

Введение в параллельные вычисления

**Лекция 2. C++11 threads.
Блокировки, мьютексы**

КС-40, КС-44
РХТУ

Преподаватель
Митричев Иван Игоревич, к.т.н.

2017

В предыдущих сериях...

Наука параллельных вычислений (parallel computing).

Топ-500. Закон Амдала. Закон Мура.

Технологии программирования для систем с общей памятью

- C++11 threads
- OpenMP

Технологии программирования для систем с распределенной памятью

- MPI

Технологии программирования для гетерогенных систем

- CUDA

Работа с потоками на Linux

#include <thread>

Eclipse:

Установить опции --std=c++11 -pthread (project->properties>c++ build>settings> GCC C++ Compiler > command)

Установить опцию -pthread (project>properties>c++ build>settings> GCC C++ Linker)

Компилятор g++

Опция -g – отладка

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

```
./code.out
```

```
gdb code.out
```

```
b main.cpp:10, r, info threads, thread 5...
```

Онлайн-компиляция

Пример <http://coliru.stacked-crooked.com/>

The screenshot shows the Coliru online compiler interface. At the top, there is a navigation bar with a "Donate" link and the Coliru logo. Below the navigation bar, the C++ code is displayed with line numbers from 1 to 15. The code defines a `main` function that creates a `std::thread` object `threadObj` and starts a thread that prints "Display Thread Executing" 10,000 times. The main thread then prints "Display From Main Thread" 10,000 times, joins the child thread, and prints "Exiting from Main Thread".

```
1  #include <thread>
2  int main()
3  {
4  std::thread threadObj( [] {
5      for(int i = 0; i < 10000; i++)
6          std::cout<<"Display Thread Executing"<<std::endl;
7      }); // объявление лямбды без параметров
8
9      for(int i = 0; i < 10000; i++)
10         std::cout<<"Display From Main Thread"<<std::endl;
11
12     threadObj.join();
13     std::cout<<"Exiting from Main Thread"<<std::endl;
14     return 0;
15 }
```

At the bottom of the interface, the compilation output is shown. It indicates that the code failed to compile due to an error in the lambda function. The error message is: "main.cpp:6:22: error: 'cout' is not a member of 'std'". The corresponding line of code is highlighted in the output area.

```
main.cpp: In lambda function:
main.cpp:6:22: error: 'cout' is not a member of 'std'
                   std::cout<<"Display Thread Executing"<<std::endl;
```

Первая программа с потоками (C++11)

```
// compile with
// g++ -std=c++11 -pthread
file_name.cpp

#include <iostream>
#include <thread>

void function_to_call(int _id)
{
    std::cout << "Launched by
thread " << _id << std::endl;
}

int main()
{
    const int num_threads = 10;
    std::thread
threads[num_threads];
    for (int i = 0; i <
num_threads; ++i)
    {
```

```
        threads[i] =
std::thread(function_to_call,
i);
    }

    for (int i = 0; i < num_threads;
++i)
    {
        threads[i].join();
    }
    return 0;
}
```

**Порядок вывода
произволен!**

Передача данных в многопоточную функцию (значение)

```
#include <thread>
#include <iostream>
void threadCallback(int x)
{
    std::cout<<"Inside Thread x =
"<<x<<std::endl;
}
int main()
{
    int x = 9;
    std::cout<<"In Main Thread :
Before Thread Start x =
"<<x<<std::endl;
    std::thread
threadObj(threadCallback,x);
    threadObj.join();
```

```
std::cout<<"In Main Thread :
After Thread Joins x =
"<<x<<std::endl;
    return 0;
}
```

In Main Thread : Before Thread Start x = 9
Inside Thread x = 10
In Main Thread : After Thread Joins x = 9

Передача данных в многопоточную функцию (ссылка)

```
#include <thread>
#include <iostream>
void threadCallback(int const & x)
{
    int & y = const_cast<int &>(x);
    y++;
    std::cout<<"Inside Thread x =
"<<x<<std::endl;
}
int main()
{
    int x = 9;
    std::cout<<"In Main Thread :
Before Thread Start x =
"<<x<<std::endl;
    std::thread
```

```
threadObj(threadCallback,std::ref(x
));
    threadObj.join();
    std::cout<<"In Main Thread :
After Thread Joins x =
"<<x<<std::endl;
    return 0;
}
```

In Main Thread : Before Thread Start x = 9
Inside Thread x = 10
In Main Thread : After Thread Joins x = 10

Передача данных в многопоточную функцию (указатель)

```
#include <thread>
#include <iostream>
void threadCallback(int * x)
{
    (*x)++;
    std::cout<<"Inside Thread x =
"<<*x<<std::endl;
}
int main()
{
    int x = 9;
    std::cout<<"In Main Thread :
Before Thread Start x =
"<<x<<std::endl;
    std::thread
threadObj(threadCallback,(&x));
```

```
threadObj.join();
    std::cout<<"In Main Thread :
After Thread Joins x =
"<<x<<std::endl;
    return 0;
}
```

In Main Thread : Before Thread Start x = 9
Inside Thread x = 10
In Main Thread : After Thread Joins x = 10

Как запустить поток

// 1. функция (lec02_4.cpp)

```
#include <thread>
#include <iostream>
void thread_function()
{
    for(int i = 0; i < 10000; i++);
    std::cout<<"thread function
Executing"<<std::endl;
}
int main()
{
    std::thread threadObj(thread_function);
    for(int i = 0; i < 10000; i++);
    std::cout<<"Display From
MainThread"<<std::endl;
    threadObj.join();
    std::cout<<"Exit of Main
function"<<std::endl;
    return 0;
}
```

// 2. функтор (функциональный объект)

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

```
class DisplayThread
{
public:
    void operator()()
    {
        for(int i = 0; i < 10000; i++)
            std::cout<<"Display Thread
Executing"<<std::endl;
    }
};

int main()
{
    std::thread threadObj( (DisplayThread()) );
    for(int i = 0; i < 10000; i++)
        std::cout<<"Display From Main Thread
"<<std::endl;
    std::cout<<"Waiting For Thread to
complete"<<std::endl;
    threadObj.join();
    std::cout<<"Exiting from Main
Thread"<<std::endl;
    return 0;
}
```

Как запустить поток

// 3. лямбда-функция

```
#include <thread>
#include <iostream>
int main()
{
    std::thread threadObj( [] {
        for(int i = 0; i < 10000; i++)
            std::cout<<"Display Thread
Executing"<<std::endl;
    }); // объявление лямбды без
параметров

    for(int i = 0; i < 10000; i++)
        std::cout<<"Display From Main
Thread"<<std::endl;

    threadObj.join();
    std::cout<<"Exiting from Main
Thread"<<std::endl;
    return 0;
}
```

А что такое лямбда-функция?

// выполнение одной операции на
рядом объектов без лямбды.

// используется функтор

```
#include <algorithm>
```

```
#include <vector>
```

```
#include <iostream>
```

```
struct my_str {
```

```
    void operator()(int a) {
```

```
        std::cout<<a<<std::endl;
```

```
    }
```

```
};
```

```
int main() {
```

```
std::vector<int> v = {1, 2, 3, 4}; // C++11
```

```
// vector construct from initializer list
```

```
my_str f;
```

```
std::for_each(v.begin(), v.end(), f);
```

```
}
```

// лямбда функция

```
#include <algorithm>
```

```
#include <vector>
```

```
#include <iostream>
```

```
int main() {
```

```
std::vector<int> v = {1, 2, 3, 4}; // C++11
```

```
// vector construct from initializer list
```

```
std::for_each(v.begin(), v.end(), [](int a) {
```

```
std::cout<<a<<std::endl; });
```

```
}
```

1

2

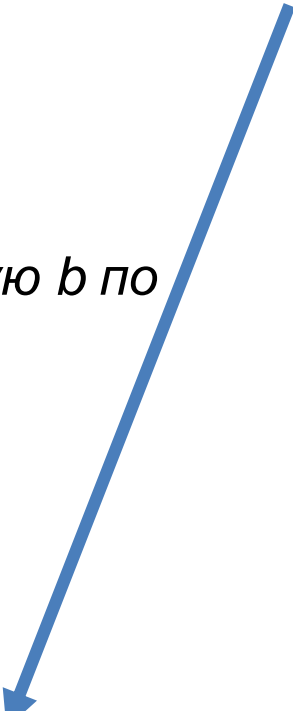
3

4

А [] скобочки зачем?

*Добавим внешнюю переменную *b* по отношению к телу лямбды*

```
#include <algorithm>
#include <vector>
#include <iostream>
int main() {
    std::vector<int> v = {1, 2, 3, 4};
    int b=9;
    std::for_each(v.begin(), v.end(), [](int a) {
        std::cout<<a+b<<std::endl; });
}
```



main.cpp: In lambda function:

main.cpp:8:60: error: 'b' is not captured

```
std::for_each(v.begin(), v.end(), [](int a) { std::cout<<a+b<<std::endl; });
                                         ^
```

А [] скобочки зачем?

Добавим внешнюю переменную b по отношению к телу лямбды

```
#include <algorithm>
#include <vector>
#include <iostream>
int main() {
    std::vector<int> v = {1, 2, 3, 4};
    int b=9;
    std::for_each(v.begin(), v.end(), [b](int a) { std::cout<<a+b<<std::endl; });
}
```




10
11
12
13

Гонка потоков (data race)

```
#include <iostream>
#include <thread>
#include <vector>
class Wallet
{
    int mMoney;
public:
    Wallet() :mMoney(0){}
    int getMoney() { return mMoney; }
    void addMoney(int money)
    {
        for(int i = 0; i < money; ++i)
        {
            mMoney++;
        }
    }
};
int testMultithreadedWallet()
{
    Wallet walletObject;
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; ++i){
```

Вначале 0 рублей



```
        threads.push_back(std::thread(&Wallet::addMoney,
            &walletObject, 1000));
    }

    for(int i = 0; i < threads.size() ; i++)
    {
        threads.at(i).join();
    }
    return walletObject.getMoney();
}

int main()
{
    int val = 0;
    for(int k = 0; k < 1000; k++)
    {
        if((val = testMultithreadedWallet()) != 5000)
        {
            std::cout << "Error at count = "<<k<<" Money in
            Wallet = "<<val << std::endl;
        }
    }
    return 0;
}
```



Гонка потоков (data race)

Добавить по одной монете к 178 монетам, используя 2 потока

mMoney	Поток 1	Поток 2
178	Загрузить mMoney в регистр 1	
178		Загрузить mMoney в регистр 2
178	Регистр 1 ++	
178		Регистр 2 ++
179	Привести mMoney в соответствие с состоянием регистра 1	
179		Привести mMoney в соответствие с состоянием регистра 2



Непредвиденный пользователем результат

Синхронизация потоков

Средства синхронизации могут использоваться для достижения двух разных целей:
захват (как правило, на короткое время) разделяемого объекта для защиты критического интервала (блоки взаимного исключения – Mutex);
ожидание (долгое или даже потенциально неограниченное) наступления некоторого события, выполнения некоторого условия (переменные состояния и семафоры). Объекты синхронизации являются переменными, к ним можно обратиться, как к данным.

Мьютексы

Mutexes

<code>mutex</code>	Mutex class (class)
<code>recursive_mutex</code>	Recursive mutex class (class)
<code>timed_mutex</code>	Timed mutex class (class)
<code>recursive_timed_mutex</code>	Recursive timed mutex (class)

Замки

Locks

<code>lock_guard</code>	Lock guard (class template)
<code>unique_lock</code>	Unique lock (class template)

Мьютекс

```
#include<thread>
#include<vector>
#include<mutex>
#include<iostream>
class Wallet
{
int mMoney;
std::mutex mutex;
public:
Wallet() :mMoney(0){}
int getMoney() { return mMoney; }
void addMoney(int money)
{
    mutex.lock();
    for(int i = 0; i < money; ++i)
    {
        mMoney++;
    }
    mutex.unlock();
};
int testMultithreadedWallet()
{
    Wallet walletObject;
    std::vector<std::thread> threads;
```

Блокируем мьютекс

Разблокируем мьютекс

```
for(int i = 0; i < 5; ++i){
    threads.push_back(std::thread(&Wallet::addMoney,
    &walletObject, 1000));
}
for(unsigned int i = 0; i < threads.size() ; i++)
{
    threads.at(i).join();
}
return walletObject.getMoney();
}
```

```
int main()
{
    int val = 0;
    for(int k = 0; k < 1000; k++)
    {
        if((val = testMultithreadedWallet()) != 5000)
        { std::cout << "Error at count = "<<k<<" Money in
        Wallet = "<<val << std::endl; }
        else
            if (k%500==0) std::cout << "Allright" <<
            std::endl;
    }
    return 0;}

```

Allright
Allright

Мьютекс

Мьютекс – синхронизирующий объект, при помощи которого множество потоков управления могут упорядочить доступ к разделяемым переменным. Mutex – mutual exclusion (взаимное исключение).

Мьютексы синхронизируют потоки, гарантируя, что только один поток в некоторый момент времени выполняет **критическую секцию** кода.

Доступны следующие действия с мьютексом: инициализация, удаление, захват или открытие, попытка захвата. Поток захватывает мьютекс в монопольное владение до тех пор, пока сам же его не освобождает. Другие потоки пытаются захватить занятый мьютекс, но им это не удастся.

Необходимость синхронизации

- Результат вычислений может быть не однозначен, зависит от условий выполнения потоков, и разные запуски программы приведут к разным результатам! Это происходит при совместном использовании любых ресурсов, например, при использовании "обычных" общих (глобальных и статических) переменных.
- Критическая секция (КС, critical section) – **часть программы, результат выполнения которой может непредсказуемо меняться, если в ходе ее выполнения состояние ресурсов, используемых в этой части программы, изменяется другими потоками.**
- Критическая секция всегда определяется по отношению к определенным критическим ресурсам (например, критическим данным), при несогласованном доступе к которым могут возникнуть нежелательные эффекты.
- Одна программа может содержать несколько критических секций, относящихся к одним и тем же данным.
- Одна критическая секция может работать с различными критическими данными. **Если несколько потоков одновременно работают с разными записями, никаких проблем нет – они возникают только при обращении нескольких потоков к одной и той же записи.**
- Для разрешения проблемы согласованного доступа к ресурсу необходимо использовать синхронизацию. Например, **обеспечить, чтобы в каждый момент времени в критической секции, связанной с определенными ресурсами, находился только один поток.** Все остальные потоки должны блокироваться на входе в критическую секцию. **Когда один поток покидает критическую секцию, один из ожидающих потоков может в нее войти.** Подобное требование обычно называется **взаимоисключением (mutual exclusion).**

Мьютекс в C++11

Мьютекс – базовый элемент синхронизации в C++11 представлен в 4 формах:

- `mutex` – предоставляет базовую взаимное исключение объекта, обеспечивает базовые функции `lock()` и `unlock()` и неблокируемый метод `try_lock()`;
- `recursive_mutex` – обеспечивает взаимное исключение объекта, который может быть заблокирован рекурсивно тем же потоком;
- `timed_mutex` – обеспечивает взаимное исключение объекта, реализующего блокировку с тайм-аутом, в отличие от обычного мьютекса, имеет еще два метода – `try_lock_for()` и `try_lock_until()`;
- `recursive_timed_mutex` – это комбинация `timed_mutex` и `recursive_mutex`.

<http://en.cppreference.com/w/cpp/thread>

Недостаток мьютекса

```
#include<thread>
#include<vector>
#include<mutex>
#include<iostream>
class Wallet
{
int mMoney;
std::mutex mutex;
public:
Wallet() :mMoney(0){}
int getMoney() { return mMoney; }
void addMoney(int money, int times)
{
mutex.lock();
for(int i = 0; i < money, times)
{
mMoney++;
}
mutex.unlock();
};
int testMultithreadedWallet()
{
Wallet walletObject;
std::vector<std::thread> threads;
```

```
for(int i = 0; i < 5; ++i){
threads.push_back(std::thread(&Wallet::addMoney,
&walletObject, 1000));
}
for(unsigned int i = 0; i < threads.size() ; i++)
{
threads.at(i).join();
}
return walletObject.getMoney();
}
```

*Исключение между блокировкой
и разблокировкой приводит
к вечной блокировке мьютекса*

```
for(int k = 0; k < 1000; k++)
{
if((val = testMultithreadedWallet()) != 5000)
{ std::cout << "Error at count = "<<k<<" Money in
Wallet = "<<val << std::endl; }
else
if (k%500==0) std::cout << "Allright" <<
std::endl;
}
return 0;}
```

std::lock_guard<std::mutex>

```
#include<thread>
#include<vector>
#include<mutex>
#include<iostream>
class Wallet
{
int mMoney;
std::mutex mutex;
public:
Wallet() :mMoney(0){}
int getMoney() { return mMoney; }
void addMoney(int money)
{
std::lock_guard<std::mutex>
lockGuard(mutex);
for(int i = 0; i < money; ++i)
{
mMoney++;
}
};
int testMultithreadedWallet()
{
Wallet walletObject;
std::vector<std::thread> threads;
```

```
for(int i = 0; i < 5; ++i){
threads.push_back(std::thread(&Wallet::addMoney,
&walletObject, 1000));
}
for(unsigned int i = 0; i < threads.size() ; i++)
{
threads.at(i).join();
}
return walletObject.getMoney();
}

int main()
{
int val = 0;
for(int k = 0; k < 1000; k++)
{
..... = 5000)
k<<" Money in

Allright" <<
std::endl;
}
return 0;}
```

*Блокировке мьютекса в конструкторе
стража замка*

*Автоматическое освобождение
по выходе из контекста*

std::unique_lock vs std::lock_guard<std::mutex>

std::unique_lock ul(mutex); // мьютекс блокируется в конструкторе. Но можно разблокировать и заблокировать снова (в отличие от RAII std::lock_guard<std::mutex>)

...
ul.unlock();

...
ul.lock();

Также вызов wait(...) для условной переменной (см. сл. 27-28)

std::condition_variable::wait(...)

требует std::unique_lock как параметр

Dead lock

```
struct Complex {  
    std::mutex mutex;  
    int i;  
    Complex() : i(0) {}  
    void mul(int x){  
        std::lock_guard<std::mutex>  
lock(mutex);  
        i *= x;  
    }  
    void div(int x){  
        std::lock_guard<std::mutex>  
lock(mutex);  
        i /= x;  
    }  
};
```

```
void both(int x, int y){  
    std::lock_guard<std::mutex>  
lock(mutex);  
    mul(x);  
    div(y);
```

```
}  
  
int main(){  
    Complex complex;  
    complex.both(32, 23);  
    return 0;  
}
```

execution expired

Рекурсивный мьютекс

```
#include <thread>
#include <mutex>
#include <iostream>
struct Complex {
    std::recursive_mutex mutex;
    int i;

    Complex() : i(5) {}
    void mul(int x){

std::lock_guard<std::recursive_mutex>
lock(mutex);
    i *= x;
    }

    void div(int x){

std::lock_guard<std::recursive_mutex>
lock(mutex);
    i /= x;
    }
```

```
void both(int x, int y){

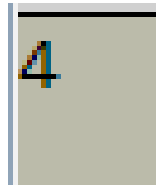
std::lock_guard<std::recursive_mutex>
lock(mutex);
    mul(x);
    div(y);
    }

void print()
{
    std::cout<<i<<std::endl;
}

};

int main(){
    Complex complex;
    complex.both(4, 5);
    complex.print();

    return 0;
}
```



Мьютекс с ожиданием по времени `std::timed_mutex`

У этого класса есть две новые функции:

`try_lock_for ()` – пытаться блокировать определенное время, и `try_lock_until ()` – пытаться блокировать до определенного момента времени

```
#include <thread>
#include <mutex>
#include <iostream>
std::timed_mutex mutex;

void work(){
    std::chrono::milliseconds timeout(100);

    if(mutex.try_lock_for(timeout)){
        std::cout <<
std::this_thread::get_id() << ": do work with
the mutex" << std::endl;

        std::chrono::milliseconds
sleepDuration(200);
```

```
std::this_thread::sleep_for(sleepDuration);

        mutex.unlock();
```

```
std::this_thread::sleep_for(sleepDuration);
    }
    else
    {
        std::cout << "Lock was busy for at
least 100 seconds" << std::endl;
    }
}
```

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

Atomic – атомарная переменная

```
#include<thread>
#include<vector>
#include<iostream>
#include<atomic>
class Wallet
{
std::atomic<int> mMoney;
public:
Wallet() :mMoney(0){}
int getMoney() { return mMoney; }
void addMoney(int money)
{
    for(int i = 0; i < money; ++i)
    {
        mMoney++;
    }
};

int testMultithreadedWallet()
{
    Wallet walletObject;
    std::vector<std::thread> threads;
```

```
    for(int i = 0; i < 5; ++i){
        threads.push_back(std::thread(&Wallet::addMoney,
&walletObject, 1000));
    }
    for(unsigned int i = 0; i < threads.size() ; i++)
    {
        threads.at(i).join();
    }
    return walletObject.getMoney();
}

int main()
{
    int val = 0;
    for(int k = 0; k < 1000; k++)
    {
        if((val = testMultithreadedWallet()) != 5000)
        { std::cout << "Error at count = "<<k<<" Money in
Wallet = "<<val << std::endl; }
        else
            if (k%500==0) std::cout << "Allright" <<
std::endl;
    }
    return 0;}
```

Сигнализация между потоками

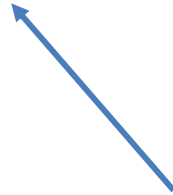
Поток 1

Делать начальные приготовления

Работать с файлом XML

Поток 2

Загрузить данные из файла XML



Это действие нельзя выполнять до загрузки

Поток 2 должен как-то сигнализировать потоку 1, что он завершил свои действия.

Возможные решение проблемы:

1) Глобальная переменная типа bool

Недостаток: постоянная проверка переменной в цикле загружает процессор

Если делать проверку реже, то время продолжения работы потока 1 (Работа с файлом XML) после завершения загрузки данных потоком 2 увеличивается, возникает задержка

2) Условные переменные (Conditional variables)

Условные переменные

```
#include <thread>
#include <functional>
#include <mutex>
#include <condition_variable>
using namespace std::placeholders;
class Application
{
    std::mutex m_mutex;
    std::condition_variable m_condVar;
    bool m_bDataLoaded;
public:
    Application()
    {
        m_bDataLoaded = false;
    }
    void loadData()
    {
        // Make This Thread sleep for 1 Second
        std::this_thread::sleep_for(std::chrono::milli
seconds(1000));
        std::cout<<"Loading Data from
XML"<<std::endl;
        // Lock The Data structure
        std::lock_guard<std::mutex>
```

```
guard(m_mutex);
        // Set the flag to true, means data is loaded
        m_bDataLoaded = true;
        // Notify the condition variable
        m_condVar.notify_one();
    }
    bool isDataLoaded()
    {
        return m_bDataLoaded;
    }
    void mainTask()
    {
        std::cout<<"Do Some
Handshaking"<<std::endl;
        // Acquire the lock
        std::unique_lock<std::mutex>
mlock(m_mutex);
```

Условные переменные (продолжение)

// Start waiting for the Condition Variable to get signaled

// Wait() will internally release the lock and make the thread to block

// As soon as condition variable get signaled, resume the thread and

// again acquire the lock. Then check if condition is met or not

// If condition is met then continue else again go in wait.

```
m_condVar.wait(mlock,  
std::bind(&Application::isDataLoaded, this));  
std::cout<<"Do Processing On loaded  
Data"<<std::endl;  
}
```

```
};
```

```
int main()
```

```
{
```

```
    Application app;
```

```
    std::thread
```

```
thread_1(&Application::mainTask, &app);  
std::thread
```

```
thread_2(&Application::loadData, &app);
```

```
thread_2.join();
```

```
thread_1.join();
```

```
return 0;
```

```
}
```