



# Multithreading

## Многопоточность и синхронизация

Евгений Козлов

2021



# Многопоточность

# Что такое процесс и поток?

## Процесс

Память

Ресурсы

Потоки

Сегмент кода

Сегмент данных

И др.

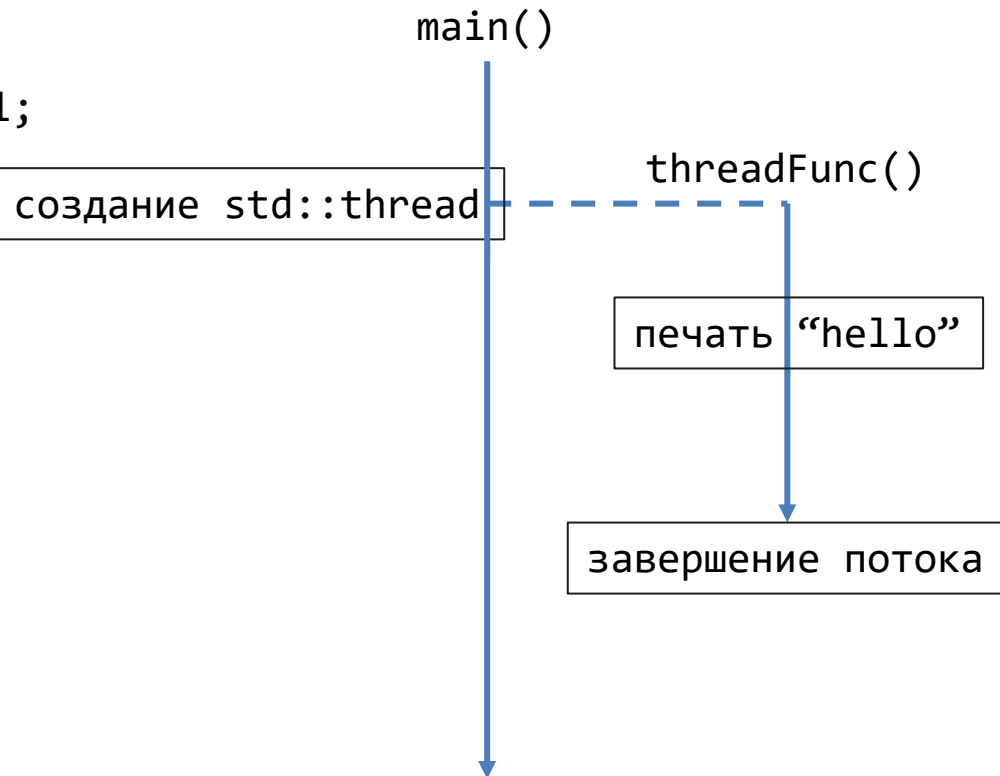
Объекты ядра

Пользовательские  
объекты

# Создание потока

```
void threadFunc()  
{  
    std::cout << "hello" << std::endl;  
}
```

```
int main()  
{  
    std::thread th(threadFunc);  
    // Новый поток начал выполняться  
}
```

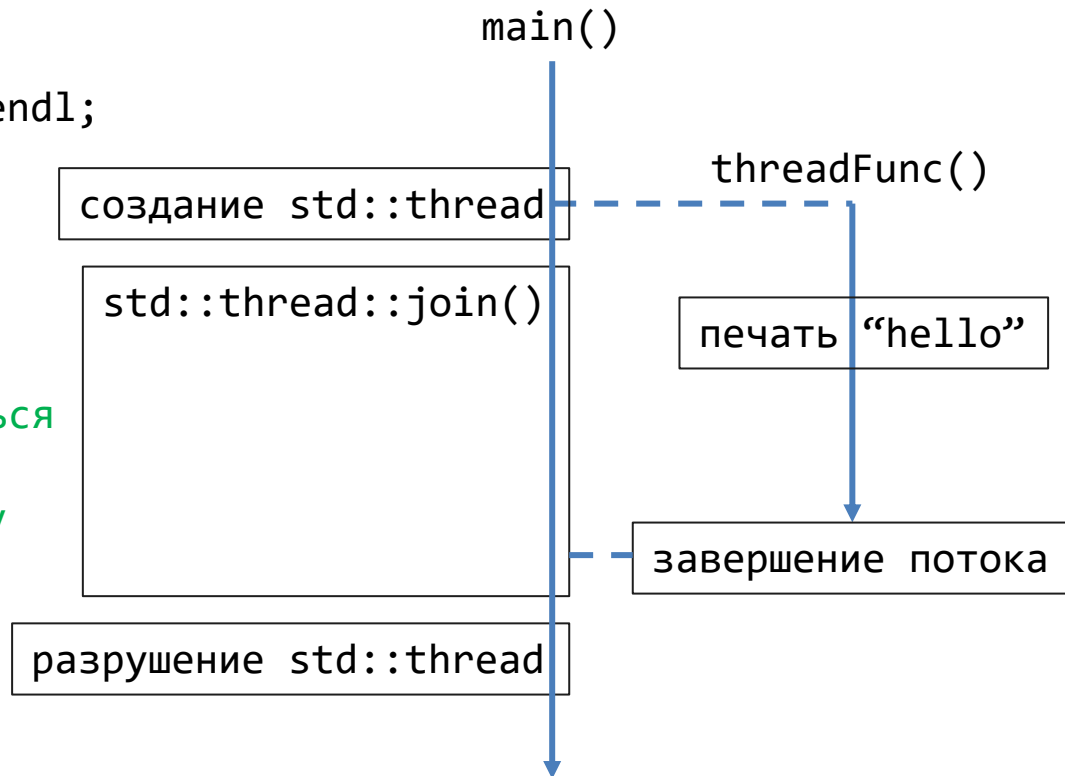




# Ожидание завершения потока

```
void threadFunc()  
{  
    std::cout << "hello" << std::endl;  
}
```

```
int main()  
{  
    std::thread th(threadFunc);  
    // Новый поток начал выполняться  
    th.join();  
    // Новый поток завершил работу  
}
```



# Фоновые потоки

```
void threadFunc()
{
    std::cout << "hello" << std::endl;
}

int main()
{
    std::thread th(threadFunc);
    // Новый поток начал выполняться
    th.detach();
} // Новый поток может все еще работать
// Новый поток будет принудительно завершен
```

Поток все еще выполняется  
в момент вызова деструктора.  
Ну и ладно.

создание `std::thread`

`std::thread::detach()`

разрушение `std::thread`

main()

threadFunc()

печать "hello"

завершение потока

# Создание потока

```
void threadFunc()
{
    std::cout << "hello" << std::endl;
}
```

```
int main()
{
    std::thread th(threadFunc);
    // Новый поток начал выполняться
} // Crash!
```

создание std::thread

main()

threadFunc()

печать "hello"

разрушение std::thread

завершение потока

Поток все еще выполняется  
в момент вызова деструктора.  
Наверное это ошибка.



# Создание потока

```
void foo() { ... }  
void bar(int a, std::string str) { ... }  
  
class Object { void baz(int i) { ... } };  
  
int main()  
{  
    std::thread th0; // Поток не был создан!  
    std::thread th1(foo);  
    std::thread th2(bar, 10, "text");  
    std::thread th3([](int i) { ... }, 10);  
  
    Object obj;  
    std::thread th4(&Object::baz, &obj, 10);  
    ...  
}
```





# Передача объектов по ссылке

```
void foo(int& a)
{
    ++a;
}

int main()
{
    int a = 1;
    std::thread th(foo, a); // Ошибка компиляции
    th.detach();
}
```



# Передача объектов по ссылке

```
void foo(int& a)
{
    ++a;
}

int main()
{
    int a = 1;
    std::thread th(foo, std::ref(a));
    th.join();
    // a == 2
}
```



# Синхронизация потоков

```
std::queue<int> q;
```

```
void input() // Поток 1
{
    while (true)
        q.push_back(rand());
}
```

```
void output() // Поток 2
{
    while (true)
    {
        if (!q.empty())
            q.pop_front();
    }
}
```



# Синхронизация потоков

```
std::queue<int> q;

void input() // Поток 1
{
    while (true)
        q.push_back(rand()); // Упадет здесь
}

void output() // Поток 2
{
    while (true)
    {
        if (!q.empty())
            q.pop_front(); // Или здесь
    }
}
```





# std::mutex

```
std::queue<int> q;  
std::mutex m;
```

```
void input() // Поток 1  
{  
    while (true)  
    {  
        m.lock();  
        q.push_back(rand());  
        m.unlock();  
        // PS: еще есть try_lock  
    }  
}
```

```
void output() // Поток 2  
{  
    while (true)  
    {  
        m.lock();  
        if (!q.empty())  
            q.pop_front();  
        m.unlock();  
    }  
}
```



# std::lock\_guard

```
// 1
{
    mutex.lock();
    ...
    mutex.unlock();
}

// 2
{
    std::lock_guard<std::mutex> guard(mutex); // mutex.lock()
    ...
} // mutex.unlock()
```



# std::unique\_lock

```
{ // std::defer_lock, std::try_to_lock, std::adopt_lock, или без параметра
  std::unique_lock<std::mutex> g(m, std::defer_lock);
  // Не заблокирован

  g.lock(); // Заблокирован
  ...
  g.unlock(); // Разблокирован

  g.lock(); // Заблокирован
  std::unique_lock<std::mutex> g2 = std::move(g);
  // Все еще заблокирован, но уже lock-ом g2
} // Разблокируется автоматически при разрушении g2
// PS: еще есть try_lock
```



# Синхронизация потоков

```
std::string* data = new std::string();

void providerThread()
{
    while (true)
        data = new std::string("one two three four five");
}

void consumerThread()
{
    while (true)
        std::cout << *data << std::endl;
}
```





# std::atomic

```
std::atomic<std::string*> data(new std::string());

void providerThread()
{
    while (true)
        data.store(new std::string("one two three four five"));
}

void consumerThread()
{
    while (true)
        std::cout << *data.load() << std::endl;
}
```



# Conditional variables

```
std::queue<int> q;  
std::mutex m;
```

```
void input() // Поток 1  
{  
    while (true)  
    {  
        std::unique_lock<mutex> g(m);  
        q.push_front(rand());  
        locker.unlock();  
        // Этот поток почти всегда спит  
        this_thread::sleep_for(  
            chrono::seconds(1));  
    }  
}
```

```
void output() // Поток 2  
{  
    while (true)  
    {  
        std::unique_lock<mutex> g(m);  
        if (!q.empty())  
        {  
            int n = q.back();  
            q.pop_back();  
            g.unlock();  
            ... // Делаем что-нибудь  
        }  
    }  
}
```

# Conditional variables

```
std::queue<int> q;  
std::mutex m;  
std::conditional_variable cv;
```

```
void input() // Поток 1  
{  
    while (true)  
    {  
        std::unique_lock<mutex> g(m);  
        q.push_front(rand());  
        locker.unlock();  
        cv.notify_one();  
        this_thread::sleep_for(  
            chrono::seconds(1));  
    }  
}
```

```
void output() // Поток 2  
{  
    while (true)  
    {  
        std::unique_lock<mutex> g(m);  
        cond.wait(g,  
            [](){ return !q.empty(); });  
        int n = q.back();  
        q.pop_back();  
        g.unlock();  
        ... // Делаем что-нибудь  
    }  
}
```



# Отложенная инициализация

```
std::unique_ptr<ThreadSafeLogger> logger;
```

```
void thread_safe_log(const Message& message)
{
    if (!logger)
        logger = std::make_unique<ThreadSafeLogger>();
    logger.log(message); // Считаем, что этот вызов потокобезопасен
}
```





# Отложенная инициализация

```
std::unique_ptr<ThreadSafeLogger> logger;  
std::mutex mutex;  
  
void thread_safe_log(const Message& message)  
{  
    {  
        std::unique_lock<std::mutex> locker(mutex);  
        if (!logger)  
            logger = std::make_unique<ThreadSafeLogger>();  
    } // std::mutex блокируется каждый раз, хотя нужно это лишь однажды  
    logger.log(message);  
}
```



# Отложенная инициализация

```
std::unique_ptr<ThreadSafeLogger> logger;  
std::once_flag flag;  
  
void thread_safe_log(const Message& message)  
{  
    std::call_once(flag, // Гарантированно вызывается не больше одного раза  
        []() { logger = std::make_unique<ThreadSafeLogger>(); });  
  
    logger.log(message);  
}
```



# Deadlock

```
struct Safe
{
    std::mutex m_mutex;
    int m_money;
};

// Должен быть потокобезопасным
void Transfer(Safe& from, Safe& to, int delta)
{
    std::lock_guard<std::mutex> fromGuard(from.m_mutex);
    std::lock_guard<std::mutex> toGuard(to.m_mutex);

    from.m_money -= delta;
    to.m_money += delta;
}
```



# Deadlock

```
struct Safe
{
    std::mutex m_mutex;
    int m_money;
};

// Должен быть потокобезопасным
void Transfer(Safe& from, Safe& to, int delta)
{
    std::lock_guard<std::mutex> fromGuard(from.m_mutex); // Deadlock
    std::lock_guard<std::mutex> toGuard(to.m_mutex);

    from.m_money -= delta;
    to.m_money += delta;
}
```

// Поток 1  
Transfer(safeA, safeB, 10);

// Поток 2  
Transfer(safeB, safeA, 20);



# Deadlock

```
struct Safe
{
    std::mutex m_mutex;
    int m_money;
};

// Должен быть потокобезопасным
void Transfer(Safe& from, Safe& to, int delta)
{
    std::lock(from.m_mutex, to.m_mutex);
    std::lock_guard<std::mutex> fromGuard(from.m_mutex, std::adopt_lock);
    std::lock_guard<std::mutex> toGuard(to.m_mutex, std::adopt_lock);

    from.m_money -= delta;
    to.m_money += delta;
}
```

// Поток 1

Transfer(safeA, safeB, 10);

// Поток 2

Transfer(safeB, safeA, 20);



# Исключения из потоков

```
void threadFunc()
{
    throw std::runtime_error("some failure");
}

int main()
{
    std::thread th(threadFunc);
    th.join();
}
```



# Исключения из потоков

```
void threadFunc()  
{    // Лучше не надо  
    // throw std::runtime_error("some failure");  
}  
  
int main()  
{  
    std::thread th(threadFunc);  
    th.join();  
}
```



# Time constraints

```
std::this_thread::sleep_for(duration);  
std::this_thread::sleep_until(time_point);
```

```
timed_mutex.try_lock_for(duration);  
timed_mutex.try_lock_until(time_point);
```

```
unique_lock.try_lock_for(duration);  
unique_lock.try_lock_until(time_point);
```

```
condition_variable.wait_for(unique_lock, duration, condition);  
condition_variable.wait_until(unique_lock, time_point, condition);  
condition_variable.wait(unique_lock);  
condition_variable.wait_for(unique_lock, duration);  
condition_variable.wait_until(unique_lock, time_point);
```





# Дополнительно

```
std::recursive_mutex rm;  
// Может быть заблокирован несколько раз из одного и того же потока  
  
std::recursive_timed_mutex rtm;  
// recursive_mutex & timed_mutex  
  
std::this_thread::get_id();  
// Системный идентификатор потока  
  
std::this_thread::yield();  
// Возможно, отдает управление другим потокам  
// А возможно и нет
```



# Асинхронное программирование



# Синхронное программирование

```
double integrate(Range range) { ... }
```

```
// 1. Передать исходные данные  
// 2. Вызвать функцию  
// 3. Сохранить результат работы функции
```

```
dest->result = integrate(src->GetRange());
```



# Асинхронное программирование

```
double integrate(Range range) { ... }
```

```
// 3. Сохранить результат работы функции  
// 1. Передать исходные данные  
// 2. Вызвать функцию
```

```
dest->result = /*...*/
```

```
...
```

```
// Прошло время
```

```
...
```

```
/*...*/ = integrate(src->GetRange());
```





# std::future & std::promise

```
double integrate(Range range) { ... }
```

```
// 3. Сохранить результат работы функции  
// 1. Передать исходные данные  
// 2. Вызвать функцию
```

```
dest->result = /* std::future */
```

```
...
```

```
// Прошло время
```

```
...
```

```
/* std::promise */ = integrate(src->GetRange());
```

# std::future & std::promise

```
// Поток 1                                // Поток 2
std::future<double> input;                  std::promise<double> output;

// Инициализация (обычно делается перед запуском Потока 2)
input = output.get_future();

// ... работа Потока 1 ...                // ... работа Потока 2 ...

// Получение результата                    // Отправка результата
double result = input.get();               output.set_value(12.98);
```

# std::future & std::promise

```
int factorial(int n) { ... }
```

```
std::future<int> fu1 = std::async(std::launch::async, factorial, 4);  
... // Факториал вычисляется прямо сейчас в другом потоке  
int x1 = fu1.get(); // Поток завершен, результат получен
```

```
std::future<int> fu2 = std::async(std::launch::deferred, factorial, 4);  
... // Никаких дополнительных потоков не создано  
int x2 = fu2.get(); // Факториал вычисляется здесь
```

```
std::future<int> fu3 = std::async(factorial, 4);  
... // Одно из двух  
int x3 = fu3.get(); // Но здесь результат будет в любом случае
```



# std::future & std::promise

```
int factorial(int n)
{
    ...
    if (n < 0)
        throw std::runtime_error("n is negative!");
    ...
}
```

```
std::future<int> fu = std::async(std::launch::async, factorial, n);
int x = fu.get(); // Либо будет получен результат, либо вылетит исключение
// future::get() можно вызвать лишь один раз
```





# std::future & std::promise

```
int increment(std::future<int> a)
{
    return a.get() + 1;
}

int main()
{
    std::promise<int> p;
    std::future<int> fa = p.get_future(); // Может быть вызвано лишь раз
    std::future<int> fb = std::async(increment, std::move(fa));
    //int b = fb.get(); // Этот вызов зависнет!
    p.set_value(5);
    int b = fb.get(); // b == 6
}
```



# std::future & std::promise

```
int increment(std::future<int> a)
{
    return a.get() + 1;
}

int main()
{
    std::promise<int> p;
    std::future<int> fa = p.get_future(); // Может быть вызвано лишь раз
    std::future<int> fb = std::async(increment, std::move(fa));
    p.set_exception(std::make_exception_ptr(
        std::runtime_error("Not going to keep promise")));
    int b = fb.get(); // Будет выброшено исключение
}
```



# std::shared\_future

```
int increment(std::shared_future<int> a)
{
    return a.get() + 1;
}

int main()
{
    std::promise<int> p;
    std::future<int> f = p.get_future(); // Может быть вызвано лишь раз
    std::shared_future<int> sf = f.share(); // Может быть вызвано лишь раз
    // f больше использовать нельзя!
    std::future<int> fu = std::async(increment, sf); // sf можно копировать
    std::future<int> fu2 = std::async(increment, sf); // sf.get() можно вызвать
    std::future<int> fu3 = std::async(increment, sf); // несколько раз
}
```



# Time constraints

```
future.wait();  
future.wait_for(duration);  
future.wait_until(time_point);
```

```
// Проверить, что результат future уже готов  
future.wait_for(0s) == future_status::ready  
// std::async(std::launch::deferred) не готов никогда!
```





# Итоги



# Зачем все это?

- Горизонтальное масштабирование:
  - Более эффективное использование процессора для вычислительно сложных задач.
- Отзывчивый интерфейс:
  - Приложение должно выполнять ресурсоемкие задачи, но при этом иметь отзывчивый пользовательский интерфейс.
- Работа с блокирующими API (сокеты, консоль, файлы).



# Q&A



# References

<https://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-1-starting-threads.html>

[http://www.bogotobogo.com/cplusplus/multithreaded4\\_cplusplus11.php](http://www.bogotobogo.com/cplusplus/multithreaded4_cplusplus11.php)

<https://habrahabr.ru/post/182610/>  
<https://habrahabr.ru/post/182626/>