

Расширения стандартной
библиотеки Си++ для
многопоточного
программирования в стандарте
Си++ 11



Новый стандарт Си++

19 декабря 2011 года международная организация по стандартизации ISO объявила о выходе стандарта C — ISO/IEC 9899:2011 (ранее неофициально известного как C1X).

Основные изменения по сравнению с C98 и C03:

Тип auto

Диапазонный for

Управление поведением по умолчанию: default и delete

Rvalue ссылки

Лямбда-выражения

Поддержка многопоточности

future & promise

...



Поддержка стандарта транслятором GNU C++

Версия GNU C++ 4.7.2 практически полностью поддерживает параллельные возможности Си++ 11.

Сборка приложения

```
g++ -std=c++11 -o ex2 ex2.cpp -pthread
```

В более ранних версиях

```
g++ -std=c++0x
```



Поддержка многопоточности

Класс **std::thread**,

заголовочный файл **<thread>**



Простейшая параллельная программа

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

void hello()
{
    std::cout<<"Hello Concurrent World\n";
}

int main()
{
    std::thread t(hello);
    t.join();
}
```



Запуск потока с экземпляром класса

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

int main()
{
    class background_task
    {
    public:
        void operator>()()
        {
            std::cout << "Hello\n";
        }
    };
};
```



Запуск потока с экземпляром класса

```
background_task f;  
    std::thread my_thread(f);  
    my_thread.join();  
}
```

В новом потоке создается копия экземпляра класса!



Вариативные шаблоны

Вариативный шаблон или шаблон с переменным числом аргументов в программировании — шаблон с заранее неизвестным числом аргументов, которые формируют один или несколько так называемых пакетов параметров.

Вариативный шаблон позволяет использовать параметризацию типов там, где требуется оперировать произвольным количеством аргументов, каждый из которых имеет произвольный тип. Он может быть очень удобен в тех ситуациях, когда сценарий поведения шаблона может быть обобщён на неизвестное количество принимаемых данных.

Вариативные шаблоны поддерживаются в C++ (начиная со стандарта C++11).

(Википедия)

Вариативные шаблоны

```
// Example program
#include <iostream>
#include <string>

double sum(double x)
{
    return x;
}

template<class... Args>
double sum(double x, Args... args)
{
    return x+sum(args...);
}

int main()
{
    std::cout << "Hello, " << sum(1,2,3,10) << "!\n";
}
```

Конструкторы класса thread

«Пустой» поток

```
thread();
```

Перемещение потока

```
thread( thread&& other );
```

Вызов функции (функтора) в потоке

```
template< class Function, class... Args >  
    explicit thread( Function&& f, Args&&... args );
```

Запрет копирования

```
thread(const thread&) = delete;
```

Функциональные объекты в C++ 11

- Начиная со стандарта C++11 шаблонный класс `std::function` является полиморфной обёрткой функций для общего использования.
- Объекты класса `std::function` могут хранить, копировать и вызывать произвольные вызываемые объекты - функции, лямбда-выражения, выражения связывания и другие функциональные объекты.
- 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.



Функциональные объекты

```
#include <stdio.h>
```

```
#include <functional>
```

```
int sum(int a, int b) {  
    return a + b;  
}
```

```
main() {  
    std::function<int (int, int)> f = sum;  
    printf("1 + 2 = %d\n", f(1, 2));  
}
```



std::bind

функция `std::bind` называется “привязывает” некоторые аргументы к функциональному объекту, создавая новый функциональный объект. То есть, вызов `std::bind` эквивалентен вызову функционального объекта с некоторыми определёнными параметрами.

Передавать можно или непосредственно значения аргументов, или специальные имена, определенные в пространстве имен `std::placeholders`.



std::bind

```
template< class F, class... Args >  
/*unspecified*/ bind( F&& f, Args&&... args );
```

Return value:

A function object of unspecified type T, for which

std::is_bind_expression<T>::value == true,
(объект получен с помощью std::bind)

and which can be stored in std::function



Пример std::bind

```
#include <stdio.h>
#include <functional>
```

```
int sum(int a, int b) {
    return a + b;
}
```

```
main() {
    std::function<int (int, int)> f = sum;
    printf("1 + 2 = %d\n", f(1, 2));
    auto g = std::bind(sum, 1, std::placeholders::_1);
    printf("1 + 4 = %d\n", g(4));
}
```

Специальный объект,
выделяющий параметр для
последующего «связывания»

Неспецифицируемый тип,
вычисляемый в процессе
компиляции



Копирование параметров при передаче в качестве аргументов функции

Параметры всегда копируются и сохраняются во временных объектах перед передачей при использовании **std::bind**. Чтобы этого не происходило, следует использовать конструкцию **std::ref**, позволяющую сделать из объекта ссылку на него, которую можно копировать.



Передача объекта класса по ссылке

Чтобы предотвратить копирование объекта, следует использовать тип **`std::reference_wrapper<T>`**, который является «оберткой» вокруг ссылки типа `T` и возвращает ссылку на объект посредством метода `get()` либо в результате неявного преобразования типа (может использоваться в контексте, где требуется обычная ссылка)

Создается с помощью **`std::ref()`** и **`std::cref()`**.

Этот класс обычно используется для явной передачи объектов ссылочного типа:

```
template <typename T> void foo (T val);  
int x;
```

```
foo (std::ref(x)); // x – ссылка типа int&
```

```
foo (std::cref(x)); // x – ссылка типа const int &
```



Пример использования reference_wrapper

```
#include <iostream>
#include <functional>
```

```
class A {
public:
    void f() {
        std::cout << "Hello, A\n";
    }
};
```

```
void process(A& a) {
    a.f();
}
```

```
int main() {
    A a;
    std::reference_wrapper<A> ra = std::ref(a);
    process(ra);
    return 0;
}
```

```
posypkin@mikhail:~/exp/sandbox/c11$ g++ -std=c++11 -o ref ref.cpp
posypkin@mikhail:~/exp/sandbox/c11$ ./ref
Hello, A
posypkin@mikhail:~/exp/sandbox/c11$
```

Пример использования reference_wrapper

```
#include <iostream>
#include <vector>

class A {
public:

    void f() {
        std::cout << "Hello, A\n";
    }
};

void process(A& a) {
    a.f();
}
```

```
int main() {
    A a;
    std::reference_wrapper<A> ra = std::ref(a);
    std::vector< std::reference_wrapper<A> >
    vra;
    vra.push_back(ra);
    vra.push_back(ra);
    vra.push_back(ra);
    for(auto o : vra) {
        process(o);
    }
    return 0;
}
```

```
posypkin@mikhail:~/exp/sandbox/c11$ g++ -std=c++11 -o vref vref.cpp
posypkin@mikhail:~/exp/sandbox/c11$ ./vref
Hello, A
Hello, A
Hello, A
posypkin@mikhail:~/exp/sandbox/c11$
```

Передача аргументов по ссылке и по значению

```
#include <stdio.h>
#include <functional>
```

```
class C {
public:
    C(int v) : a(v) {}
    int a;
};
```

```
main() {
    C c1(1);
    C& rc1 = c1;
    C c2(2);
    C& rc2 = c2;
```

```
#if 1
    std::reference_wrapper<C> rc3 = c1;
    rc3 = std::ref(rc2);
    rc3.get().a = 10;
    printf("c1.a = %d\n", c1.a);
    printf("c2.a = %d\n", c2.a);
#else
    C& rc3 = c1;
    rc3 = rc2;
    rc3.a = 10;
    printf("c1.a = %d\n", c1.a);
    printf("c2.a = %d\n", c2.a);
#endif
}
```



Использование обычных ссылок

```
C& rc3 = c1;  
rc3 = rc2;  
rc3.a = 10;  
printf("c1.a = %d\n", c1.a);  
printf("c2.a = %d\n", c2.a);
```

напечатает (присваивание ссылок приводит к присваиванию значений объектов)

```
c1.a = 10  
c2.a = 2
```



Использование reference_wrapper

```
std::reference_wrapper<C> rc3 = c1;  
rc3 = std::ref(rc2);  
rc3.get().a = 10;  
printf("c1.a = %d\n", c1.a);  
printf("c2.a = %d\n", c2.a);
```

напечатает (присваивание ссылок приводит к изменению
указываемого объекта)

```
c1.a = 1  
c2.a = 10
```



Пример использования ref для передачи объекта по ссылке

```
class F {  
public:  
    F() : a(10) {}  
  
    void operator()() {  
        a = 5;  
    }  
    int a;  
};
```



Пример использования ref для передачи объекта по ссылке

```
main(int argc, char* argv[]) {  
    F f;  
    std::thread th(f);  
    th.join();  
    printf("a = %d\n", f.a);  
  
    std::thread thr(std::ref(f));  
    thr.join();  
    printf("a = %d\n", f.a);  
}
```

```
mikdebian@debian32: ~/exp/course/c11$ ./thref  
a = 10  
a = 5  
mikdebian@debian32: ~/exp/course/c11$
```



Передача аргументов

Передача аргументов в создаваемый поток производится также просто, как и при обычном вызове функции, например:

```
void f(int i, std::string const& s);  
std::thread t(f, 3, "hello");
```

При этом необходимо помнить, что аргументы **копируются** в запускаемый поток. Игнорирование этого факта может служить источником ошибок.



Возможные скрытые источники ошибок

```
void f(int n, std::string& s);
```

```
void oops(int some_param)
```

```
{
```

```
    char buffer[1024];
```

```
    sprintf(buffer,"%i",some_param);
```

```
    std::thread t(f,3,buffer);
```

```
    t.detach();
```

```
}
```

Преобразование типа производится в созданном потоке, при этом локальная переменная **buffer** может перестать существовать к этому моменту, что вызовет ошибку



Корректный код

```
void not_oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer,"%i",some_param);
    std::thread t(f,3,std::string(buffer));
    t.detach();
}
```

Явное преобразование
производится до создания
экземпляра объекта
std::thread в потоке, в новый
поток копируется полностью
объект типа **std::string**



Передача аргументов по ссылке

```
int a = 5;
main(int argc, char* argv[]) {
    std::thread th(f, a);
    th.join();
    printf("a = %d\n", a);

    std::thread thr(f, std::ref(a));
    thr.join();
    printf("a = %d\n", a);
}
```

```
mikdebian@debian32:~/exp/course/c11$ ./thrparref
a = 5
a = 10
mikdebian@debian32:~/exp/course/c11$ █
```



Получение значения функции, работающей в потоке

- Результат, возвращаемый функциональным объектом-аргументом потока, игнорируется.
- Для передаче родительскому потоку возвращаемого значения, нужно использовать другие конструкции, например `std::future`

Потоки- «демоны»

- Метод **join()** класса **std::thread** позволяет дождаться завершения созданного потока. В этом методе происходит освобождение ресурсов, занятых потоком.
- Метод **detach()** разрывает ассоциацию объекта типа **std::thread** и созданного потока, переменная может использоваться повторно для создания других потоков. Освобождение занятых ресурсов берет на себя библиотека Си ++ и выполняется после завершения потока.
- Все потоки приложения завершаются при завершении приложения.



Пример создания потоков-«демонов»

```
class background_task
{
public:
    void operator()(char c)
    {
        for(int i = 0; i < 10; i++) {
            std::cout << c << std::flush;
            sleep(1);
        }
        std::cout << "thread [" << c << "] finished\n" << std::flush;
    }
};
```



Пример создания потоков-«демонов»

```
main() {  
    background_task f;  
    std::cout << "Enter y to terminate\n" << std::flush;  
    while(true) {  
        char c;  
        std::cin >> c;  
        if(c == 'y')  
            break;  
        else {  
            std::thread my_thread(f, c);  
            my_thread.detach();  
        }  
    }  
}
```



Результат работы программы

Enter y to terminate

a

aaaaba

babababcb

cab

thread [a] finished

bcbcbcb

thread [b] finished

cccc

thread [c] finished

y



Копирование экземпляров класса `std::thread`

Для сокращения числа возможных ошибок не разрешается иметь более одной ссылки на объект типа **`std::thread`**. Поэтому **копирование** запрещено, разрешено только **перемещение** объекта.



std::move

std::move – функция, преобразующая тип **lvalue** (T&) к типу **rvalue** (T&&). Это позволяет перегружать конструкторы по умолчанию и операторы присваивания таким образом, чтобы упростить перемещение объекта.

Применение **std::move** указывает на то, что аргумент не может далее использоваться и должен «умереть» после завершения присваивания. Чтобы избежать двойного удаления аллоцированные объекты конструктор перемещения или оператор присваивания с семантикой перемещения обнуляет аллоцируемые объекты перемещаемого класса (правой части присваивания) или переписывает туда указатели на объекты изменяемого класса (левая часть присваивания).

Обычно имеют два конструктора и оператора присваивания (для lvalue и rvalue).

C(C && c) // перемещение

C(const C & c) // копирование



Неправильное использование rvalue

```
class A {  
    char* p;  
  
    ...  
    A(A && a) {  
        delete [] p;  
        p = a.p;  
        //последующий вызов  
        delete  
        //в деструкторе класса a  
        //приведет к ошибке  
    }
```



Правильное использование **rvalue**

1. Обнуление

```
class A {  
    char* p;  
    ...  
    A(A && a) {  
        delete [] p;  
        p = a.p;  
        a.p = nullptr; //  
        //последующий вызов delete  
        //в деструкторе класса a не  
        //приведет к ошибке  
    }
```

2. Обмен указателями

```
class A {  
    char* p;  
    ...  
    A(A && a) {  
        char *tmp = p;  
        p = a.p;  
        a.p = tmp; // будет  
        // освобожден при  
        // вызове деструктора a  
    }
```



Нюансы использования **rvalue**

```
class C {  
    public:  
    C(int n) : num(n) {  
    }
```

```
    int num;  
};
```

```
void f(C && rc) {  
    std::cout << "rvalue = " << rc.num << "\n";  
}
```

```
void f(C & rc) {  
    std::cout << "lvalue = " << rc.num << "\n";  
}
```

```
int main()  
{  
    C c(1);  
    f(c);  
    f(std::move(c));  
    f(g(2));  
    f(C(3));  
}
```

```
lvalue = 1  
rvalue = 1  
rvalue = 2  
rvalue = 3
```

Конструкторы и операторы присваивания `std::thread`

`std::thread::thread`

<code>thread();</code>	(1)	(since C++11)
<code>thread(thread&& other);</code>	(2)	(since C++11)
<code>template< class Function, class... Args > explicit thread(Function&& f, Args&&... args);</code>	(3)	(since C++11)
<code>thread(const thread&) = delete;</code>	(4)	(since C++11)

`std::thread::operator=`

<code>thread& operator=(thread&& other);</code>	(since C++11)
---	---------------

Assigns the state of `other` to `*this` using move semantics.

If `*this` still has an associated running thread (i.e. `joinable() == true`), `std::terminate()` is called



Копирование экземпляров класса `std::thread`

```
std::thread my_thread(f);  
std::thread new_thread;
```

Неправильно:

```
new_thread = my_thread;
```

Правильно:

```
new_thread = std::move(my_thread);
```



Возврат потока функцией – корректный вариант

```
std::thread make_thread() {  
    background_task f;  
    std::thread my_thread(f);  
    return my_thread;  
}  
main() {  
    std::thread new_thread;  
    new_thread = make_thread();  
    new_thread.join();  
}
```

Возвращается
rvalue



Запуск группы потоков

```
#include <thread>
#include <iostream>
#include <vector>

void bt() {
    printf("Hello\n");
}

main() {
    std::vector<std::thread> v;
    for(int i = 0; i < 8; i++) {
        std::thread th(bt);
        v.push_back(std::move(th));
    }
    for(auto& t : v) t.join();
}
```

```
admin@irbis8:~/exp/lectures$ ./ex5
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```



Идентификация потоков

Идентификатор имеет тип **`std::thread::id`** и является уникальным.

Методы для получения идентификатора:

Метод **`get_id()`** класса **`std::thread`** позволяет получить идентификатор потока, соответствующего заданному классу. Метод **`std::this_thread::get_id()`** возвращает экземпляр вызывающего потока.



Идентификация потоков

```
std::thread::id master_id;
```

```
void bt() {  
    if(std::this_thread::get_id() == master_id)  
        printf("Hello from master thread\n");  
    else  
        printf("Hello\n");  
}
```



Идентификация потоков

```
main() {  
    std::vector<std::thread> v;  
    int n = 8;  
    for(int i = 0; i < n; i++) {  
        std::thread th(bt);  
        if(i == 0) master_id = th.get_id();  
        v.push_back(std::move(th));  
    }  
    for(auto& t : v) t.join();  
}
```

Ошибка!!! `mastr_id` может не
получить значение до вызова
`bt()`

```
admin@irbis8:~/exp/lectures$ ./ex6  
Hello from master thread  
Hello  
Hello  
Hello  
Hello  
Hello  
Hello  
Hello  
Hello
```



Вычисление интеграла

```
void integrate(int n, int start, int step, const
std::function <double (double)>& f, double&
rv) {
    double h = (B - A) / n;
    double v = 0;
    for(int i = start; i < n; i += step) {
        double x = A + i*h + h/2;
        v += f(x);
    }
    rv = v * h;
}
```



Вычисление интеграла

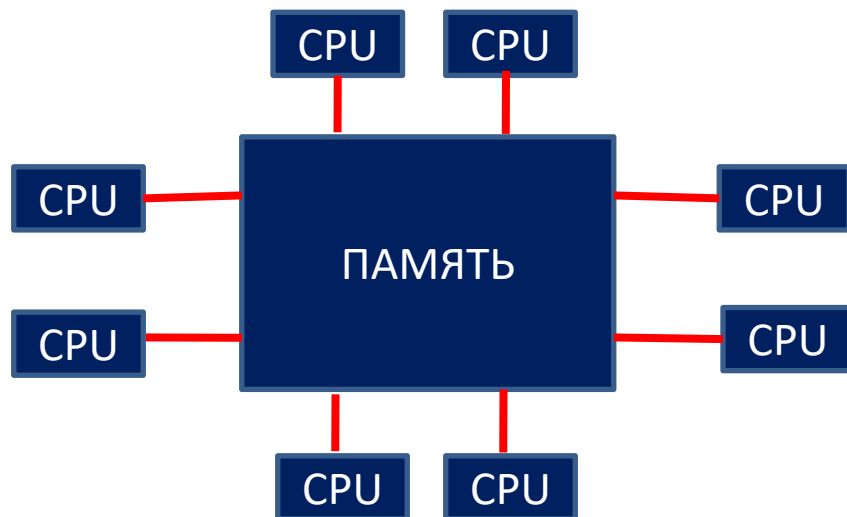
```
int main(int argc, char* argv[]) {  
    int n, nt;  
    if(argc != 3) {  
        std::cout << "usage: " << argv[0] << "  
number_of_boxes number_of_threads\n";  
        return -1;  
    }  
    n = atoi(argv[1]);  
    nt = atoi(argv[2]);  
    double v[nt];  
    std::thread thr[nt];
```

```
    for(int i = 0; i < nt; i++) {  
        std::thread t(integrate, n, i, nt,  
std::cref(sin),std::ref(v[i]));  
        thr[i] = std::move(t);  
    }  
    double u = 0;  
    for(int i = 0; i < nt; i++) {  
        thr[i].join();  
        u += v[i];  
    }  
    std::cout << "The integral value is " << u <<  
"\n";  
    return 0;  
}
```



Экспериментальные исследования

Эксперименты проводились на с двумя процессорами Intel®Xeon® E5620 (2.40)ГГц, 16 Гб оперативной памяти, 8 вычислительных ядер суммарно с гипертрейдингом – 16 виртуальных.



Результат работы программы на экране

```
mposypkin@u5: ~/exp/myanm/numint
mposypkin@u5:~/exp/myanm/numint$ time ./numintp 100000000 1
The integral value is 2

real    0m11.088s
user    0m11.064s
sys     0m0.024s
mposypkin@u5:~/exp/myanm/numint$ time ./numintp 100000000 2
The integral value is 2

real    0m5.557s
user    0m11.067s
sys     0m0.030s
mposypkin@u5:~/exp/myanm/numint$ time ./numintp 100000000 4
The integral value is 2

real    0m2.788s
user    0m11.072s
sys     0m0.032s
mposypkin@u5:~/exp/myanm/numint$ time ./numintp 100000000 8
The integral value is 2

real    0m1.529s
user    0m11.330s
sys     0m0.040s
mposypkin@u5:~/exp/myanm/numint$ time ./numintp 100000000 16
The integral value is 2

real    0m1.073s
user    0m16.429s
sys     0m0.091s
mposypkin@u5:~/exp/myanm/numint$ time ./numintp 100000000 32
The integral value is 2

real    0m1.074s
user    0m16.465s
sys     0m0.116s
mposypkin@u5:~/exp/myanm/numint$
```

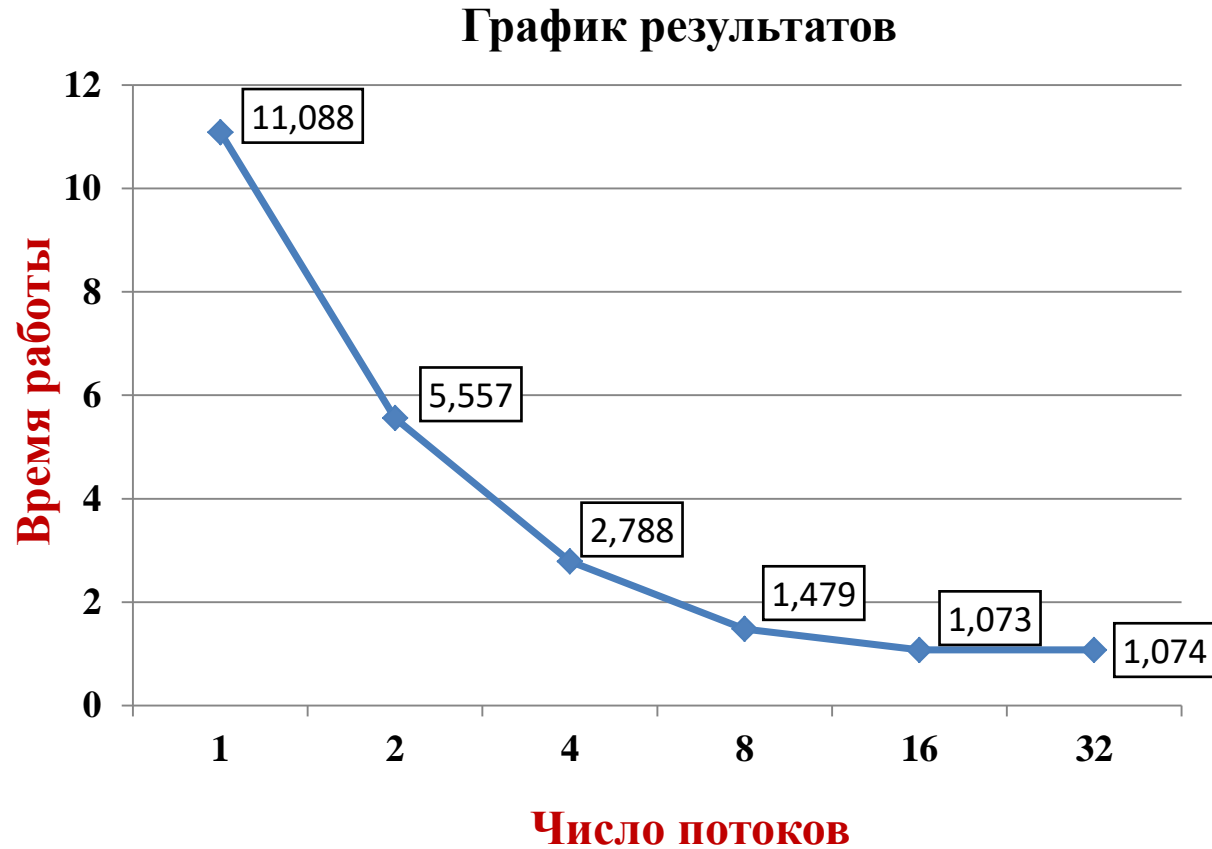


Таблица результатов эксперимента время решения на языке C++11

Число потоков	1	2	4	8	16	32
Время работы	11.088	5.557	2.788	1.479	1.073	1.074



График зависимости времени работы параллельной программы от числа запускаемых потоков на языке C++11



Мьютексы

Класс **std::mutex**

Методы:

lock() – захват мьютекса (должен вызываться только если поток не владеет мьютексом) – приводит к блокировке потока до момента освобождения мьютекса другим потоком

unlock() – освобождение мьютекса (должен вызываться только, если мьютекс захвачен потоком)

bool try_lock() – Пытается заблокировать мьютекс. Возврат происходит немедленно. В случае успешной установки блокировки возвращается true, в противном случае возвращается false.



Resource Acquisition Is Initialization (RAII)

Концепция, предполагающая что получение ресурсов происходит при создании объекта, а освобождение происходит при вызове деструктора.

Это позволяет гарантировать освобождение в случае возникновения исключений, т.к. при обработке исключений вызываются деструкторы объектов, находящихся на стеке.



Опасный подход к выделению ресурсов

```
class A {  
public:  
    void alloc() { printf("resource allocated\n");}  
    void free() { printf("resource freed\n"); }  
};  
  
main() {  
    A a;  
  
    try {  
        a.alloc();  
        throw E();  
        a.free();  
    } catch(...) {  
    }  
}
```

Не
вызывается

```
mikdebian@debian32:~/exp/course/c11$ ./rail1  
resource allocated  
mikdebian@debian32:~/exp/course/c11$
```



Правильный подход к управлению ресурсами

```
class A {  
public:  
    A() { printf("resource allocated\n");}  
    ~A() { printf("resource freed\n"); }  
};
```

```
struct E {  
};
```

```
main() {  
    try {  
        A a;  
        throw E();  
    } catch(...) {  
    }  
}
```

Деструктор вызывается
при возникновении
исключения и ресурсы
освобождаются

```
mikdebian@debian32: ~/exp/course/c11$ ./raii  
resource allocated  
resource freed  
mikdebian@debian32: ~/exp/course/c11$
```



RAII применительно к мьютексам

```
#include <mutex>
```

```
std::mutex mut;
```

```
int a;
```

```
void f() {
```

```
    std::lock_guard<std::mutex> lg(mut);
```

```
    a ++;
```

```
}
```

`std::lock_guard` – обертка для мьютекса, обеспечивающая его захват при создании «гарда» и освобождение на выходе из блока (в том числе и при исключении)



Взаимная блокировка и ее преодоление

Thread1:

```
mutex1.lock();  
mutex2.lock();
```

Thread2:

```
mutex2.lock();  
mutex1.lock();
```

Необходимость захвата двух мьютексов может быть вызвана защитой двух элементов данных в некоторой операции (mutex1 защищает первый элемент, mutex2 – второй).



Взаимная блокировка и ее преодоление

Thread1:

```
mutex1.lock();  
mutex2.lock();
```

Thread2:

```
mutex2.lock();  
mutex1.lock();
```

Возникает взаимная блокировка из-за того, что каждый из потоков пытается выполнить захват мьютекса, захваченного ранее другим потоком.



Взаимная блокировка и ее преодоление

```
template< class Lockable1, class Lockable2, class... LockableN >  
void lock( Lockable1& lock1, Lockable2& lock2, LockableN&... lockn );
```

Выполняет безопасную блокировку нескольких мьютексов одновременно

Пример использования для защиты некоторой операции some_op()

```
{  
std::lock(lhs.m,rhs.m);  
std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock);  
std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock);  
return some_op(lhs.some_detail, rhs.some_detail);  
}
```

std::adopt_lock означает, что lock_guard создается на базе заблокированного ранее мьютекса



std::unique_lock

Класс `unique_lock` является оболочкой над мьютексами, которая обладая всем потенциалом `lock_guard` позволяет выполнять отложенную блокировку, попытки блокировки с ограничением по времени ожидания, рекурсивную блокировку и использование с **условными переменными**.

The class `unique_lock` is movable, but not copyable -- it meets the requirements of `MoveConstructible` and `MoveAssignable` but not of `CopyConstructible` or `CopyAssignable`.

Решение проблемы взаимной блокировки с использованием `unique_lock`

```
void transfer(Box &from, Box &to, int num)
{
    // don't actually take the locks yet
    std::unique_lock<std::mutex> lock1(from.m, std::defer_lock);
    std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);

    // lock both unique_locks without deadlock
    std::lock(lock1, lock2);

    from.num_things -= num;
    to.num_things += num;

    // 'from.m' and 'to.m' mutexes unlocked in 'unique_lock' destructors
}
```

Основные методы `unique_lock`

`lock` – блокирует ассоциированный мьютекс

`try_lock` выполняет попытку захвата мьютекса, возвращает управление, если мьютекс недоступен (захвачен другим потоком)

`try_lock_for` для `TimeLockable` мьютексов выполняет попытку захвата, возвращает управление, если мьютекс недоступен в течение указанного периода

`try_lock_until` для `TimeLockable` мьютексов выполняет попытку захвата, возвращает управление, если мьютекс недоступен до указанного момента времени

`unlock` разблокирует ассоциированный мьютекс

Перемещение unique_lock

```
std::unique_lock<std::mutex> get_lock()
{
    extern std::mutex some_mutex;
    std::unique_lock<std::mutex> lk(some_mutex);
    prepare_data();
    return lk;
}
```

```
void process_data()
{
    std::unique_lock<std::mutex> lk(get_lock());
    do_something();
}
```

unique_lock позволяет передавать «владение» мьютексом между функциями через возвращаемое значение или через присваивание в форме перемещения

Рекурсивная блокировка

`std::recursive_mutex`

Позволяет выполнять рекурсивную блокировку, т.е. поток может несколько раз подряд блокировать данный мьютекс. При этом разблокировка выполняется то же число раз, что и выполнялась блокировка.

Не рекомендуется к использованию, т.к. ведет к плохому дизайну.

shared_mutex (C++ 17)

Подразумевает два типа владения: эксклюзивный и не эксклюзивный. В неэксклюзивном режиме мьютексом могут владеть сразу несколько потоков. Это дает возможность различать блокировку на чтение и запись.

Основные методы:

Exclusive locking

lock	locks the mutex, blocks if the mutex is not available
try_lock	tries to lock the mutex, returns if the mutex is not available
unlock	unlocks the mutex

Shared locking

lock_shared	locks the mutex for shared ownership, blocks if the mutex is not available
try_lock_shared	tries to lock the mutex for shared ownership, returns if the mutex is not available
unlock_shared	unlocks the mutex (shared ownership)

shared_lock (C++ 14)

Оболочка общего назначения для мьютексов, допускающих два режима блокировки (эксклюзивный режим и неэксклюзивный).

```
<template< class Mutex >  
class shared_lock;
```

Mutex - the type of the shared mutex to lock. The type must meet the SharedMutex requirements

SharedMutex:

std::shared_mutex(since C++17)

std::shared_timed_mutex(since C++14)

```
class ThreadSafeCounter {  
public:  
    ThreadSafeCounter() = default;
```

// Multiple threads/readers can read the counter's value at the same time.

```
    unsigned int get() const {  
        std::shared_lock<std::shared_mutex> lock(mutex_);  
        return value_;  
    }
```

// Only one thread/writer can increment/write the counter's value.

```
    void increment() {  
        std::unique_lock<std::shared_mutex> lock(mutex_);  
        value_++;  
    }
```

// Only one thread/writer can reset/write the counter's value.

```
void reset() {  
    std::unique_lock<std::shared_mutex> lock(mutex_);  
    value_ = 0;  
}
```

private:

```
mutable std::shared_mutex mutex_;  
unsigned int value_ = 0;  
};
```

```
int main() {
    ThreadSafeCounter counter;

    auto increment_and_print = [&counter]() {
        for (int i = 0; i < 3; i++) {
            counter.increment();
            std::cout << std::this_thread::get_id() << ' ' << counter.get() << '\n';

            // Note: Writing to std::cout actually needs to be synchronized as well
            // by another std::mutex. This has been omitted to keep the example
            small.
        }
    };

    std::thread thread1(increment_and_print);
    std::thread thread2(increment_and_print);

    thread1.join();
    thread2.join();
}
```

Условные переменные

Тип **std::condition_variable**

Основные методы

notify_one – уведомить один ждущий поток

notify_all – уведомить все ждущие потоки

wait - блокировка потока до пробуждения

wait_for – блокировка до пробуждения или истечения времени

wait_until – блокировка до пробуждения или до заданного момента времени



Ожидание «пробуждения»

```
void wait( std::unique_lock<std::mutex>& lock );
```

Ставит поток в очередь на ожидание по данной переменной. Возвращает управление в случае если другой поток вызвал **notify_one ()** или **notify_all()** по этой переменной. Допускаются также спонтанные срабатывания.

Для защиты от ложных срабатываний добавляется предикат

```
template< class Predicate >  
void wait( std::unique_lock<std::mutex>& lock, Predicate pred );
```

Сигнатура предиката: **bool pred()**

Эквивалентно коду:

```
while (!pred()) {  
    wait(lock);  
}
```



Условные переменные: пример

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;
void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}
```



Условные переменные: пример

```
void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

Лямбда-
выражение



Модификатор **mutable**

Переменная или поле данных класса, объявленное с этим модификатором, может изменяться в константных функциях.

Позволяет делать константные методы thread-safe, защищая чтение данных с помощью mutex-ов, объявленных с помощью mutable.



Многопоточная очередь

```
template<typename T>
class threadsafe_queue
{
private:
mutable std::mutex mut;
std::queue<T> data_queue;
std::condition_variable data_cond;
```

```
public:
threadsafe_queue() {}
```

```
threadsafe_queue(threadsafe_queue const&
other)
{
std::lock_guard<std::mutex> lk(other.mut);
data_queue=other.data_queue;
}
```

mutable нужен для
использования мьютекса в
константных методах



Многопоточная очередь

```
void push(T new_value)
{
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(new_value);
    data_cond.notify_one();
}
```

```
void wait_and_pop(T& value)
{
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    value=data_queue.front();
    data_queue.pop();
}
```



Многопоточная очередь

```
std::shared_ptr<T> wait_and_pop()
{
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    std::shared_ptr<T>
    res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}
```

В константном методе
используется mutex,
объявленный как
mutable

```
bool empty() const
{
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
}
```



Атомарные типы

- Над данными атомарного типа определены атомарные операции, т.е. операции, состояние которых доступно другим потокам либо до, либо после, но не в процессе операции.
- Атомарные операции могут читаться/записываться без синхронизации разными потоками.
- Одновременное обращения к общим неатомарным данным без синхронизации приводит к неопределенному поведению.

Общий шаблон для атомарных ТИПОВ

template< class T > struct atomic;

Определяет атомарную «версию» типов, которые:

1. поддерживают тривиальную генерируемую операцию присваивания;
2. поддерживают тривиальное побитовое сравнение.

Пример

```
#include <iostream>
#include <atomic>
```

```
class A {
public:
    A(int a) : mA(a) {
    }

    virtual void f() = 0;

protected:
    int mA;
};
```

```
class B : public A {
public:
    B(int b) : A(b) {
    }

    void f() {
        std::cout << A::mA;
    }
};
```

```
class C {
public:
    C(int a) : mA(a) {
    }

    void f() {
        std::cout << mA << "\n";
    }

protected:
    int mA;
};
```



Какие варианты правильные?

```
int main() {  
  
    B b(1);  
    A& a = b;  
    C c(3);  
    std::atomic<A> atomicb(a);  
    std::atomic<B> atomicb(b);  
    std::atomic<C> atomicc(c);  
    return 0;  
}
```

Какие варианты правильные?

```
int main() {
```

```
    B b(1);
```

```
    A& a = b;
```

```
    C c(3);
```

```
    std::atomic<A> atomicb(a);
```

```
    std::atomic<B> atomicb(b);
```

```
    std::atomic<C> atomicc(c);
```

```
    return 0;
```

```
}
```

Абстрактный класс

Имеет виртуальную
функцию f()

Атомарные варианты для арифметических типов

<code>std::atomic_bool</code>	<code>std::atomic<bool></code>
<code>std::atomic_char</code>	<code>std::atomic<char></code>
<code>std::atomic_schar</code>	<code>std::atomic<signed char></code>
<code>std::atomic_uchar</code>	<code>std::atomic<unsigned char></code>
<code>std::atomic_short</code>	<code>std::atomic<short></code>
<code>std::atomic_ushort</code>	<code>std::atomic<unsigned short></code>
<code>std::atomic_int</code>	<code>std::atomic<int></code>
<code>std::atomic_uint</code>	<code>std::atomic<unsigned int></code>
<code>std::atomic_long</code>	<code>std::atomic<long></code>
<code>std::atomic_ulong</code>	<code>std::atomic<unsigned long></code>
<code>std::atomic_llong</code>	<code>std::atomic<long long></code>
<code>std::atomic_ullong</code>	<code>std::atomic<unsigned long long></code>

`std::atomic<T*>` определен для всех типов указателей.

Копирование атомарных объектов

Атомарные объекты нельзя копировать, поскольку копирование вовлекает чтение и запись двух объектов – такая операция не может быть атомарной.

Копирование атомарных объектов

- Атомарные объекты нельзя присваивать друг другу и нельзя использовать в конструкторе копирования
- Объекты базового типа могут использоваться в конструкторе атомарной версии (конструктор не является атомарной операцией)
- Можно присваивать значения атомарных объектов соответствующим типам языка C++, эти операции атомарны
- Можно присваивать атомарному объекту значения базового типа, эта операция атомарна

Примеры присваивания atomic

```
#include <iostream>
#include <atomic>
int main() {
    int a1 = 1;
    int a2 = 2;
    std::atomic<int> ai1;
    std::atomic<int> ai2(a2);
    ai1 = ai2;
    return 0;
}
```

```
./atomiccopy.cpp: In function 'int main()':
./atomiccopy.cpp:11:7: error: use of deleted function 'std::atomic<int>
& std::atomic<int>::operator=(const std::atomic<int>&)'
    ai1 = ai2;
    ^
In file included from ./atomiccopy.cpp:2:0:
/usr/include/c++/5/atomic:613:15: note: declared here
    atomic& operator=(const atomic&) = delete;
    ^
posypkin@mikhail:~/exp/sandbox/c11/atomic$
```

Примеры присваивания atomic

```
#include <iostream>
```

```
#include <atomic>
```

```
int main() {  
    int a1 = 1;  
    int a2 = 2;  
    std::atomic<int> ai1;  
    std::atomic<int> ai2(a2);  
    ai1 = a1;  
    a2 = ai1;  
    std::cout << "a1 = " << a1 << ", a2 = " << a2 << ", ai1 = " << ai1 << ", ai2 = " << ai2 <<  
    "\n";  
    return 0;  
}
```

```
posypkin@mikhail:~/exp/sandbox/c11/atomic$ ./atomiccopy  
a1 = 1, a2 = 1, ai1 = 1, ai2 = 2
```

Различия со стандартным присваиванием

- Обычное присваивание возвращает ссылку на объект из левой части

`T& class_name :: operator= (T)`

- Присваивание атомарной переменной возвращает присаиваемое значение

`T operator=(T desired);`

Пример, показывающий различия в семантике присваивания

```
#include <iostream>
```

```
#include <atomic>
```

```
int main() {
```

```
    int a = 1;
```

```
    (a = 2) = 3;
```

Ok

```
    std::cout << "a = " << a << "\n";
```

Напечатает 3

```
    std::atomic<int> ai(a);
```

```
    ai = 5;
```

Ok

```
    (ai = a) = 5;
```

```
    return 0;
```

```
}
```

Ошибка (нельзя присваивать не l-value)

Lock Free

- Для некоторых типов реализация `std::atomic` не требует использования внешних синхронизаций, так как на целевой платформе поддерживается аппаратная синхронизация для базовых типов (`int`, `double` ...). (**lock free**)
- Для других типов, напротив, требуется использование примитивов синхронизации (мьютексы и т.п.). (**not lock free**)
- Аппаратная поддержка атомарности операций намного эффективнее, так как не требует синхронизации.

Проверка lock-free

- Метод **is_lock_free** возвращает **true** для безблокировочных атомарных типов и **false** для всех остальных
- Стандартные атомарные типы, как правило, **lock free**

Пример

```
#include <iostream>
#include <atomic>
```

```
struct C1 {
    int mA;
    double mD;
};
```

```
struct C2 {
    char mC[100];
};
```

```
struct C3 {
    int mA;
    double mD;
    double mH;
};
```

```
struct C4 {
    int mA;
    double mD;
    double mH;
    char mC[100];
};
```

Пример

```
int main() {  
  
    std::atomic_int ai;  
    std::cout << "atomic_int is " << (ai.is_lock_free() ? "lock free " : "not  
lock free ") << "\n";  
  
    std::atomic<int*> api;  
    std::cout << "std::atomic<int*> is " << (api.is_lock_free() ? "lock free  
" : "not lock free ") << "\n";  
}
```

Пример

```
{  
    C1 c;  
    std::atomic<C1> atomicc(c);  
    std::cout << "atomic<C1> is " << (atomicc.is_lock_free() ? "lock free  
" : "not lock free ") << "\n";  
}
```

```
{  
    C2 c;  
    std::atomic<C2> atomicc(c);  
    std::cout << "atomic<C2> is " << (atomicc.is_lock_free() ? "lock free  
" : "not lock free ") << "\n";  
}
```

Пример

```
{
    C3 c;
    std::atomic<C3> atomicc(c);
    std::cout << "atomic<C3> is " << (atomicc.is_lock_free() ? "lock free " :
"not lock free ") << "\n";
}

{
    C4 c;
    std::atomic<C4> atomicc(c);
    std::cout << "atomic<C4> is " << (atomicc.is_lock_free() ? "lock free " :
"not lock free ") << "\n";
}

return 0;
}
```

Результат выполнения

В gcc необходимо при сборке подключать библиотеку `atomic` с помощью флага **-latomic**

```
posypkin@mikhail: ~/exp/sandbox/c11/atomic
posypkin@mikhail:~/exp/sandbox/c11/atomic$ c++ -std=c++11 -o lockfree ./lockfree.cpp -latomic
posypkin@mikhail:~/exp/sandbox/c11/atomic$ ./lockfree
atomic_int is lock free
std::atomic<int*> is lock free
atomic<C1> is lock free
atomic<C2> is not lock free
atomic<C3> is not lock free
atomic<C4> is not lock free
posypkin@mikhail:~/exp/sandbox/c11/atomic$
```


std::atomic_flag

- Простейший атомарный объект логического (булевского) типа.
- Является lock free
- Не допускает присваивания и конструктора копирования

Методы `std::atomic`

- **clear** – атомарная операция присваивания флагу значения `false`
- **test_and_set** – атомарная операция, которая устанавливается значение флага в `true` и возвращает его предыдущее значение

vector::emplace_back

- Создание элемента контейнера «на месте» без дополнительного копирования
- Как правило, выигрывает по производительности у push_back

`template< class... Args > void emplace_back(Args&&... args);` C++ 11


`template< class... Args > reference emplace_back(Args&&... args);` C++ 17
(возвращает ссылку на созданный элемент в контейнере)

Пример использования atomic_flag для организации блокировок

```
#include <unistd.h>
#include <thread>
#include <vector>
#include <iostream>
#include <atomic>

std::atomic_flag lock = ATOMIC_FLAG_INIT;

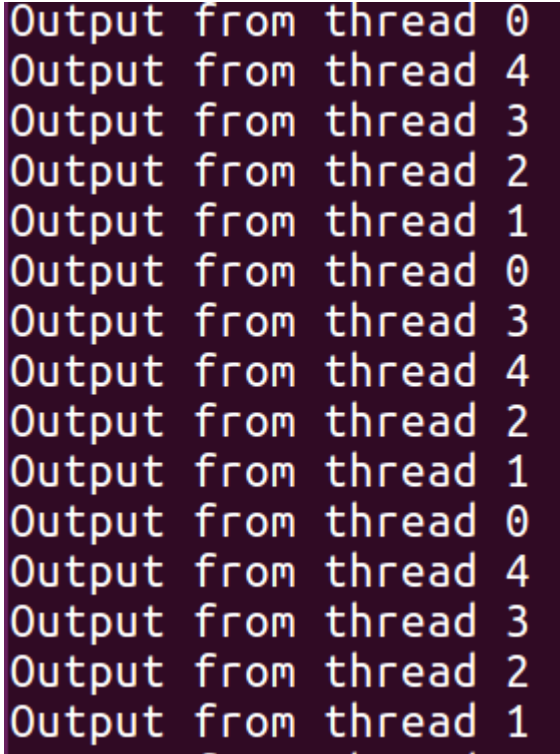
void f(int n)
{
    for (int cnt = 0; cnt < 10; ++cnt) {
        while (lock.test_and_set()) ;
        std::cout << "Output from thread " << n << '\n';
        lock.clear();
        sleep(1);
    }
}
```



Циклическая
блокировка (не
лучший, хотя и
корректный,
способ)

Пример использования `atomic_flag` для организации блокировок

```
int main()
{
    std::vector<std::thread> v;
    for (int n = 0; n < 5; ++n) {
        v.emplace_back(f, n);
    }
    for (auto& t : v) {
        t.join();
    }
}
```



```
Output from thread 0
Output from thread 4
Output from thread 3
Output from thread 2
Output from thread 1
Output from thread 0
Output from thread 3
Output from thread 4
Output from thread 2
Output from thread 1
Output from thread 0
Output from thread 4
Output from thread 3
Output from thread 2
Output from thread 1
Output from thread 0
Output from thread 4
Output from thread 3
Output from thread 2
Output from thread 1
```

Методы класса `std::atomic`

- `T load(std::memory_order order = std::memory_order_seq_cst) const;` - атомарное чтение значения атомарного объекта (эквивалентна `()`)
- `void store(T desired, std::memory_order order = std::memory_order_seq_cst);` - атомарная запись данных в атомарный объект (эквивалентна `=`)

Методы класса `std::atomic`

- `T exchange(T desired, std::memory_order order = std::memory_order_seq_cst);`

Атомарная операция. Заменяет значения атомарной переменной на значение `desired`, возвращает старое значение.

Методы класса `std::atomic`

- `bool compare_exchange_weak(T& expected, T desired, std::memory_order order = std::memory_order_seq_cst);`
- `bool compare_exchange_strong(T& expected, T desired, std::memory_order order = std::memory_order_seq_cst);`

Если значение атомарной переменной совпадает со значением `expected`, то ее значение заменяется на `desired`. Возвращаемое значение при этом `true`. В противном случае, значение атомарной переменной записывается в `expected`. Возвращаемое значение при этом `false`.

Методы класса `std::atomic`

- `bool compare_exchange_weak` – не всегда срабатывает, если `expected` совпадает со значением атомарной переменной, поэтому должна использоваться в цикле с проверкой возвращаемого значения (*spurious failure*)
- `bool compare_exchange_strong` – если `expected` совпадает со значением атомарной переменной, то запись производится

Если значение атомарной переменной совпадает со значением `expected`, то ее значение заменяется на `desired`. Возвращаемое значение при этом `true`. В противном случае, значение атомарной переменной записывается в `expected`. Возвращаемое значение при этом `false`.

Цикл для compare_exchange_weak

```
expected = current.load();  
do desired = function(expected);  
while (!current.compare_exchange_weak(expected, desired));
```

Семантика

compare_exchange_strong/weak

```
if (memcmp(object, expected, sizeof(*object)) == 0)
    memcpy(object, &desired, sizeof(*object));
else
    memcpy(expected, object, sizeof(*object));
```

Может отличаться от сравнения с помощью == и приводить к парадоксальным результатам. Особенно актуально для нецелочисленной арифметики.

Атомарные операции сложного присваивания

```
T fetch_add( T arg,  
             memory_order = std::memory_order_seq_cst );
```

Атомарно добавляет `arg` к значению, хранящемуся в атомарном объекте, возвращает «старое» значение. Эквивалентно оператору `+=` (с точностью до аргумента, определяющего дисциплину работы с памятью)

```
T fetch_sub( T arg,    memory_order = std::memory_order_seq_cst );
```

Атомарно вычитает `arg` из значения, хранящегося в атомарном объекте, возвращает «старое» значение. Эквивалентно оператору `-=` (с точностью до аргумента, определяющего дисциплину работы с памятью)

Атомарные операции сложного присваивания

```
T fetch_and( T arg, memory_order = std::memory_order_seq_cst );
```

Атомарно выполняет побитовое И между аргументом и значением атомного объекта, изменяя значение атомарного объекта, и возвращает «старое» значение. Эквивалентно оператору `&=` (с точностью до аргумента, определяющего дисциплину работы с памятью)

```
T fetch_or( T arg, memory_order = std::memory_order_seq_cst );
```

Атомарно выполняет побитовое ИЛИ между аргументом и значением атомного объекта, изменяя значение атомарного объекта, и возвращает «старое» значение. Эквивалентно оператору `|=` (с точностью до аргумента, определяющего дисциплину работы с памятью)

Атомарные операции сложного присваивания

```
T fetch_xor ( T arg, memory_order =  
std::memory_order_seq_cst );
```

Атомарно выполняет побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ между аргументом и значением атомного объекта, изменяя значение атомарного объекта, и возвращает «старое» значение. Эквивалентно оператору `|=` (с точностью до аргумента, определяющего дисциплину работы с памятью)

Operation	atomic_flag	atomic<bool>	atomic<T*>	atomic<integral-type>	atomic<other-type>
test_and_set	✓				
clear	✓				
is_lock_free		✓	✓	✓	✓
load		✓	✓	✓	✓
store		✓	✓	✓	✓
exchange		✓	✓	✓	✓
compare_exchange_weak, compare_exchange_strong		✓	✓	✓	✓
fetch_add, +=			✓	✓	
fetch_sub, -=			✓	✓	
fetch_or, =				✓	
fetch_and, &=				✓	
fetch_xor, ^=				✓	
++, --			✓	✓	

Модели памяти

- sequentially consistent ordering
(memory_order_seq_cst)
- acquire-release ordering
(memory_order_consume,
memory_order_acquire,
memory_order_release, and
memory_order_acq_rel)
- relaxed ordering (memory_order_relaxed)

ПОСЛЕДОВАТЕЛЬНАЯ МОДЕЛЬ

Последовательная модель

- Является моделью памяти по-умолчанию
- Наиболее интуитивно понятна
- Считается, что введена Leslie Lamport в статье Lamport L. How to make a multiprocessor computer that correctly executes multiprocess program //IEEE transactions on computers. – 1979. – №. 9. – С. 690-691.:

«the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program»

Последовательная модель

- Все изменения данных одинаковым образом видны на всех потоках, так, как **если бы** все атомарные операции выполнялись в некотором заданном порядке
- Изменения согласованы с порядком следования операций в каждом из потоков

- 1) фактический порядок выполнения может отличаться
- 2) важен только порядок изменения памяти

Пример последовательной модели

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;

void write_x()
{
    x.store(true, std::memory_order_seq_cst);
}

void write_y()
{
    y.store(true, std::memory_order_seq_cst);
}
```

Пример последовательной модели

```
void read_x_then_y()  
{  
    while(!x.load(std::memory_order_seq_cst));  
    if(y.load(std::memory_order_seq_cst))  
        ++z;  
}
```

```
void read_y_then_x()  
{  
    while(!y.load(std::memory_order_seq_cst));  
    if(x.load(std::memory_order_seq_cst))  
        ++z;  
}
```

Пример последовательной модели

```
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);
}
```

Разные порядки выполнения операций

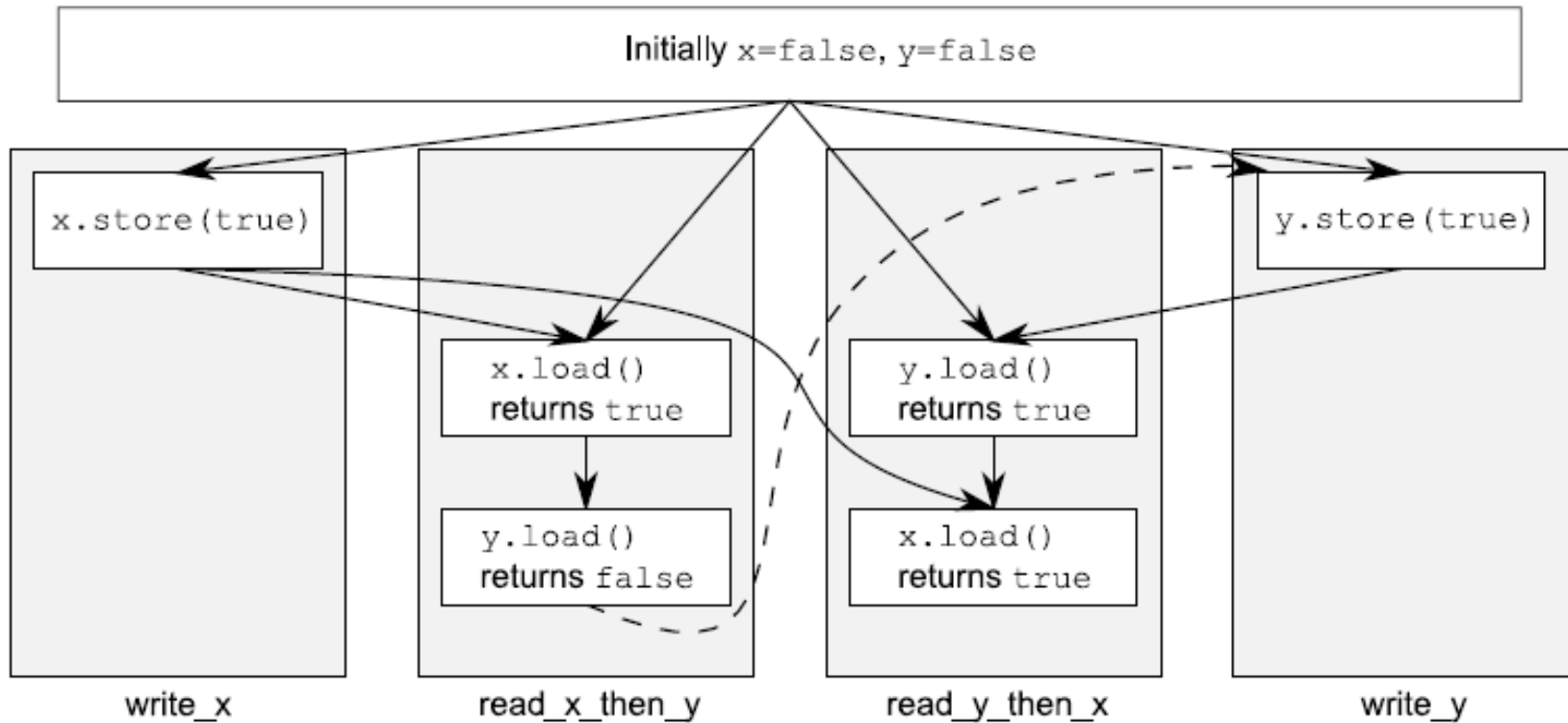
ВОЗМОЖНО **wx** **wy** **rx** **ry** **ry** **rx** **z++** **z++** **z = 2**

ВОЗМОЖНО **wx** **rx** **ry** **wy** **ry** **rx** **z++** **z = 1**

НЕВОЗМОЖНО **rx** **ry** **wy** **ry** **rx** **wx** **z = 0**

очевидно, что **wx** предшествует **rx**, а **wy** предшествует **ry**
в зависимости от того, что выполнится ранее – **wx** или **wy** операция **z++** обязательно выполнится хотя бы одним из потоков

Граф отношений предшествования



AQUIRE-RELEASE МОДЕЛЬ

Порядок работы с памятью

- На одном потоке порядок работы с памятью определяется выполнением операций чтения и записи и выражается отношением **«выполнилось ранее»**.
- Взаимный порядок выполнения операций на разных потоках определяется синхронизацией и определяется отношением **«синхронизовано с»**.

Отношение «выполнилось ранее»

Означает, что в одном потоке одна операция выполнялась до другой. Например, в следующем коде:

1. `a = 5;`
2. `b = a;`

операция 1 выполняется до 2. Здесь тоже:

1. `a = 5;`
2. `b = 10.`

А вот порядок вычисления аргументов функции при вызове не определен, поэтому между выражениями в аргументах не возникает отношения «sequenced before»:

`f(a = 10, b = 5);`

Отношение синхронизации

- Атомарная операция чтения R значения атомарной переменной (кроме **relaxed** и **consume**) синхронизируется с операцией W (кроме **relaxed**) записи этого значения на другом потоке, если прочитанное значение записано операцией W либо последующими операциями записи в том же потоке
- Операция блокировки мьютекса синхронизируется с операцией освобождения мьютекса.

Отношение синхронизации

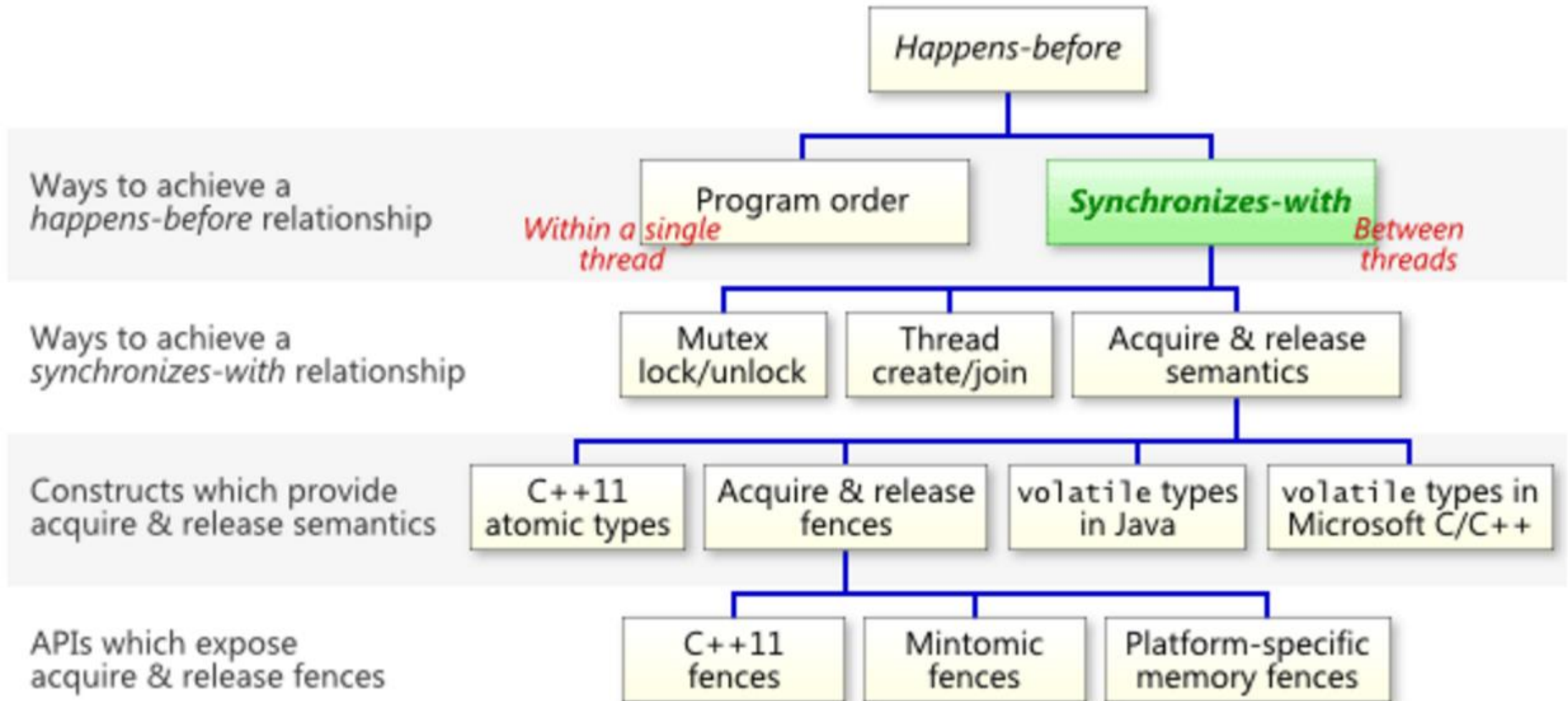
- Операция запуска потока, выполняемая родительским потоком, синхронизована с операциями созданного потока.
- Операция объединения (join) дочернего потока, выполняемая родительским потоком синхронизована с операциями, выполняемыми на дочернем потоке.
- Операция ожидания по условной переменной синхронизируется с операцией сигнала по ней.

Отношение предшествования «случилось до»

- A happens before B, если выполнение A предшествует B в одном потоке («выполнилось ранее»)
- A happens before B, если A «синхронизовано с B» (A – запись, B – чтение)
- $A \text{ h.b. } B \text{ и } B \text{ h.b. } C \Rightarrow A \text{ h.b. } C$

Выполнение программы должно соблюдать семантику отношения «случилось до». Операция видит все побочные эффекты, которые «случились до» нее.

Отношение «случилось до»



Для чего нужно отношение «случилось до»?

Отношение «случилось до» задает семантику выполнения программы. Если две операции упорядочены с помощью «случилось до», то побочный эффект первой операции «виден» второй операцией.

Для чего нужно отношение «случилось до»?

From C++ Standard: A **visible side effect** A on a scalar object or bit-field M with respect to a value computation B of M satisfies the conditions:

- A happens before B and
- there is no other side effect X to M such that A happens before X and X happens before B.

The value of a non-atomic scalar object or bit-field M, as determined by evaluation B, shall be the value stored by the visible side effect A.

Пример отношения предшествования

```
std::vector<int> data;  
std::atomic<bool> data_ready(false);
```

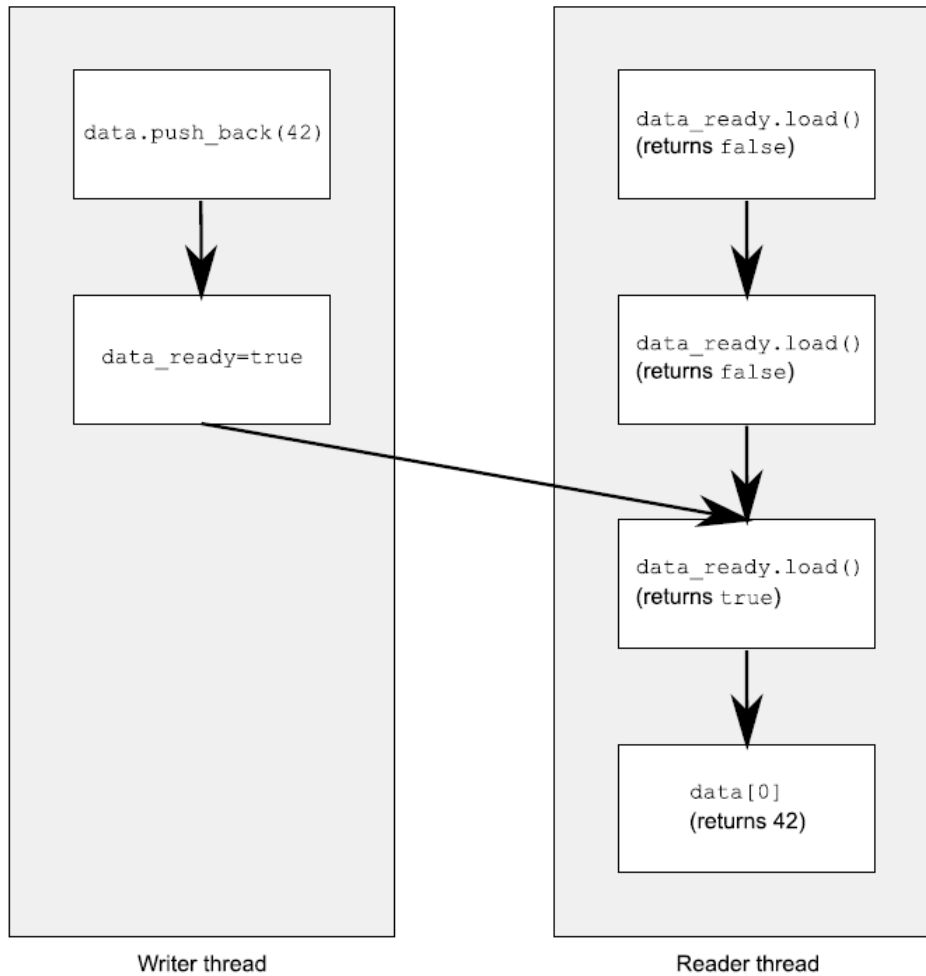
```
void reader_thread()  
{  
    while(!data_ready.load())  
    {  
        std::this_thread::sleep(std::milliseconds(1));  
    }  
    std::cout<<"The answer="<<data[0]<<"\n";  
}
```

```
void writer_thread()  
{  
    data.push_back(42);  
    data_ready=true;  
}
```

← 42

Отношение предшествования, инициируемое операциями над атомарными переменными гарантирует, что прочитано будет значение 42, в вектор data произойдет после

Граф отношения предшествования для данного примера



Семантика Acquire-Release

- **Acquire** всегда применяется только для операций чтения, либо чтения-записи-модификации
- **Release** всегда применяется для операций записи, либо для чтения-записи-модификации

Acquire semantics Операция загрузки с этим порядком памяти выполняет операцию получения в затронутом месте памяти: никакие операции чтения или записи в текущем потоке не могут быть переупорядочены до этой загрузки. Все записи в других потоках, которые освобождают ту же атомарную переменную, видны в текущем потоке.

Release semantics Операция записи данных с этим порядком памяти выполняет операцию освобождения: никакие операции чтения или записи в текущем потоке не могут быть переупорядочены после этой записи. Все записи в текущем потоке видны в других потоках, которые получают ту же атомарную переменную.

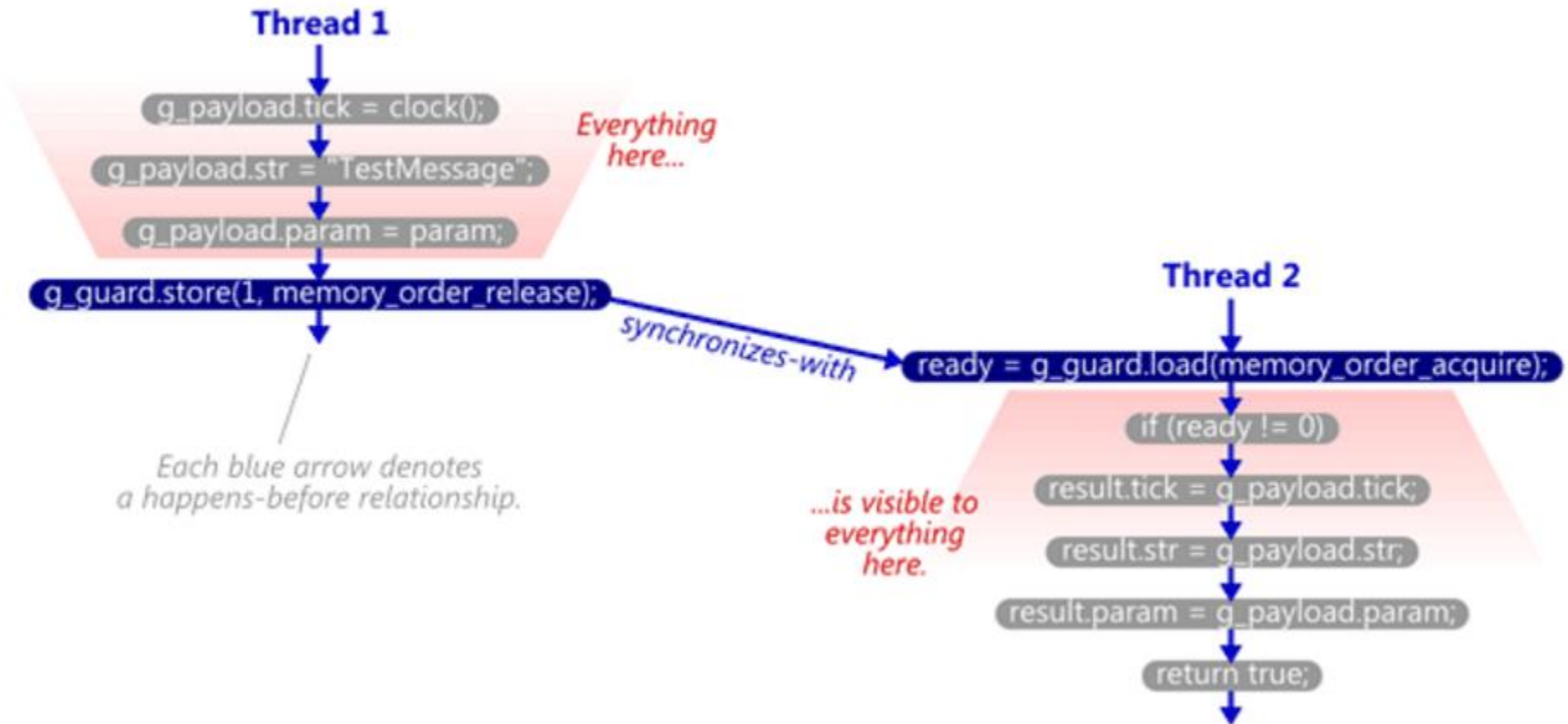
Семантика Acquire-Release

- **Acquire** не дает компилятору/ран-тайму переместить операции чтения и записи до нее
- **Release** не дает компилятору/ран-тайму переместить операции чтения и записи после нее
- Поток выполняющий **acquire** «видит» изменения, произошедшие с общими переменными до **release**

Использование acquire-release для «обмена» сообщениями

```
struct Message { clock_t tick; const char* str; void* param; };  
Message g_payload;  
std::atomic<int> g_guard(0);  
  
void SendTestMessage(void* param)  
{  
    // Copy to shared memory using non-atomic stores.  
    g_payload.tick = clock();  
    g_payload.str = "TestMessage";  
    g_payload.param = param;  
  
    // Perform an atomic write-release to indicate that the message is ready.  
    g_guard.store(1, std::memory_order_release);  
}
```

Иллюстрация acquire-release



Использование acquire-release для «обмена» сообщениями

```
std::atomic<int> g_guard(0);
```

```
void SendTestMessage(void* param)
```

```
{
```

```
    // Copy to shared memory using non-atomic stores.
```

```
    g_payload.tick = clock();
```

```
    g_payload.str = "TestMessage";
```

```
    g_payload.param = param;
```

```
    // Perform an atomic write-release to indicate that the message is ready.
```

```
    g_guard.store(1, std::memory_order_release);
```

```
}
```

Использование acquire-release для «обмена» сообщениями

```
bool TryReceiveMessage(Message& result)
{
    // Perform an atomic read-acquire to check whether the message is ready.
    int ready = g_guard.load(std::memory_order_acquire);

    if (ready != 0)
    {
        // Yes. Copy from shared memory using non-atomic loads.
        result.tick = g_payload.tick;
        result.str  = g_payload.str;
        result.param = g_payload.param;

        return true;
    }
    return false;
}
```

Пример использования Acquire-Release

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;
void write_x()
{
x.store(true,std::memory_order_release);
}
void write_y()
{
y.store(true,std::memory_order_release);
}
```

Пример ограничений Acquire-Release

```
void read_x_then_y()  
{  
  while(!x.load(std::memory_order_acquire));  
  if(y.load(std::memory_order_acquire))  
    ++z;  
}
```

```
void read_y_then_x()  
{  
  while(!y.load(std::memory_order_acquire));  
  if(x.load(std::memory_order_acquire))  
    ++z;  
}
```

Пример использования Acquire-Release

```
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);
}
```

**В данном
случае
срабатывание
assert не
гарантируется!**

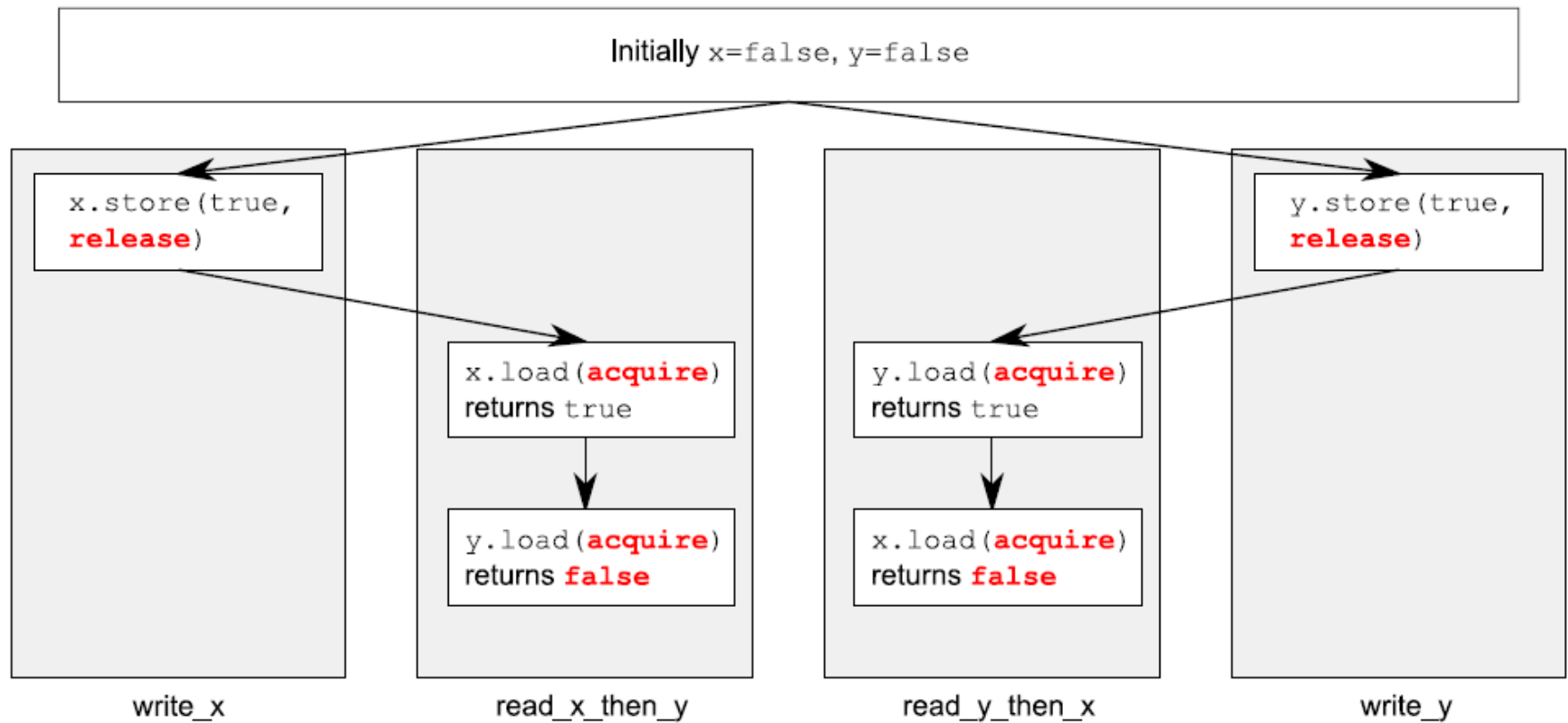


Figure 5.6 Acquire-release and happens-before

RELAXED MODEL

Relaxed Model

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);
    y.store(true,std::memory_order_relaxed);
}
```

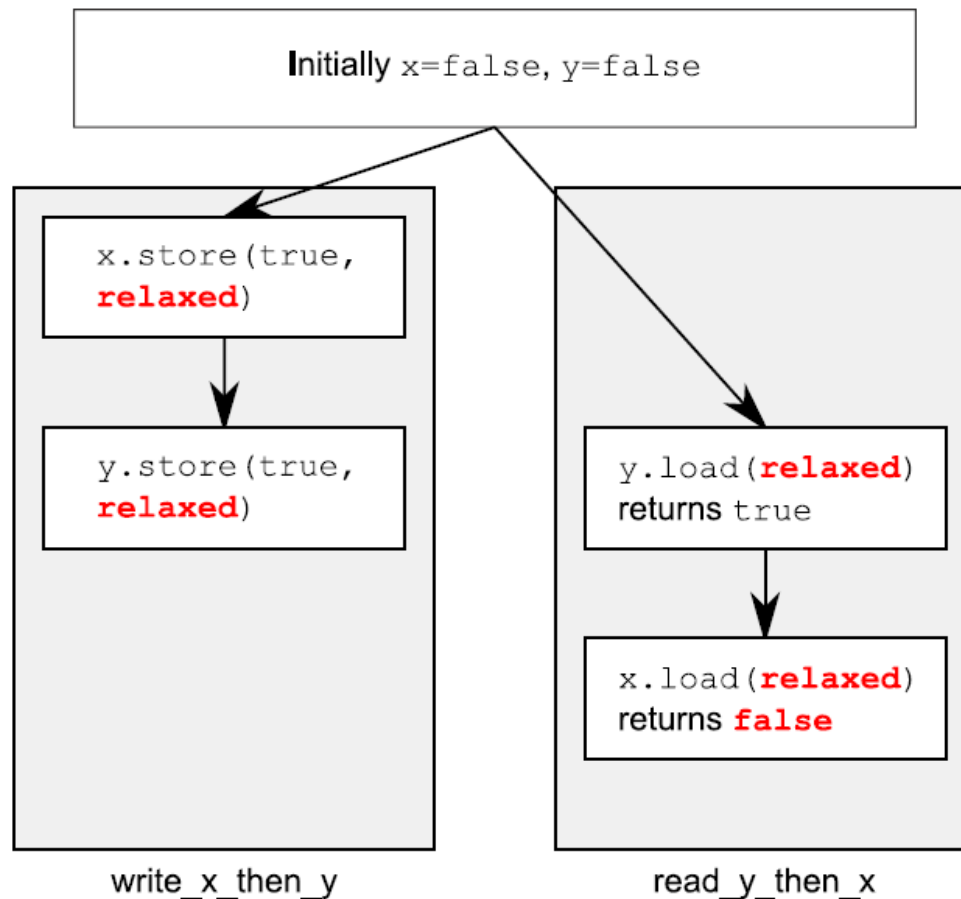

Relaxed Model

```
void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));
    if(x.load(std::memory_order_relaxed))
        ++z;
}

int main()
{
    x=false; y=false; z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);
}
```

Relaxed Model

Проверка может сработать, т.к. релаксационная модель не гарантирует упорядочивания.



Использование Relaxed Model

Relaxed model
полезно
использовать там,
где не важен
порядок изменения
данных, но требуется
атомарность.
Например –
счетчики.

```
1  // relaxed.cpp
2
3  #include <vector>
4  #include <iostream>
5  #include <thread>
6  #include <atomic>
7
8  std::atomic<int> cnt = {0};
9
10 void f()
11 {
12     for (int n = 0; n < 1000; ++n) {
13         cnt.fetch_add(1, std::memory_order_relaxed);
14     }
15 }
16
17 int main()
18 {
19     std::vector<std::thread> v;
20     for (int n = 0; n < 10; ++n) {
21         v.emplace_back(f);
22     }
23     for (auto& t : v) {
24         t.join();
25     }
26     std::cout << "Final counter value is " << cnt << '\n';
27 }
```

БАРЬЕРЫ ПАМЯТИ

Барьеры памяти

- Барьеры позволяют устраивать синхронизации без использования атомарных операций
- Барьеры порождают отношение «синхронизовано с»
- Ограничимся рассмотрением acquire-release барьеров, как наиболее употребительных

Барьеры памяти

```
#include <atomic>
#include <thread>
#include <assert.h>
std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_release);
    y.store(true,std::memory_order_relaxed);
}
```

Барьеры памяти

```
void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));
    std::atomic_thread_fence(std::memory_order_acquire);
    if(x.load(std::memory_order_relaxed))
        ++ z;
}
```

```
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);
}
```

Правило синхронизации барьеров

assert сработает, т.к. операция

```
y.store(true,std::memory_order_relaxed);
```

расположенная после release-барьера записала значение, прочитанное операцией

```
y.load(std::memory_order_relaxed),
```

расположенной до acquire-барьера, таким образом вызвав синхронизацию барьеров.

Барьеры памяти

```
#include <atomic>
#include <thread>
#include <assert.h>
```

```
bool x; // неатомарная
```

```
std::atomic<bool> y;
```

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

```
void write_x_then_y()
```

```
{
```

```
    x.store(true,std::memory_order_relaxed);
```

```
    std::atomic_thread_fence(std::memory_order_release);
```

```
    y.store(true,std::memory_order_relaxed);
```

```
}
```

Барьеры памяти

```
void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));
    std::atomic_thread_fence(std::memory_order_acquire);
    if(x.load(std::memory_order_relaxed))
        ++ z;
}
```

```
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);
}
```

Проверка сработает, т.к.
синхронизация барьеров
происходит (обеспечена
синхронизацией **y**)

Общее правило

Fence-fence synchronization

A release fence FA in thread A synchronizes-with an acquire fence FB in thread B, if

- There exists an atomic object M,
- There exists an atomic write X (with any memory order) that modifies M in thread A
- FA is sequenced-before X in thread A
- There exists an atomic read Y (with any memory order) in thread B
- Y reads the value written by X (or the value would be written by release sequence headed by X if X were a release operation)
- Y is sequenced-before FB in thread B

In this case, all non-atomic and relaxed atomic stores that happen-before FA in thread A will be synchronized-with all non-atomic and relaxed atomic loads from the same locations made in thread B after FB.

Синхронизация барьера и атомарного чтения

Fence-atomic synchronization

A release fence F in thread A synchronizes-with atomic acquire operation Y in thread B, if

- there exists an atomic store X (with any memory order)
- Y reads the value written by X (or the value would be written by release sequence headed by X if X were a release operation)
- F is sequenced-before X in thread A

In this case, all non-atomic and relaxed atomic stores that happen-before X in thread A will be synchronized-with all non-atomic and relaxed atomic loads from the same locations made in thread B after F.

Синхронизация атомарной записи и барьера

Atomic-fence synchronization

An atomic release operation X in thread A synchronizes-with an acquire fence F in thread B, if

- there exists an atomic read Y (with any memory order)
- Y reads the value written by X (or by the release sequence headed by X)
- Y is sequenced-before F in thread B

In this case, all non-atomic and relaxed atomic stores that happen-before X in thread A will be synchronized-with all non-atomic and relaxed atomic loads from the same locations made in thread B after F.

Использование барьеров для упорядочивания неатомарных операций с памятью

```
void SendTestMessage(void* param)
{
    g_payload.tick = clock();
    g_payload.str = "TestMessage";
    g_payload.param = param;

    std::atomic_thread_fence(std::memory_order_release);

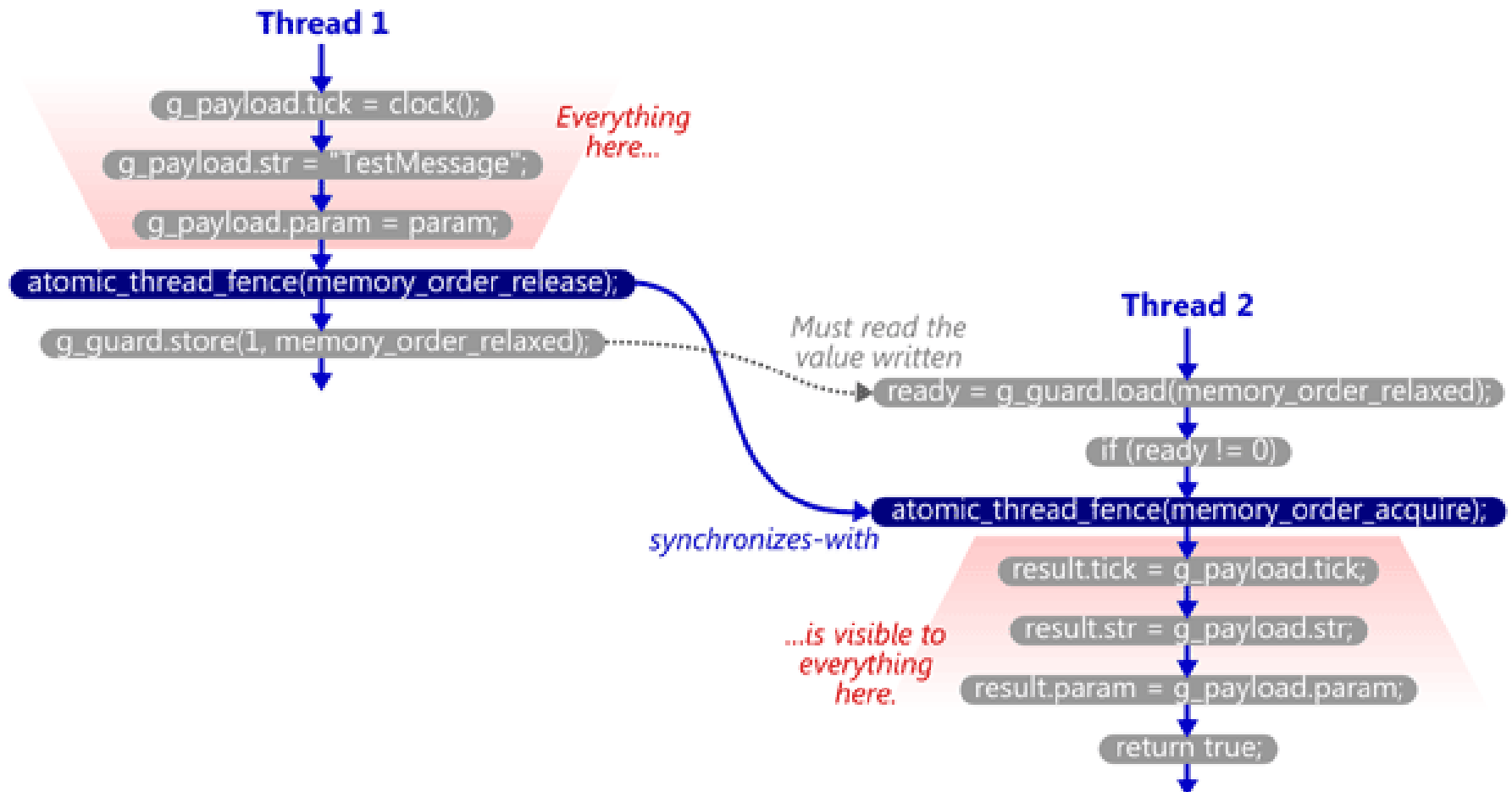
    g_guard.store(1, std::memory_order_relaxed);
}
```

Использование барьеров для упорядочивания неатомарных операций с памятью

```
bool TryReceiveMessage(Message& result)
{
    int ready = g_guard.load(std::memory_order_relaxed);
    if (ready != 0)
    {
        std::atomic_thread_fence(std::memory_order_acquire);
        result.tick = g_payload.tick;
        result.str = g_msg_str;
        result.param = g_payload.param;

        return true;
    }
    return false;
}
```

Использование барьеров для упорядочивания неатомарных операций с памятью



Реализация барьеров

- Барьеры приводят к определенным запретам переупорядочивания на этапе трансляции и исполнения программы
- Очевидно, что любые инструкции записи не могут быть перемещены после release-барьера, а любые инструкции чтения – до acquire

Реализация барьеров

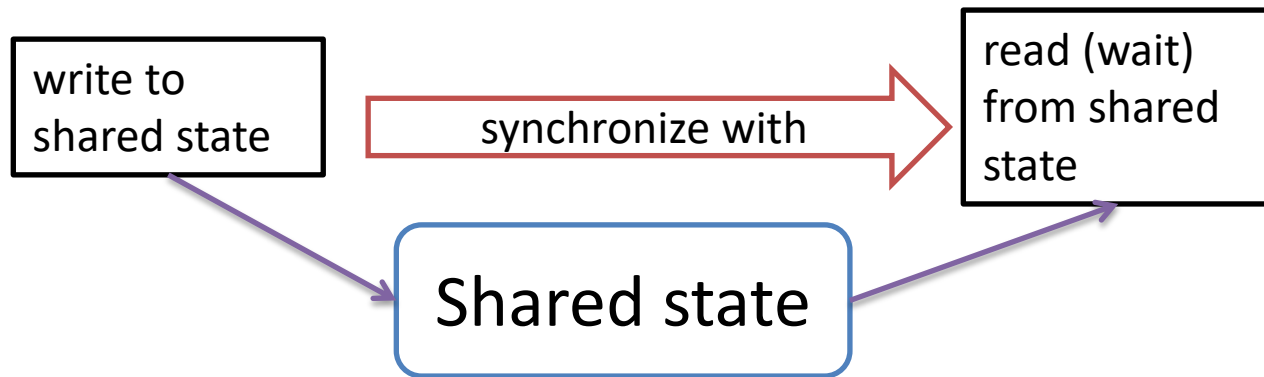
- Также очевидно, что (по причине их использования для определения отношения предшествования) операции чтения атомарных объектов не могут быть перенесены после acquire-барьеров, а операции атомарной записи — до release-барьеров.
- Для понимания работы программы необходимо использовать сведения из стандарта, не делая предположения относительно дополнительных свойств барьеров по отношению к упорядочиванию инструкций.

**РАБОТА С ОБЩИМ СОСТОЯНИЕМ:
FUTURE, ПАКЕТНЫЕ ЗАДАНИЯ,
PROMISE**

Разделяемое состояние

Разделяемое состояние (shared state) – это способ взаимодействия потоков. Оно обеспечивает передачу данных (возвращаемых значений и исключений) и синхронизацию между потоками.

Синхронизация с разделяемым состоянием



From Standard: Calls to functions that successfully set the stored result of a shared state **synchronize with** calls to functions successfully detecting the ready state resulting from that setting. The storage of the result (whether normal or exceptional) into the shared state synchronizes with the successful return from a call to a waiting function on the shared state.

std::future

- Основной инструмент для работы с результатом, возвращаемым функцией потока
- Позволяет идентифицировать событие, которое должно произойти в будущем и его результат
- Параметр шаблона T указывает на тип результата события `std::future<T>`
- `std::future<void>` означает что событие не связано с конкретным типом данных

Конструкторы `std::future`

- **`future()`** - пустой конструктор, обычно не используется напрямую;
- **`future(future&& other)`** - конструктор перемещения;
- **`future(const future& other) = delete`** - копирующий конструктор не поддерживается.

Присваивание std::future

- **future& operator=(future&& other);** - перемещение объекта;
- **future& operator=(const future& other) = delete;** - обычное копирование запрещено.

Основные методы `std::future`

- `void wait() const;` - блокирует поток до момента, когда результат `std::future` готов
- `T get();` - возвращает значение результата;
- `T& get();` - ссылка на результат (для `std::future<T&>`;
- `void get()` – только для `std::future<void>`

Основные методы `std::future`

- `template< class Rep, class Period > std::future_status
wait_for(const std::chrono::duration<Rep,Period>&
timeout_duration) const;`

Ожидает получения результата не более заданного времени;

Возвращаемое значение:

future_status::deferred функция, которая должна вернуть результат, еще не стартовала;

future_status::ready результат готов;

future_status::timeout закончился период ожидания.

Основные методы `std::future`

- `template< class Clock, class Duration > std::future_status
wait_until(const std::chrono::time_point<Clock,Duration>&
timeout_time) const;`

Ожидает получения результата не позднее времени;

Возвращаемое значение:

future_status::deferred функция, которая должна
вернуть результат, еще не стартовала;

future_status::ready результат готов;

future_status::timeout закончился период ожидания.

Асинхронные операции

- **std::async** позволяет инициировать выполнение операции, результат которой может быть получен позднее
- Операцией является объект или функция (Callable object)
- Результатом является объект типа **std::future**

Асинхронные операции

```
enum class launch  
{  
    async, deferred  
};
```

```
template<typename Callable,typename ... Args>  
future<result_of<Callable(Args...)>::type>  
async(Callable&& func,Args&& ... args);
```

```
template<typename Callable,typename ... Args>  
future<result_of<Callable(Args...)>::type>  
async(launch policy, Callable&& func,Args&& ... args);
```



Пример использования

```
#include <future>
#include <iostream>

int find_the_answer_to_ltuae();

void do_other_stuff();

int main()
{
    std::future<int> the_answer=std::async(find_the_answer_to_ltuae);

    do_other_stuff();

    std::cout<<"The answer is "<<the_answer.get()<<std::endl;
}
```

Пример использования

```
#include <thread>
#include <future>
#include <iostream>
#include <functional>

class A {
public:
    A(int a) : mA(a) { }

    int operator () (int b) const { return mA + b; }

    int mul(int b) const { return mA * b; }

    void update(int v) { mA = v; }

private:
    int mA;
};
```

Пример использования

```
int main() {  
    A a(5);  
    std::future<int> fut = std::async(a, 4);  
    std::cout << "sum = " << fut.get() << "\n";  
    std::future<int> fut1 = std::async(&A::mul, a, 4);  
    std::cout << "mul = " << fut1.get() << "\n";  
  
    std::future<void> fut2 = std::async(&A::update, a, 10);  
    fut2.wait();  
    std::cout << "new sum = " << a(20) << "\n";  
  
    std::future<void> fut3 = std::async(&A::update, &a, 10);  
    fut3.wait();  
    std::cout << "new sum = " << a(20) << "\n";  
  
    std::future<int> fut4 = std::async([](int a, int b) { return a + b;}, 10, 100);  
    std::cout << "10 + 100 = " << fut4.get() << "\n";  
    return 0;  
}
```


Пример использования

```
posypkin@mikhail:~/exp/sandbox/c11/async$ ./async1  
sum = 9  
mul = 20  
new sum = 25  
new sum = 30  
10 + 100 = 110
```

Thread Oversubscription

Oversubscription - ситуация, когда число потоков превышает число физически доступных ядер.

Небольшое превышение может позитивно сказаться на производительности: когда один поток ожидает на операциях с памятью, другой выполняет вычисления. Это позволяет задействовать внутрипроцессорный параллелизм и, тем самым, ускорить выполнение программы.

Негативные последствия Oversubscription: переключение контекста

Переключение контекста потока приводит к непродуктивным накладным расходам. Поэтому если число потоков намного превышает число ядер, то переключение контекста может существенно снизить производительность программы.

Негативные последствия Oversubscription: cache cooling

Разные потоки, как правило, работают с разными данными. Поэтому при переключении ядра между потоками из кэш-памяти вытесняются данные, с которыми работал поток. Когда поток опять ставится на выполнение, увеличивается время на доступ к данным, т.к. требуется их загрузка в кэш из основной памяти.

std::thread vs std::async

std::thread

- + Больше контроля за выполнением потоков («ручное управление»)
- + Доступ к низкоуровневому API (pthreads)
- Проблемы балансировки и перегрузки системы (oversubscription)
- Сложный доступ к результату выполнения

std::async

- + больше свободы системе в плане балансировки нагрузки (система решает запускать новый поток или нет)
- + легкий доступ к результату через std::future
- отсутствие низкоуровневого контроля

Управление дисциплиной вызова

```
auto f6=std::async(std::launch::async,Y(),1.2);  
auto f7=std::async(std::launch::deferred,baz,std::ref(x));  
auto f8=std::async(  
    std::launch::deferred | std::launch::async,  
    baz,std::ref(x));  
auto f9=std::async(baz,std::ref(x));  
f7.wait();
```

← **Run in new thread**

← **Run in wait() or get()**

← **Implementation chooses**

← **Invoke deferred function**

В некоторых случаях, тем не менее, необходимо обеспечить, чтобы асинхронная операция обязательно выполнялась в рамках отдельного потока. Тогда используется флаг `std::launch::async`.

По умолчанию

При создании потока через `std::thread` исключения не передаются

```
class E {
public:
    E(const std::string& name): mName(name){
    }
    std::string name() const {
        return mName;
    }

private:
    std::string mName;
};

void f() {
    std::cout << "Hi from thread\n";
    throw E("function f");
}
```

```
int main()
{
    std::thread th;
    try {
        th = std::thread(f);
    } catch(E excpt) {
        std::cout << "in constructor " <<
            excpt.name() << std::endl;
    }

    try {
        th.join();
    } catch(E excpt) {
        std::cout << "in join " <<
            excpt.name() << std::endl;
    }
}
```

Результат:

Hi from thread

Передача исключений через разделяемое состояние

```
#include <iostream>
#include <string>
#include <thread>
#include <future>
```

Исключения также могут передаваться через **shared state**

```
class E {
public:
    E(const std::string& name): mName(name){
    }
    std::string name() const {
        return mName;
    }

private:
    std::string mName;
};
```



```
void f() {  
    throw E("function f");  
}  
  
int main()  
{  
    auto fut = std::async(f);  
  
    try {  
        fut.get();  
    } catch(E excpt) {  
        std::cout << excpt.name() << std::endl;  
    }  
}
```

Результат:

function f

std::packaged_task

```
template< class R, class ...Args >  
class packaged_task<R(Args...)>;
```

Связывает сущность типа Callable с аргументами, которая может быть потом вызвана посредством оператора ():

```
void operator()( ArgTypes... args );
```

Результат типа R возвращается через фьючерс с помощью метода

```
std::future<R> get_future();
```

Задания `packaged_task`

- Могут передаваться между потоками, тем самым облегчая создание различных схем планирования заданий
- **`std::packaged_task<T(...)>`** связывает вызываемый объект (Callable) с **`std::future<T>`**
- Шаблонный параметр представляет собой сигнатуру функции, из которой выводится тип результата
- Могут сами использоваться как аргументы для конструктора потока (в силу возможности вызова)

Использование packaged_task с lambda-выражениями

```
#include <iostream>
#include <cmath>
#include <thread>
#include <future>
#include <functional>

void task_lambda()
{
    std::packaged_task<int(int,int)> task([](int a, int b) {
        return std::pow(a, b);
    });
    std::future<int> result = task.get_future();
    task(2, 9);
    std::cout << "task_lambda:\t" << result.get() << '\n';
}
```

Использование `packaged_task` с функциональными объектами

```
int f(int x, int y) { return std::pow(x,y); }
```

```
void task_bind()  
{  
    std::packaged_task<int()> task(std::bind(f, 2, 11));  
    std::future<int> result = task.get_future();  
  
    task();  
  
    std::cout << "task_bind:\t" << result.get() << '\n';  
}
```

Передача задания в поток

```
void task_thread()  
{  
    std::packaged_task<int(int,int)> task(f);  
    std::future<int> result = task.get_future();  
  
    std::thread task_td(std::move(task), 2, 10);  
    task_td.join();  
  
    std::cout << "task_thread:\t" << result.get() << '\n';  
}
```

Выполнение и результат

```
int main()
{
    task_lambda();
    task_bind();
    task_thread();
}
```

```
task_lambda: 512
task_bind: 2048
task_thread: 1024
```

Передача исключения через задание

```
class E {  
    public:  
        E(const std::string& name): mName(name){  
        }  
        std::string name() const {  
            return mName;  
        }  
  
    private:  
        std::string mName;  
};
```



```
std::ostream& operator<< (std::ostream& o, const E & e)
{
    return o << e.name();
}
```

```
int f(int x, int y) {
    throw E(std::string("except ") + " " + std::to_string(x) + " " +
            std::to_string(y));
    return std::pow(x,y);
}
```

Запуск с функциональным объектом

```
void task_bind()
{
    std::packaged_task<int()> task(std::bind(f, 2, 11));
    std::future<int> result = task.get_future();

    task();
    try{
        std::cout << "task_bind:\t" << result.get() << std::endl;
    } catch (E e){
        std::cout << "got exception " << e << std::endl;
    }
}
```

Использование в потоке

```
void task_thread()  
{  
    std::packaged_task<int(int,int)> task(f);  
    std::future<int> result = task.get_future();  
  
    std::thread task_td(std::move(task), 2, 10);  
    task_td.join();  
  
    try{  
        std::cout << "task_bind:\t" << result.get() << std::endl;  
    } catch (E e){  
        std::cout << "got exception " << e;  
    }  
}
```

Выполнение и результат

```
int main()
{
    task_bind();
    task_thread();
}
```

```
got exception except 2 11
got exception except 2 10
```

Возможное применение: очередь заданий

```
#include <deque>
#include <mutex>
#include <future>
#include <thread>
#include <utility>
std::mutex m;
std::deque<std::packaged_task<void()> > tasks;
bool gui_shutdown_message_received();
void get_and_process_gui_message();
```

```
void gui_thread()
{
    while(!gui_shutdown_message_received())
    {
        get_and_process_gui_message();
        std::packaged_task<void()> task;
        {
            std::lock_guard<std::mutex> lk(m);
            if(tasks.empty())
                continue;
            task=std::move(tasks.front());
            tasks.pop_front();
        }
        task();
    }
}
```

```
template<typename Func>
std::future<void> post_task_for_gui_thread(Func f)
{
    std::packaged_task<void()> task(f);
    std::future<void> res=task.get_future();
    std::lock_guard<std::mutex> lk(m);
    tasks.push_back(std::move(task));
    return res;
}
```

std::promise

Позволяет явным образом сохранять в разделяемый объект (shared state) значение или исключение, которое в дальнейшем может быть получено с помощью std::future.

Конструкторы `std::promise`

- Конструкторы
 - `promise()`; создает объект с пустым разделяемым состоянием;
 - `template< class Alloc > promise(std::allocator_arg_t, const Alloc& alloc);` создает объект с помощью заданного аллокатора;
 - `promise(promise&& other);` - конструктор перемещения;
 - `promise(const promise& other) = delete;` конструктор копирования запрещен.

Методы std::promise

- **std::future<T> get_future();** - возвращает фьючерс, соответствующий изменяемому состоянию
- установка значения:
void set_value(const R& value);
void set_value(R&& value);
void set_value(R& value); - для promise<R>
void set_value(); - для promise<void>
может вызываться только один раз (при повторном вызове возникает исключение)
- **установка значения по выходу из потока:**
void set_value_at_thread_exit(const R& value);
void set_value_at_thread_exit(R&& value);
void set_value_at_thread_exit(R& value);
void set_value_at_thread_exit();

Пример использования std::promise

```
#include <vector>
#include <thread>
#include <future>
#include <numeric>
#include <iostream>
#include <chrono>

void accumulate(std::vector<int>::iterator first,
               std::vector<int>::iterator last,
               std::promise<int> accumulate_promise)
{
    int sum = std::accumulate(first, last, 0);
    accumulate_promise.set_value(sum); // Notify future
}
```

Пример использования std::promise

```
int main()
{
    std::vector<int> numbers = { 1, 2, 3, 4, 5, 6 };
    std::promise<int> accumulate_promise;
    std::future<int> accumulate_future = accumulate_promise.get_future();
    std::thread work_thread(accumulate, numbers.begin(), numbers.end(),
                           std::move(accumulate_promise));
    accumulate_future.wait();
    std::cout << "result=" << accumulate_future.get() << '\n';
    work_thread.join();
}
```

Второй пример использования `std::promise::set_value`

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

int main()
{
    std::promise<int> p;
    std::future<int> f = p.get_future();
    std::chrono::time_point<std::chrono::system_clock> start, end;
    start = std::chrono::system_clock::now();
    std::thread([&p] {
        p.set_value(9);
        std::this_thread::sleep_for(std::chrono::seconds(3));
    }).detach();
```

Второй пример использования `std::promise::set_value`

```
std::cout << "Waiting..." << std::flush;
std::cout << "Done!\nResult is: " << f.get() << '\n';
end = std::chrono::system_clock::now();
std::cout << "Elapsed " << std::chrono::duration_cast<std::chrono::seconds>
(end-start).count() << " seconds\n";
}
```

```
posypkin@mikhail:~/exp/sandbox/c11/promise$ ./promise1
Waiting...Done!
Result is: 9
Elapsed 0 seconds
```

Пример использования set_value_at_thread_exit

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

int main()
{
    std::promise<int> p;
    std::future<int> f = p.get_future();
    std::chrono::time_point<std::chrono::system_clock> start, end;
    start = std::chrono::system_clock::now();
    std::thread([&p] {
        p.set_value_at_thread_exit(9);
        std::this_thread::sleep_for(std::chrono::seconds(3));
    }).detach();
```

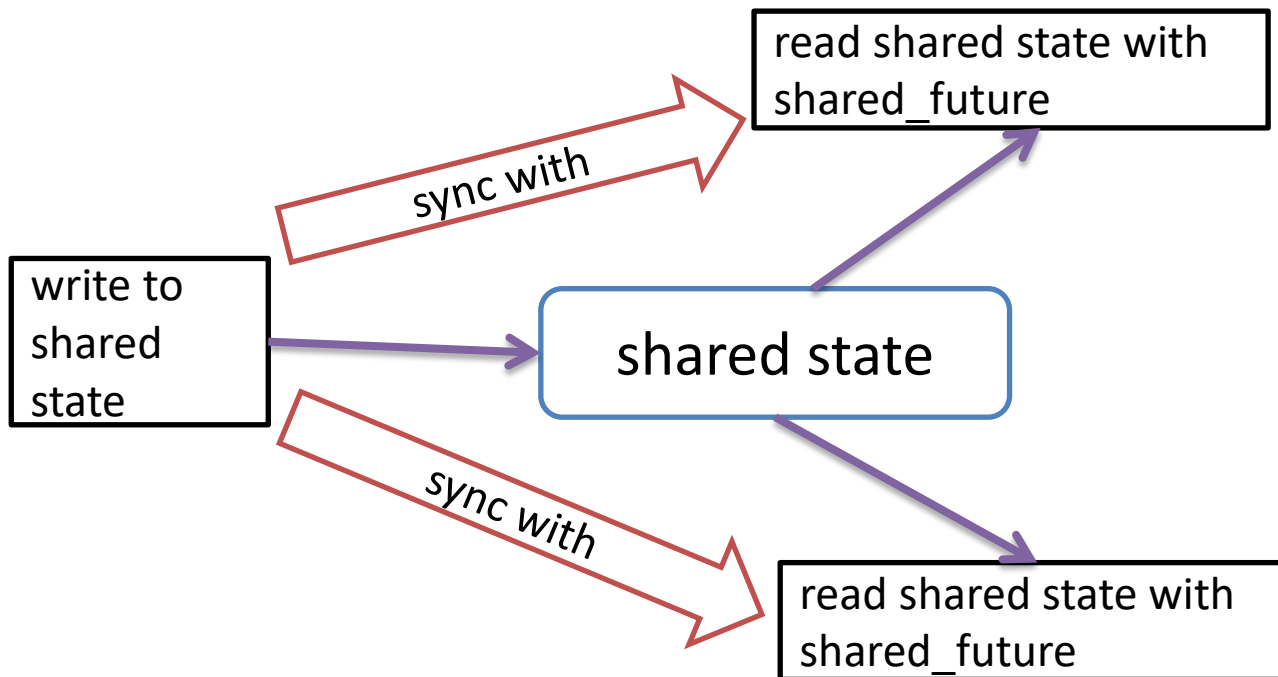
Пример использования set_value_at_thread_exit

```
std::cout << "Waiting..." << std::flush;  
std::cout << "Done!\nResult is: " << f.get() << '\n';  
end = std::chrono::system_clock::now();  
std::cout << "Elapsed " << std::chrono::duration_cast<std::chrono::seconds>  
(end-start).count() << " seconds\n";  
}
```

```
posypkin@mikhail:~/exp/sandbox/c11/promise$ ./promise1  
Waiting...Done!  
Result is: 9  
Elapsed 3 seconds
```


shared_future

`std::shared_future` позволяет нескольким потокам ожидать изменения ожидаемого состояния, в отличие от `future` – допускает копирование. Индуцирует отношение «синхронизовано с».



Конструкторы shared_future

Создает несвязанный с разделяемым состоянием объект
`shared_future()` noexcept;

Конструктор копирования
`shared_future(const shared_future& other)` noexcept;

Конструктор перемещения
`shared_future(shared_future&& other)` noexcept;

Создает shared_future по «обычному» future
`shared_future(std::future<T>&& other)` noexcept;

Методы shared_future

valid

checks if the future has a shared state

(public member function)

wait

waits for the result to become available

(public member function)

wait_for

waits for the result, returns if it is not available for the specified timeout duration

(public member function)

wait_until

waits for the result, returns if it is not available until specified time point has been reached

(public member function)

Пример метода valid()

```
#include <iostream>
#include <string>
#include <thread>
#include <future>

int main()
{
    std::future<void> empty_future;
    std::cout << "empty_future is " << (empty_future.valid() ? "valid" : "invalid") <<
std::endl;

    std::shared_future<void> empty_shared_future;
    std::cout << "empty_shared_future is " << (empty_shared_future.valid() ?
"valid" : "invalid") << std::endl;
}
```

Результат:

empty_future is invalid

empty_shared_future is invalid

Пример использования shared_future

```
#include <iostream>
#include <string>
#include <thread>
#include <future>

int main()
{
    std::promise<void> ready_promise, t1_ready_promise, t2_ready_promise;
    std::shared_future<void> ready_future(ready_promise.get_future());

    std::chrono::time_point<std::chrono::high_resolution_clock> start;

    auto fun1 = [&, ready_future]() -> std::chrono::duration<double, std::milli>
    {
        t1_ready_promise.set_value();
        ready_future.wait(); // waits for the signal from main()
        return std::chrono::high_resolution_clock::now() - start;
    };
};
```

Пример использования shared_future

```
auto fun2 = [&, ready_future]() -> std::chrono::duration<double, std::milli>
{
    t2_ready_promise.set_value();
    ready_future.wait(); // waits for the signal from main()
    return std::chrono::high_resolution_clock::now() - start;
};
```

```
auto fut1 = t1_ready_promise.get_future();
auto fut2 = t2_ready_promise.get_future();
```

```
auto result1 = std::async(std::launch::async, fun1);
auto result2 = std::async(std::launch::async, fun2);
```

```
// wait for the threads to become ready
fut1.wait();
fut2.wait();
```

Пример использования shared_future

```
// the threads are ready, start the clock
start = std::chrono::high_resolution_clock::now();

std::cout << "=> ";
char c;
std::cin >> c;
// signal the threads to go
ready_promise.set_value();

std::cout << "Thread 1 received the signal "
    << result1.get().count() << " ms after start\n"
    << "Thread 2 received the signal "
    << result2.get().count() << " ms after start\n";
}
```

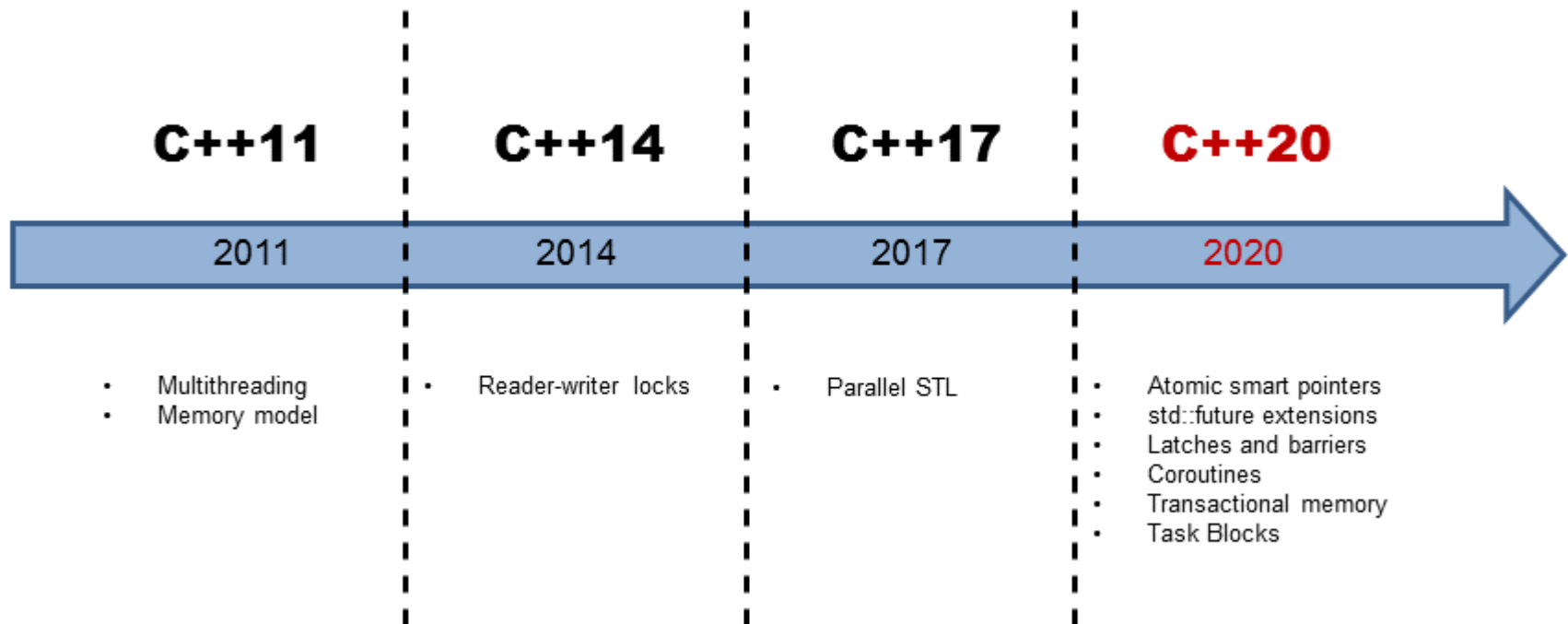
Результат:

input m => m

Thread 1 received the signal 1204.84 ms after start

Thread 2 received the signal 1204.9 ms after start

Современные стандарты C++



Параллельные алгоритмы

В C++ 17 появилась возможность указывать «политику выполнения» для некоторых алгоритмов стандартной библиотеки.

Пример:

```
std::vector<int> my_data;  
std::sort(std::execution::par, my_data.begin(), my_data.end());
```

При этом библиотека может (но не обязана) применить многопоточный вариант исполнения.

Виды политик выполнения

<code>std::execution::seq</code>	Последовательная политика (C++ 17)
<code>std::execution::par</code>	Параллельная упорядоченная политика: гарантируется то, что операции (вызовы пользовательских функций внутри алгоритмов) выполняются до конца на одном потоке и не прерываются. (C++ 17)
<code>std::execution::par_unseq</code>	Параллельная неупорядоченная политика. Операции (вызовы пользовательских функций внутри алгоритмов) могут прерываться и мигрировать с потока на поток. Допускается комбинирование распараллеливания и векторизации. (C++ 17)
<code>std::execution::unseq</code>	Допускается векторизация (C++ 20)

Правильное использование

1. `std::for_each(std::execution::par, v.begin(), v.end(), [](auto& x){++x;});`

правильно? да/нет

2. `std::for_each(std::execution::par, v.begin(), v.end(), [&](int& x){ x=++count; });`

правильно? да/нет

Правильное использование

1. `std::for_each(std::execution::par, v.begin(), v.end(), [](auto& x){++x;});`

правильно? **да**/нет

2. `std::for_each(std::execution::par, v.begin() ,v.end(), [&](int& x){ x=++count; });`

правильно? если count – атомарная, то правильно, в противном случае - нет

Варианты параллельных политик выполнения

Правильно:

```
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
    std::lock_guard<std::mutex> guard(m);
    ++x; // correct
});
```

Неправильно:

```
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par_unseq, std::begin(a), std::end(a), [&](int) {
    std::lock_guard<std::mutex> guard(m); // Error: lock_guard constructor calls
    m.lock()
    5. ++x;
});
```

Варианты параллельных политик выполнения

1. `std::for_each(std::execution::par_unseq, v.begin(), v.end(),
[&](auto& x){std::lock_guard<std::mutex> lg(lock); ++x;});`

правильно? **да**/нет

2. `std::for_each(std::execution::par_unseq, v.begin() ,v.end(),
[&](auto& x){std::lock_guard<std::mutex> lg(lock); ++x;});`

правильно? да/**нет** (т.к. не гарантируется непрерывность операций)

Барьеры C++ 20

- `latch` – однократный барьер со счетчиком
- `barrier` – обычный барьер

std::future extensions

- **then** – предшественник «ГОТОВ»
- **when_any** – один из предшественников «ГОТОВ»
- **when_all** – все предшественники ГОТОВЫ

futures then

```
future<int> f1= async([]() {return 123;});  
future<string> f2 = f1.then([](future<int> f) {  
    return f.get().to_string();  
});
```

После готовности f2 вызывается функция,
переданная в качестве аргумента then

futures when_any

```
future<int> futures[] = {async([]() { return intResult(125); }),  
                        async([]() { return intResult(456); })};
```

```
future<vector<future<int>>> any_f =  
    when_any(begin(futures),end(futures));
```

создает future, отвечающий состоянию «готового» future
из списка

when_any

```
template < class Sequence >
struct when_any_result {
    std::size_t index; // индекс готового future
    Sequence futures; // готовый future
};

template < class InputIt >
auto when_any(InputIt first, InputIt last)
    -> future<when_any_result<std::vector<typename
std::iterator_traits<InputIt>::value_type>>>>;
```

futures when_all

```
future<int> futures[] = {async([]() { return intResult(125); }),  
                        async([]() { return intResult(456); })};
```

```
future<vector<future<int>>> all_f = when_all(begin(futures),  
end(futures));
```

Возвращает future, который становится «готовым» когда готовы все аргументы. Future ассоциирован с вектором futures:

```
template < class InputIt >  
auto when_all(InputIt first, InputIt last)  
    -> future<std::vector<typename  
std::iterator_traits<InputIt>::value_type>>;
```

Latches

```
void doWork(threadpool* pool){
    latch completion_latch(NUMBER_TASKS);
    for (int i = 0; i < NUMBER_TASKS; ++i){
        pool->add_task([&]{
            // perform the work
            ...
            completion_latch.count_down();
        });
    }
    // block until all tasks are done
    completion_latch.wait();
}
```

Barriers

```
explicit barrier( std::ptrdiff_t num_threads );
```

`arrive_and_wait` – вход в барьерную синхронизацию и ожидание барьера

`arrive_and_drop` – вход в барьерную синхронизацию и удаление потока из списка ожидающих (продолжает выполнение)

СПАСИБО ЗА ВНИМАНИЕ!
ВОПРОСЫ?



Умножение матриц

```
#include <stdlib.h>
#include <iostream>
#include <thread>
#include <vector>
#include <chrono>

// Number of matrix rows and
// columns
int n;

// Number of threads
int nt;

double *a, *b, *c;

// Thread-friendly matrix multiply
// tn - thread number
void mmul(int tn) {
    for(int i = tn; i < n; i += nt) {
        for(int j = 0; j < n; j++) {
            double v = 0;
            for(int k = 0; k < n; k++)
                v += a[i*n + k] * b[k*n + j];
            c[i*n + j] = v;
        }
    }
}
```




```
// Fill in matrix
void init_matr()
{
    a = new double[n * n];
    b = new double[n * n];
    c = new double[n * n];
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            a[i*n + j] = 1;
            b[i*n + j] = 2;
        }
    }
}
```

```
// Check result
void check() {
    int check_step = n / 8;
    for(int i = 0; i < n; i +=
check_step) {
        for(int j = 0; j < n; j +=
check_step) {
            if(c[i*n + j] != 2 * n) {
                std::cerr << "Error
detected\n";
                exit(-1);
            }
        }
    }
}
```

```

main(int argc, char* argv[]) {
    if(argc != 3) {
        std::cerr << "usage: mmul matrix_dim
num_of_thread\n";
        exit(-1);
    }
    n = atoi(argv[1]);
    nt = atoi(argv[2]);
    init_matr();
    std::vector< std::thread > thv;

```

```

std::chrono::time_point<std::chrono::sy
stem_clock> start, end;
    start =
std::chrono::system_clock::now();
    for(int i = 0; i < nt; i++) {
        std::thread th(mmul, i);
        thv.push_back(std::move(th));
    }
    for(auto& t : thv) {
        t.join();
    }
    end =
std::chrono::system_clock::now();
    std::chrono::duration<double> dur =
end - start;
    std::cout << "Matrix multiply took " <<
dur.count() << "\n";
    check();
}

```

Пример специализации packaged_task

```
template<>
class packaged_task<std::string(std::vector<char>*,int)>
{
public:
    template<typename Callable>
    explicit packaged_task(Callable&& f);
    std::future<std::string> get_future();
    void operator()(std::vector<char>*,int);
};
```