

Многопоточное программирование на C++

Многопоточность

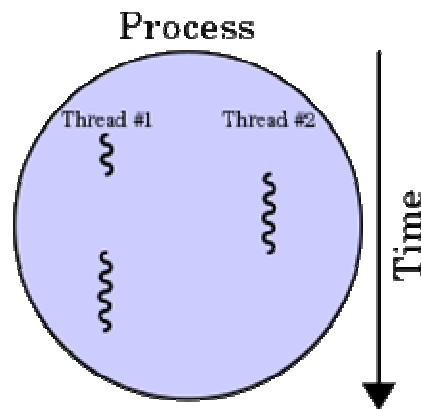
- — свойство платформы или приложения, состоящее в том, что **процесс**, порождённый в операционной системе, может **состоять из** нескольких **потоков**, выполняющихся «**параллельно**», то есть без предписанного порядка во времени.
- Такие *потоки* называют также *потоками выполнения* (от англ. *thread of execution*); иногда называют «**нитьями**» (букв. пер. англ. *thread*) или неформально «тредами».

Квазимногозадачность на уровне одного исполняемого процесса

- Все потоки процесса выполняются в **адресном пространстве** процесса.
- Все потоки процесса имеют общие **дескрипторы файлов**.
- Выполняющийся процесс имеет как минимум один (главный) поток.

На одном процессоре

- *многопоточность* обычно реализуется путём **временного мультиплексирования**: процессор переключается между разными потоками выполнения. Это переключение контекста обычно происходит достаточно часто, чтобы пользователь воспринимал выполнение потоков или задач как одновременное.



В многопроцессорных и многоядерных системах

- потоки или задачи могут реально выполняться одновременно, при этом каждый процессор или ядро обрабатывает отдельный поток или задачу.

- Стандарт C++ 1998-ого года не имел упоминаний о существовании потоков

Кроссплатформенные многопоточные библиотеки

- Набор библиотек Boost
- OpenMP
- OpenThreads
- POCO Thread (часть проекта POCO — <http://pocoproject.org/poco/info/index.html>)
- Zthread
- Pthreads (Pthreads-w32)
- Qt Threads
- Intel Threading Building Blocks
- Стандартная библиотека STL (C++11)

Многопоточное программирование в C++0x

C++0x определяет

- новую модель памяти
- библиотеку для разработки многопоточных приложений C++ threading library, включающую в себя
 - средства синхронизации,
 - создания потоков,
 - атомарные типы и операции

Класс std::thread

```
#include <iostream>
```

```
#include <thread>
```

```
#include <string>
```

```
void say_hello(const std::string& name) {  
    std::cout << "hello " << name << std::endl;  
}
```

```
int main(int argc, char * argv[]) {  
    std::thread th(say_hello, "world");  
    th.join();  
    return 0;  
}
```

Класс `std::thread`

- нельзя копировать,
- но его можно перемещать (`std::move`) и присваивать.
 - Присваивать можно только те объекты, которые не связаны ни с каким потоком, тогда объекту будет присвоено только состояние,
 - а при перемещении объекту передается состояние и право на управление потоком.

Идентификатор потока

- Каждый поток имеет свой идентификатор типа `std::thread::id`,
- который можно получить вызовом метода `get_id`
- или же вызовом статического метода `std::thread::this_thread::get_id` из функции самого потока

Идентификатор потока

```
void thread_func()
{
    std::cout << std::this_thread::get_id() << std::endl;
}
```

```
int main(int argc, char * argv[])
{
    std::thread th(thread_func);
    std::thread::id th_id = th.get_id();

    th.join();

    std::cout << th_id << std::endl;

    return 0;
}
```

Класс `std::thread`

- статический метод
- `hardware_concurrency`,
- который возвращает количество потоков, которые могут быть выполнены действительно параллельно,
- но стандарт разрешает функции возвращать 0, если на данной системе это значение нельзя подсчитать или оно не определено

Класс `std::thread`

- пара статических методов для усыпления потоков `sleep_for` и `sleep_until`
- функция `yield` для возможности передачи управления другим потокам

Мьютекс

- (англ. *mutex*, от *mutual exclusion* — «взаимное исключение») — служит для синхронизации одновременно выполняющихся потоков.
- Когда какой-либо поток, принадлежащий любому процессу, становится владельцем объекта *mutex*, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным.

Мьютекс

- Мьютексы — это простейшие двоичные семафоры, которые могут находиться в одном из двух состояний — отмеченном или неотмеченном (открыт и закрыт соответственно).
- Мьютекс отличается от семафора общего вида тем, что только владеющий им поток может его освободить, т.е. перевести в отмеченное состояние.

Задача мьютекса

- — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом.
- В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом.
- Если другому потоку будет нужен доступ к переменной, защищённой мьютексом, то этот поток засыпает до тех пор, пока мьютекс не будет освобождён.

Цель использования мьютексов

- — защита данных от повреждения в результате асинхронных изменений (состояние гонки), однако могут порождаться другие проблемы — такие, как взаимная блокировка (клинч)

std::mutex

```
#include <vector>
#include <mutex>
#include <thread>
```

```
std::vector<int> x;
std::mutex mutex;
```

```
void thread_func1() {
    mutex.lock();  x.push_back(0);  mutex.unlock();
}
```

```
void thread_func2() {
    mutex.lock();  x.pop_back();  mutex.unlock();
}
```

std::mutex

```
int main() {  
  
    std::thread th1(thread_func1);  
    std::thread th2(thread_func2);  
  
    th1.join();  
    th2.join();  
  
    return 0;  
}
```

Последний стандарт языка C++ (ISO/IEC 14882:2011)

определяет различные классы мьютексов:

- `mutex` — нет контроля повторного захвата тем же потоком;
- `recursive_mutex` — повторные захваты тем же потоком допустимы, ведётся счётчик таких захватов;
- `timed_mutex` — нет контроля повторного захвата тем же потоком, поддерживается захват мьютекса с тайм-аутом;
- `recursive_timed_mutex` — повторные захваты тем же потоком допустимы, ведётся счётчик таких захватов, поддерживается захват мьютекса с тайм-аутом.

Библиотека Boost

обеспечивает:

- реализацию мьютексов совместимых по интерфейсу со стандартом C++11 для компиляторов и платформ которые не поддерживают этот стандарт;
- реализацию дополнительных классов мьютексов: `shared_mutex` и др., которые позволяют захватывать мьютекс для совместного владения несколькими потоками только для чтения данных.

Проблема безопасности исключений в C++ threading library

- Тем не менее не рекомендуется использовать класс `std::mutex` напрямую,
- так как если между вызовами `lock` и `unlock` будет сгенерировано исключение - произойдет `deadlock` (т.е. заблокированный поток так и останется ждать).

Шаблонный класс `std::lock_guard`

- — обертка
- конструктор вызывает метод `lock` для заданного объекта, а деструктор вызывает `unlock`
- в конструктор класса `std::lock_guard` можно передать аргумент `std::adopt_lock` - индикатор, означающий, что `mutex` уже заблокирован и блокировать его заново не надо
- `std::lock_guard` не содержит никаких других методов, его нельзя копировать, переносить или присваивать

std::lock_guard

```
#include <vector>
#include <mutex>
#include <thread>

std::vector<int> x;
std::mutex mutex;

void thread_func1() {
    std::lock_guard<std::mutex> lock(mutex);
    x.push_back(0);
}

void thread_func2() {
    std::lock_guard<std::mutex> lock(mutex);
    x.pop_back();
}
```

std::lock_guard

```
int main() {  
  
    std::thread th1(thread_func1);  
    std::thread th2(thread_func2);  
  
    th1.join();  
    th2.join();  
    return 0;  
}
```

std::unique_lock

- еще один класс, контролирующий блокировки mutex-а
- предоставляет немного больше возможностей, чем std::lock_guard
- предоставляет возможность ручной блокировки и разблокировки контролируемого mutex-а с помощью методов `lock` и `unlock` соответственно

std::unique_lock

- std::unique_lock также можно перемещать с помощью вызова std::move
- объект класса std::unique_lock **может не владеть правами на mutex**, который он контролирует
- при создании объекта можно **отложить блокирование mutex-а** передачей аргумента **std::defer_lock** конструктору std::unique_lock и указать, что объект не владеет правами на mutex и вызывать unlock в деструкторе не надо
- права на mutex можно получить **позже**, вызвав метод **lock** для объекта
- функцией **owns_lock** можно проверить, **владеет ли** текущий объект **правами** на mutex

Deadlock возможен, если

- между вызовами **lock** и **unlock** будет сгенерировано исключение
- потоки **блокируют более** одного mutex-а:
 - два mutex-а А и В защищают два разных ресурса, и, двум потокам, одновременно необходим доступ к этим двум ресурсам.
 - Блокировка одного mutex-а - атомарна, но блокировка двух mutex-ов - это два отдельных действия, и, если первый поток заблокирует mutex А, в то время, как второй заблокирует mutex В, оба потока зависнут ожидая друг друга.

std::lock

- блокирует переданные ей mutex-ы без опасности deadlock-а. Функция принимает бесконечное количество шаблонных аргументов, которые должны иметь методы lock и unlock

std::lock

- `std::unique_lock<std::mutex> la(mut_a, std::defer_lock);`
- `std::unique_lock<std::mutex> lb(mut_b, std::defer_lock);`
- `std::lock(la, lb);`

std::call_once

- std::call_once создан для того, чтобы защищать общие данные во время инициализации
- это техника, позволяющая вызвать некий участок кода один раз, независимо от количества потоков, которые пытаются выполнить этот участок кода
- std::call_once - быстрый и удобный механизм для создания потокобезопасных singleton-ов

std::call_once

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

```
struct x {
    x() { std::cout << std::this_thread::get_id() << std::endl; }
};
```

```
x* instance;
```

```
void create_x() {
    instance = new x();
}
```

std::call_once

```
std::once_flag instance_flag;

void thread_func() {
    std::call_once(instance_flag, create_x);
}

int main() {
    std::thread th1(thread_func);
    std::thread th2(thread_func);

    th1.join();
    th2.join();

    return 0;
}
```

Поток ожидает наступления некого события

- Один из вариантов реализации - регулярно в цикле проверять условие наступления события,
- но это не эффективно, так как поток, вместо того, чтобы спать до наступления нужного момента, постоянно спрашивает о статусе, тем самым, мешая другим потокам.

`std::condition_variable`

это объект синхронизации, предназначенный для блокирования одного потока, пока он не будет оповещен о наступлении некоего события из другого.

std::condition_variable

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

std::vector<int> data;
std::condition_variable data_cond;
std::mutex m;
```

std::condition_variable

```
void thread_func1() {  
    std::unique_lock<std::mutex> lock(m);  
    data.push_back(10);  
    data_cond.notify_one();  
}  
  
void thread_func2() {  
    std::unique_lock<std::mutex> lock(m);  
    data_cond.wait( lock, [] { return !data.empty(); } );  
    std::cout << data.back() << std::endl;  
}
```

std::condition_variable

```
int main() {  
  
    std::thread th1(thread_func1);  
    std::thread th2(thread_func2);  
  
    th1.join();  
    th2.join();  
  
    return 0;  
}
```

Отправка оповещения

- `data_cond.notify_one();`
- `data_cond.notify_all ();`

std::condition_variable_any

- std::condition_variable работает только с блокировками типа std::unique_lock
- std::condition_variable_any может работать с любыми блокировками, поддерживающими соответствующий интерфейс

Задача

- Необходимо вызвать функцию в отдельном потоке, которая, после долгих подсчетов, вернет значение.
- Можно создать новый поток с помощью `std::thread`, но, тогда придется заботиться о возвращении результата вызывающему потоку.
- `std::thread` не дает прямой возможности это сделать.

Решение

- Поток запускается вызовом функции `std::async` и передачей ей функции/функтора для вызова в потоке.
- `std::async` возвращает объект типа `std::future<T>`, где T - тип, возвращаемый переданной в `std::async` функцией

std::async и std::future

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

```
int calculate() {
    return 2 * 2;
}
```

```
int main() {
    std::future<int> result = std::async(calculate);
    std::cout << result.get() << std::endl;
}
```

std::async и std::future

- функция `calculate` выполняется в отдельном потоке,
- при вызове метода `get`, текущий поток переходит в режим ожидания (если поток, выполняющий функцию, еще не завершил свою работу), и, возвращает результат только тогда, когда он готов.
- Если в функции `calculate` произошло исключение, то оно сохранится до вызова метода `get` и сгенерирует его заново.

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

int calculate() { throw std::runtime_error("fatal error"); }

int main() {
    std::future<int> result = std::async(calculate);
    try {
        std::cout << result.get() << std::endl;
    }
    catch (const std::runtime_error& e) {
        std::cout << e.what() << std::endl;
    }
}
```

std::promise

- позволяет передавать значение между потоками
- Каждый объект `std::promise` связан с объектом `std::future`.
- Это пара классов,
 - один из которых (`std::promise`) отвечает за установку значения,
 - а другой (`std::future`) - за его получение.

std::promise

- Первый поток может **ожидать** установки значения с помощью вызова метода
 - `std::future::wait`
 - или `std::future::get`,
- в то время, как **второй** поток
 - **установит** это значение с помощью вызова метода `std::promise::set_value`,
 - или передаст первому исключение вызовом метода `std::promise::set_exception`.


```
#include <iostream>
```

```
#include <future>
```

```
#include <thread>
```

```
std::promise<int> promise;
```

```
void thread_func1() { promise.set_value(10); }
```

```
void thread_func2() { std::cout << promise.get_future().get() << std::endl; }
```

```
int main() {
```

```
    std::thread th1(thread_func1);
```

```
    std::thread th2(thread_func2);
```

```
    th1.join();
```

```
    th2.join();
```

```
    return 0;
```

```
}
```

Для передачи исключения

- должен вызываться метод `std::promise::set_exception`, который принимает объект типа `std::exception_ptr`.
- Получить объект этого типа можно, либо вызвав `std::current_exception()` из блока `catch`, либо создать объект этого типа напрямую с помощью вызова функции `std::make_exception`.

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

std::promise<int> promise;

void thread_func1() {
    promise.set_exception(std::make_exception(std::runtime_error("fatal
error")));
}

void thread_func2() {
    try {
        std::cout << promise.get_future().get() << std::endl;
    } catch (const std::exception& e) {
        std::cout << e.what() << std::endl;
    }
}
```

```
int main() {  
    std::thread th1(thread_func1);  
    std::thread th2(thread_func2);  
  
    th1.join();  
    th2.join();  
    return 0;  
}
```