ВВЕДЕНИЕ В МУЛЬТИПРОГРАММИРОВАНИЕ часть 2

ЛЕКЦИЯ №15-16

Что мы передаем в std::thread?

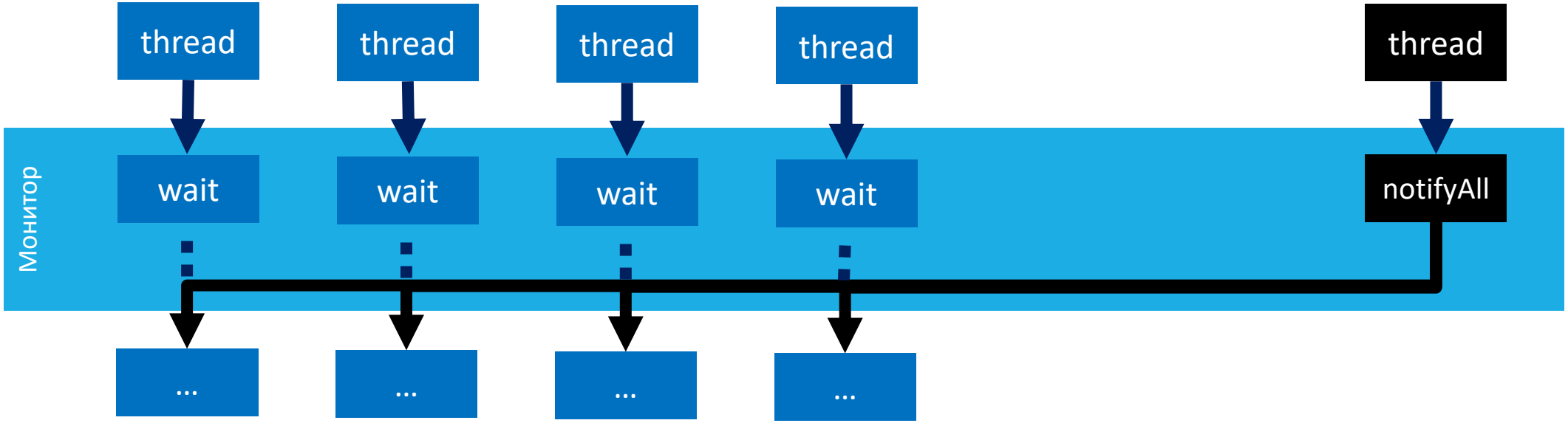
std::thread имеет varidaic конструктор, который **копирует** все параметры в вызываемую в потоке функцию;

```
void f(int x,MyObject& o);
int y;
MyObject m;
std::thread t(f,y,m); // запускает копию f передавая в него копии y и m
```

Внутри **f** мы работаем с копиями **y** и **m**.

Условная переменная

примитив синхронизации, обеспечивающий блокирование одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания. Условные переменные используются вместе с ассоциированным мьютексом и являются элементом некоторых видов мониторов.



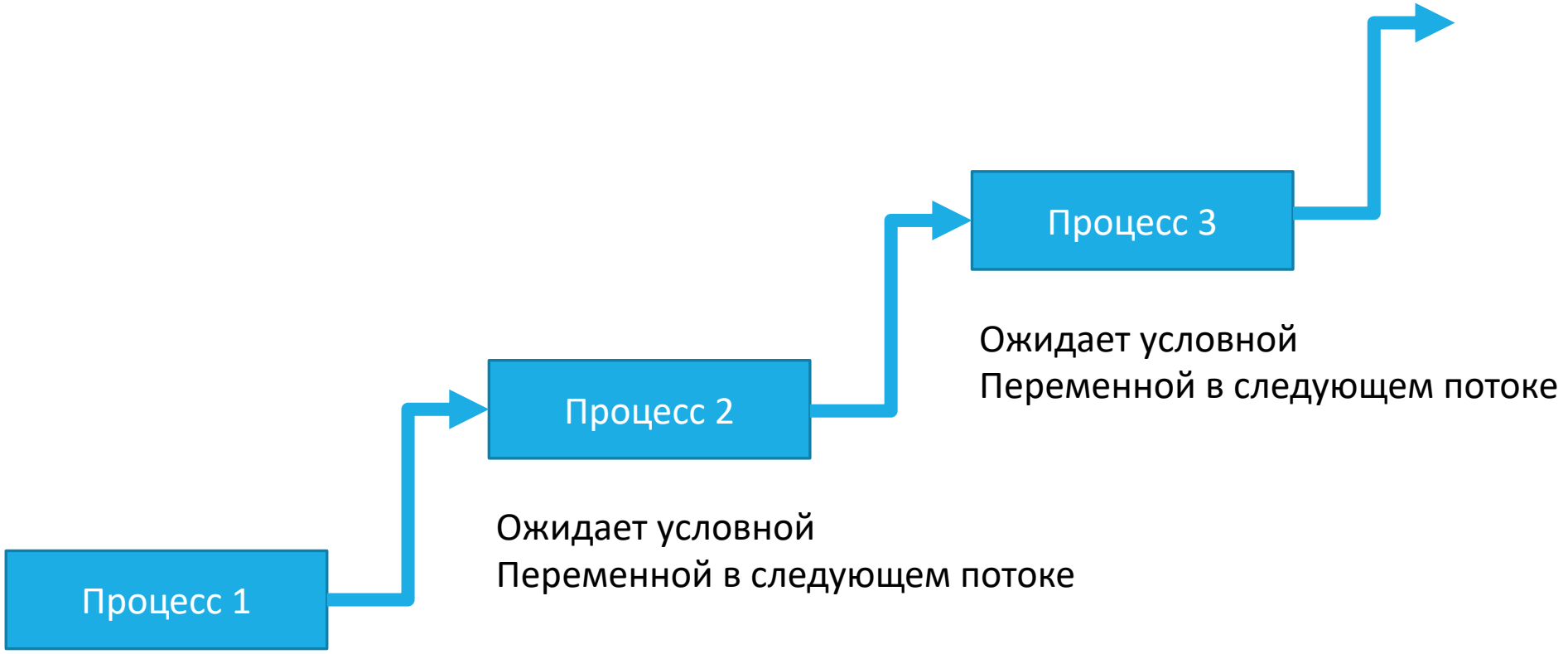
Условные переменные <condition_variable>
Example116_CondVariable1

condition_variable

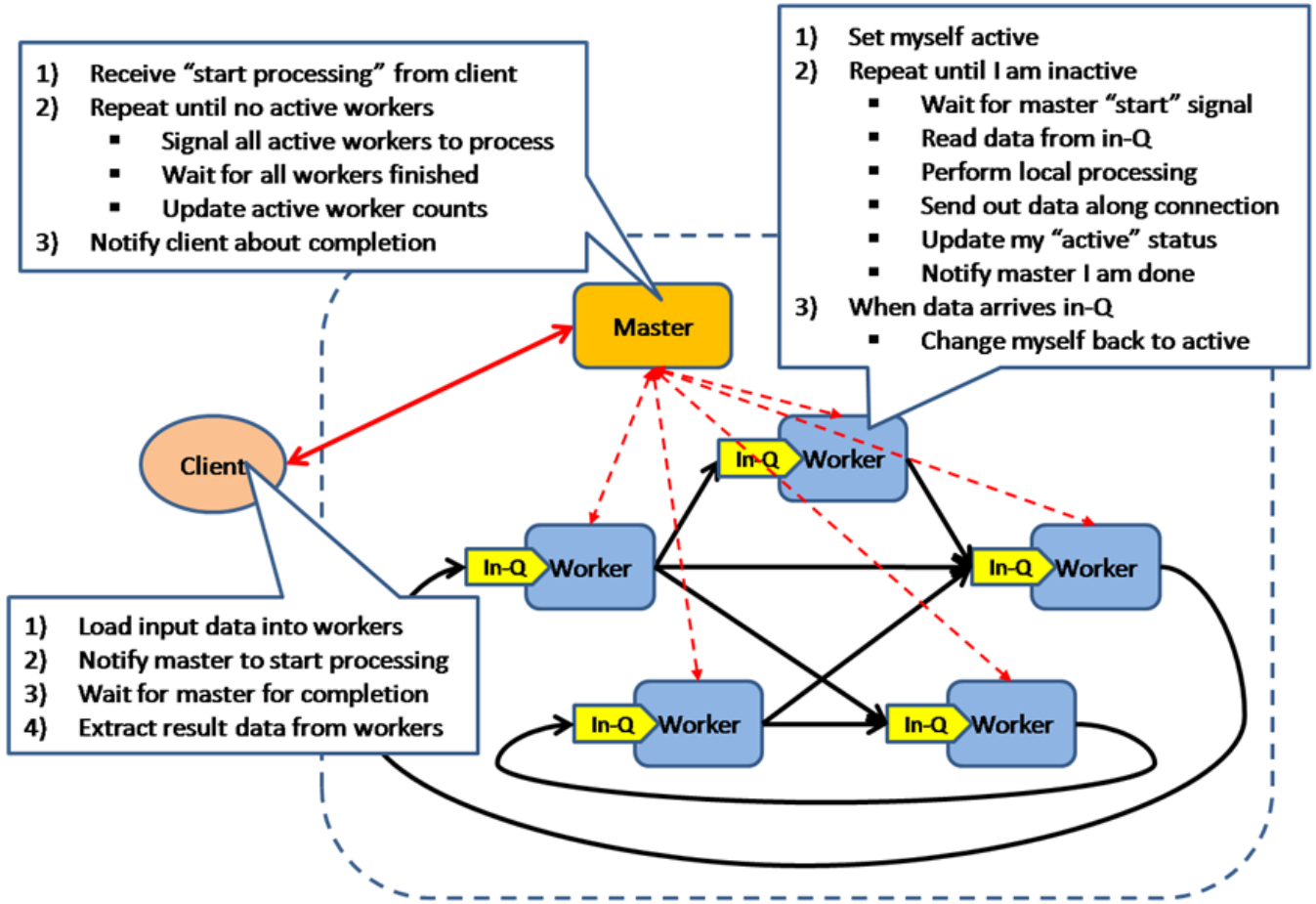
требует от любого потока перед ожиданием сначала выполнить std::unique_lock

1. Должен быть хотя бы один поток, ожидающий, пока какое-то условие станет истинным. Ожидающий поток должен сначала выполнить unique_lock.
2. Должен быть хотя бы один поток, сигнализирующий о том, что условие стало истинным. Сигнал может быть послан с помощью notify_one(), при этом будет разблокирован один (любой) поток из ожидающих, или notify_all(), что разблокирует все ожидающие потоки.
3. В виду некоторых сложностей при создании пробуждающего условия, которое может быть предсказуемых в многопроцессорных системах, могут происходить ложные пробуждения (spurious wakeup). Это означает, что поток может быть пробужден, даже если никто не сигнализировал условной переменной. Поэтому необходимо еще проверять, верно ли условие пробуждение уже после то, как поток был пробужден.

Запускаем цепочку заданий
Example117_CondVariable2



Пример организации вычислений Example103_Zookeeper



Проблема блокировок

1. Взаимоблокировки (Deadlocks)
2. Надежность — вдруг владелец блокировки помрет?
3. Performance
   - Параллелизма в критической секции нет!
   - Владелец блокировки может быть вытеснен планировщиком

Закон Амдала

Джин Амдал (Gene Amdahl) - один из разработчиков всемирно известной системы IBM 360 в 1967 году предложил формулу, отражающую зависимость ускорения вычислений, достигаемого на многопроцессорной ВС, как от числа процессоров, так и от соотношения между последовательной и распараллеливаемой частями программы. Проблема рассматривалась Амдалом исходя из положения, что объем решаемой задачи (рабочая нагрузка - число выполняемых операций) с изменением числа процессоров, участвующих в ее решении, остается неизменным.

Пусть
f - доля операций, которые должны выполняться последовательно одним из процессоров и
1-f - доля, приходящаяся на распараллеливаемую часть программы.
Тогда ускорение, которое может быть получено на ВС из *n* процессоров, по сравнению с однопроцессорным решением не будет превышать величины:

S(n)=T(1)/T(n)=1/[f+(1-f)/n].

Например, если половина операций подлежит распараллеливанию на 4 машинах, то ускорение равно:

S(4) = 1/(0.5+0.5/4)=1.6 т.е. Только в полтора раза!

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

Атомарные переменные

Atomic Design

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 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
```

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*
```

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. Более 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)
```

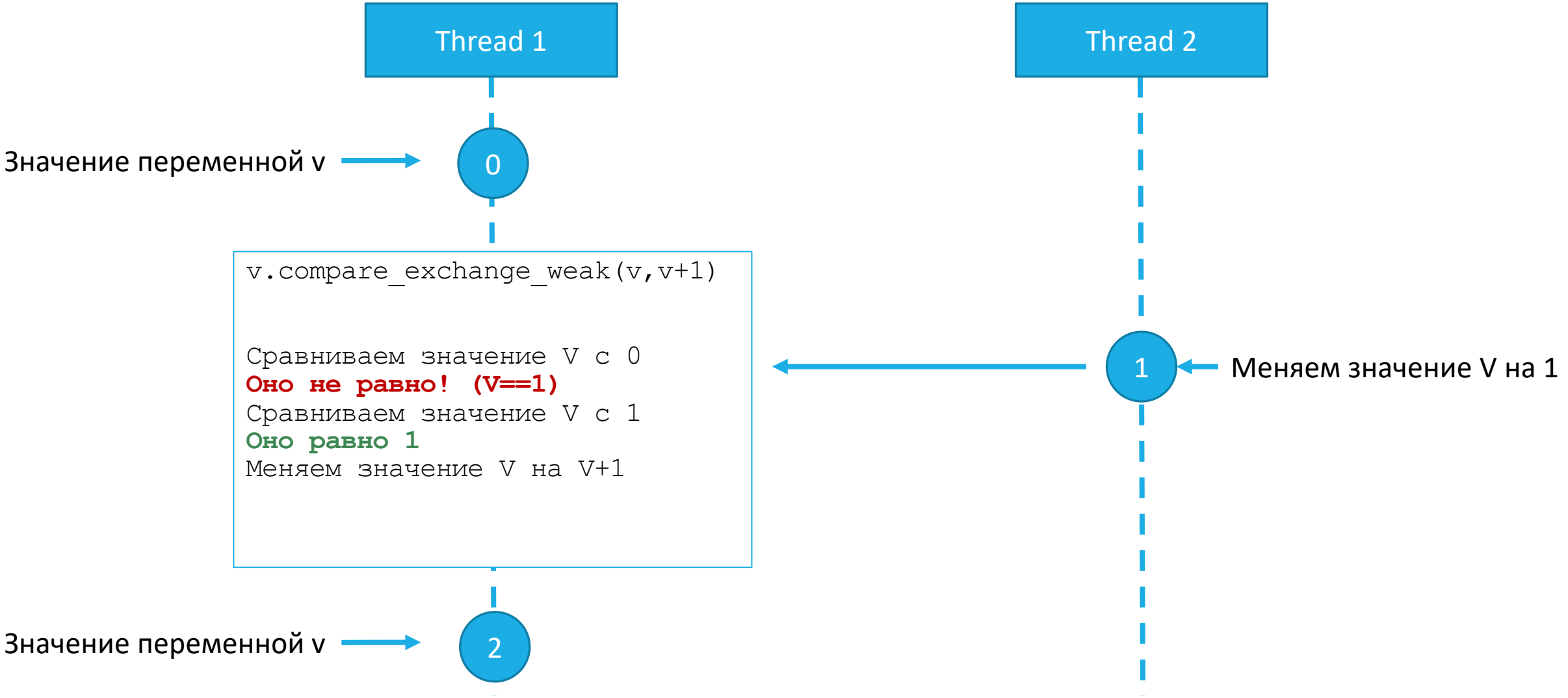
Сравнивает значения которые хранится в *this с `OldValue`.

- 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



Простой пример Example120_CASSimple

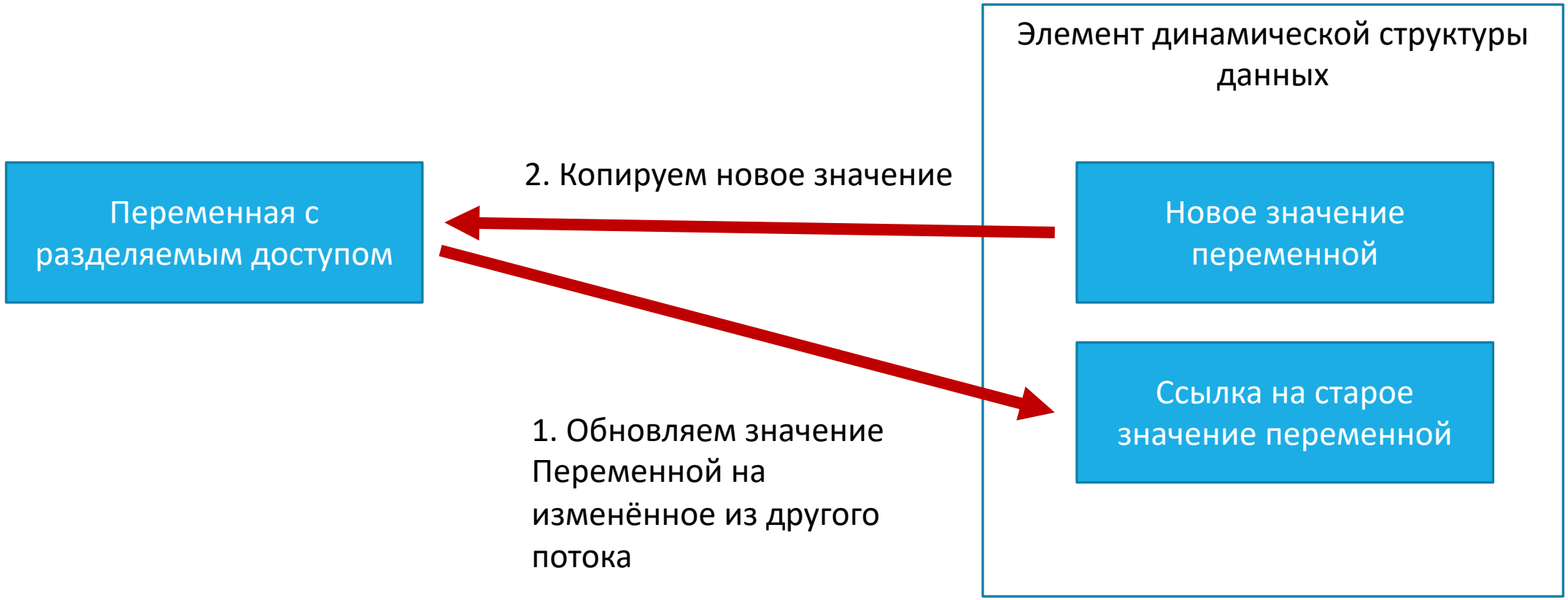
```
1.class Counter {
2.private:
3.    std::atomic_int value_a;
4.public:
5.    Counter() : value_a(0) {    }
6.    int get_a() {
7.        return value_a.load();
8.    }
9.    int increment_atomic() {
10.         int v = value_a.load();
11.         while (!value_a.compare_exchange_weak(v, v + 1));
12.         return value_a.load();
13.     }
14.};
```

Пример: spin_lock example119_spinlock

```
1.class spin_lock {

2.atomic<unsigned int> m_spin ;
3.public:
4. spin_lock(): m_spin(0) {}
5.~spin_lock() { assert( m_spin.load(memory_order_relaxed) == 0);}
6.
7. void lock()    {
8.         unsigned int nCur;
9.         do { nCur = 0; } // устанавливаем "старое значение" в 0
10.         while ( !m_spin.compare_exchange_weak( nCur, 1, memory_order_acquire ));
11.          // если значение m_spin == 0 то меняем m_spin на 1 и выходим из цикла
12.         // если значение m_spin == 1 то меняем nCur на 1 и идем на еще один цикл
13.    }
14.
15.void unlock()    {
16.         m_spin.store( 0, memory_order_release );
17.}
18.};
```

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

Example119_CAS

```
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

Модели памяти

1. **Sequential Consistency**
   - Сохраняется порядок всех операций
   - Включен по умолчанию
2. **Aquire/Release/Consume**
   - Сохраняется порядок пар чтения/запись
   - Независимые чтения и записи не требуют синхронизации CPU
3. **Relaxed Atomics**
   - Чтения и записи без какого-либо порядка (поддерживается не на всех аппаратных архитектурах)

Лямбда выражения

Подсчет вхождений числа 42 в векторе

```
// C++03

struct functor {

int &a;
functor(int& _a) : a(_a) { } bool operator()(int x)
const {

return a == x; }

};

int a = 42;
count if(v.begin(),v.end(),functor(a));
```

```
// C++11

int a = 42; count_if(v.begin(), v.end(),

[&a](int x){ return x == a;});
```

Lambda
Example121_Lambda

Лямбда-выражения в C++ — это краткая форма записи анонимных функторов.

Например:

```
[](int _n)   {    cout << _n << " ";}
```

Соответствует:

```
class MyLambda
{
   public: void operator ()(int _x) const { cout << _x << " "; }
};
```

Lambda == Functor

```
[ captures ]   ( params ) -> ret   { statements; }
```

```
class functor {
private:
  CaptureTypes __captures;
public:
  __functor( CaptureTypes captures )

  auto operator() ( params ) -> ret
    { statements; }
```

Лямбда функции могут возвращать значения

Example122_Lambda2

В случае, если в лямбда-функции только один оператор return то тип значения можно не указывать. Если несколько, то нужно явно указать.

```
[] (int i) -> double
    {
        if (i < 5)
            return i + 1.0;
        else if (i % 2 == 0)
            return i / 2.0;
        else
            return i * i;
    }
```

Захват переменных из внешнего контекста

Example123_Lambda3

```
[]                      // без захвата переменных из внешней области видимости
[=]                     // все переменные захватываются по значению
[&]                     // все переменные захватываются по ссылке
[this]                  // захват текущего класса
[x, y]                  // захват x и y по значению
[&x, &y]                // захват x и y по ссылке
[in, &out]              // захват in по значению, а out — по ссылке
[=, &out1, &out2]       // захват всех переменных по значению, кроме out1 и out2,
                        // которые захватываются по ссылке
[&, x, &y]              // захват всех переменных по ссылке, кроме x…
```

Взболтать но не смешивать: lambda+std::async
Example124_LambdaAsync

```
1.// создаем vector для результатов вычислений
2. std::vector<std::future < double>> results;

3.// запускаем подзадачи с помощью std::async
4.    for (int i = 0; i < 100; i++)
5.        results.push_back(std::async([i]() -> double {
6.            double result = 0;
7.            for(int j=1;j<=STEP;j++) result+= std::log10(i*STEP+j);
8.            return result;
9.        }));

10.// собираем результаты
11.    for (auto &i : results) val+=i.get();
```

Генерация лямбда-выражений

Example125_LambdaGeneration

Начиная со стандарта C++11 шаблонный класс std::function является полиморфной оберткой функций для общего использования. Объекты класса std::function могут хранить, копировать и вызывать произвольные вызываемые объекты - функции, лямбда-выражения, выражения связывания и другие функциональный объекты. Говоря в общем, в любом месте, где необходимо использовать указатель на функцию для её отложенного вызова, или для создания функции обратного вызова, вместо него может быть использован std::function, который предоставляет пользователю большую гибкость в реализации.

Определение класса

```
template<class> class function; // undefined

template<class R, class... ArgTypes> class
function<R(ArgTypes...)>;
```

Функция, создающая функцию

Caller

- a=42
- b=15
- a=42 b=15
- a=42 b=15
- a=42 b=15

Closure

- a,b lives inside c
- a=33 b=16 z=49
- a=33 b=17 z=50
- a=33 b=18 z=51

C++11

```
void caller()
{
int a = 42;
int b = 15;
auto c = [=]() mutable{a = 33;++b; int z =
    a + b; return z; };
c();
c();
c();
}
```

Лямбда в C++14

C++11

```
auto lambda = [](int x, int y)
   {return x + y;}
struct unnamed_lambda {
  auto operator()(int x, int y)
   const {return x + y;}
};
```

C++14

```
auto lambda = [](auto x,
   auto y) {return x + y;}
struct unnamed_lambda {
   template<typename T,
   typename U> auto
   operator()(T x, U y) const
   {return x + y;}
};
```

Лямбды + variadic template Example123_Lambda3_Ex3

```
template <typename T, typename ...Ts>
auto concat(T t, Ts ...ts)
{
if constexpr (sizeof...(ts) > 0)
    return [=](auto ...parameters)
        {
         return t(concat(ts...)(parameters...));
        };
 } else {return t;}
}
```

Guideline for std::thread arguments

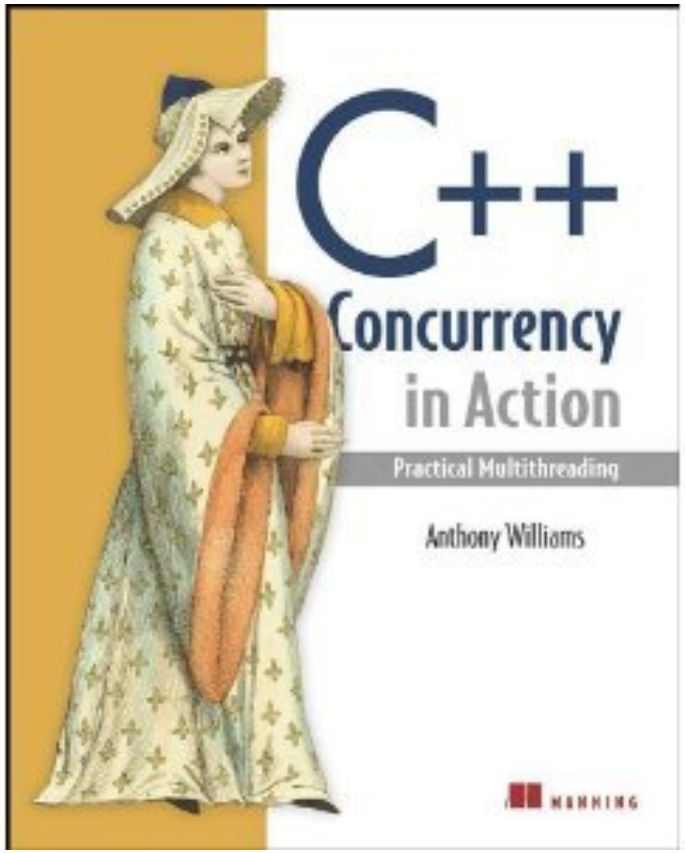
1. Параметры по значению/по ссылке несут риск
   - Лучше передавать параметры по значению
   - И в лямбды то же лучше передавать параметры по значению
2. Две стратегии для преодоления проблем с временем жизни параметра
   - Копирование данных в параллельный поток
   - Построение кода так, что бы данные жили достаточно долго

```
void f(int xParam); // function to call asynchronously
{
    int x;
    ...
    std::thread t1([&]{ f(x); }); // risky! closure holds a ref to x
    std::thread t2([=]{ f(x); }); // okay, closure holds a copy of x
    ...
} // x destroyed
```

Ссылка должна жить достаточное время

```
void f(int xVal, const Widget& wVal); // as before
{
    int x;
    Widget w;
    ...
    std::thread t([&]{ f(x, w); }); // wVal really refers to w
    ...
    t.join(); // destroy w only after t
}
```

Что еще почитать?



C++ Concurrency in Action
Practical Multithreading
Anthony Williams

February, 2012 | 528 pages
ISBN: 9781933988771

Разные блоги, например:

http://habrahabr.ru/post/182610/

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ