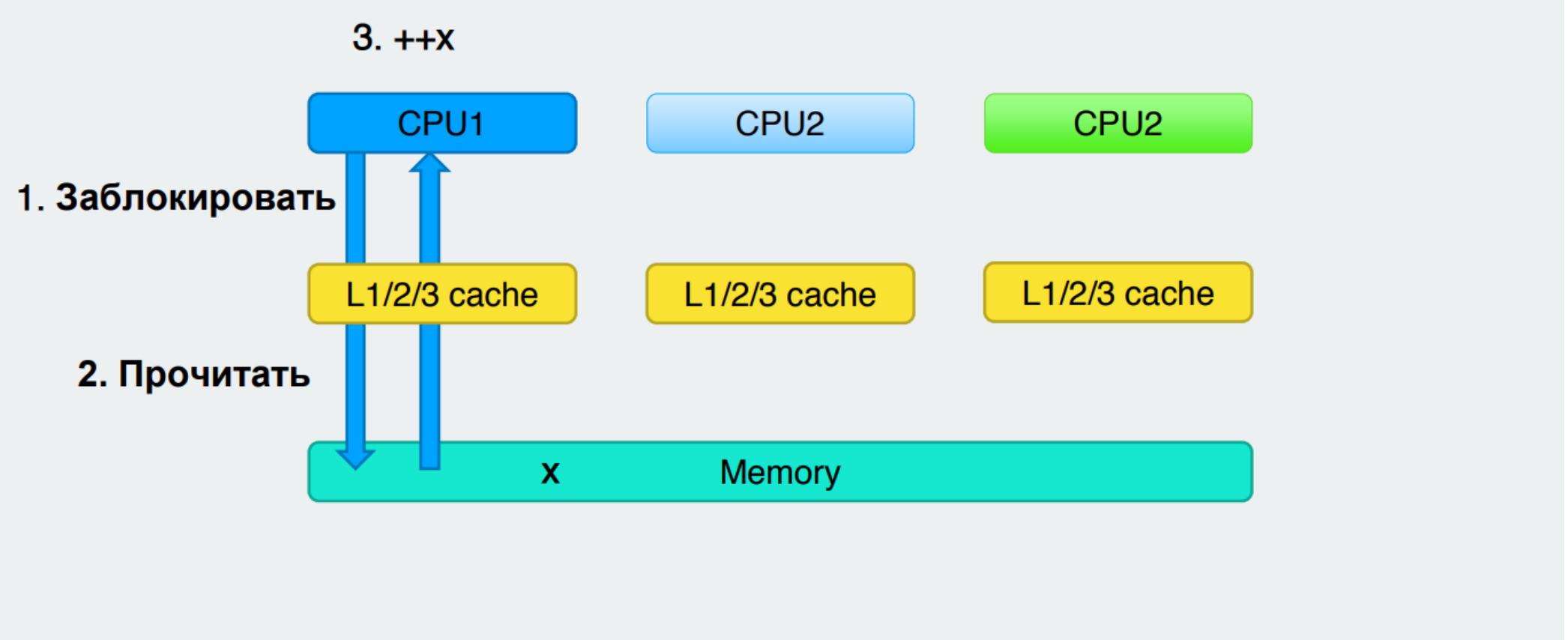
атомарные операции



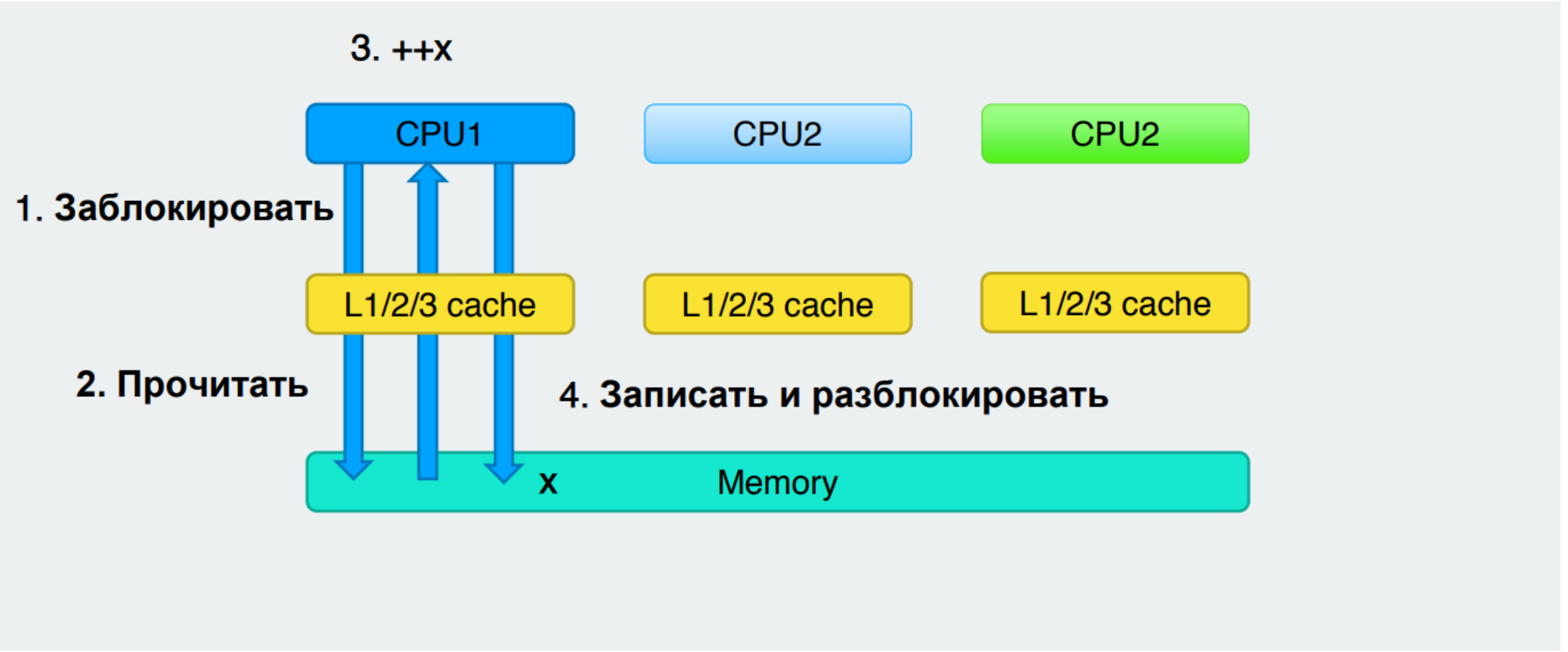
атомарные операции



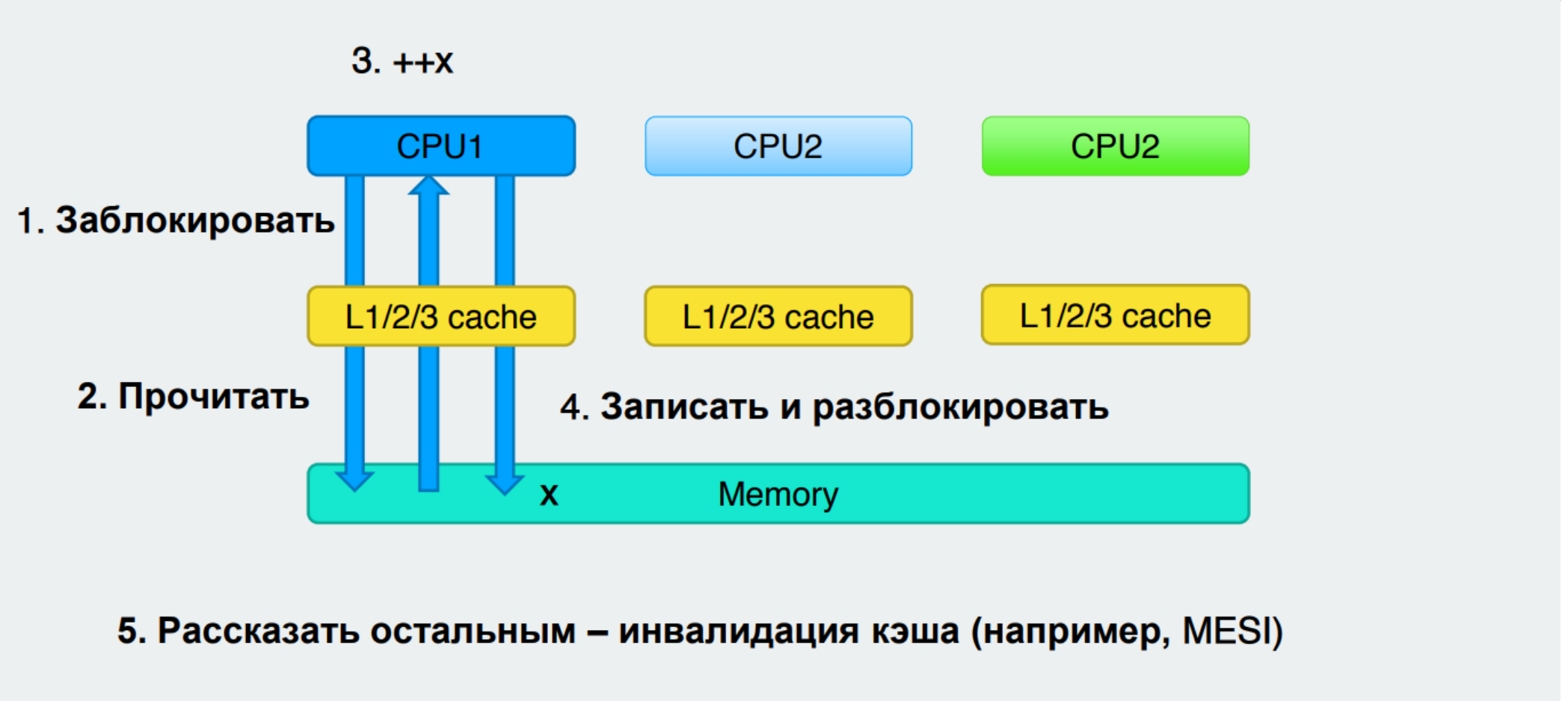
атомарные операции



атомарные операции



атомарные операции



Последовательное (memory_order_seq_cst):

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

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

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

Компилятор

- переупорядочение
- вырезание кода
- «оптимизация» проверок

Кэширование в современном CPU

- это минимум один,
- а в норме три уровня кэширования
- и каждый может дать неприятные постэффеты
- достаточно посмотреть MESI и MOESI

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ