

# Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 13

08.12.2017

# Многозадачность

Основные варианты реализации многозадачности

1. Многопроцессорность
2. Многопоточность

# Многопоточность



# Многопоточность в C++11

Простой пример:

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

int main() {
    std::thread t(
        [] {
            for (size_t i = 0; i < 5; ++i)
                std::cout << "thread" << std::endl;
        }
    );
    for (size_t i = 0; i < 5; ++i)
        std::cout << "main" << std::endl;
    t.join();
}
```

# Многопоточность в C++11

Простой пример:

```
void Do(int k) {
    for (size_t i = 0; i < k; ++i) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "id = " << std::this_thread::get_id()
                  << " k = " << k << std::endl;
    }
}

int main() {
    std::thread T1(Do, 5);
    std::thread T2(Do, 3);
    T2.join();
    T1.join();
}
```

# Класс `std::thread`

Описатель потока.

- ▶ `join` — дождаться, когда поток завершится,
- ▶ `get_id` — идентификатор потока,
- ▶ `joinable` — можно ли дождаться потока,
- ▶ `detach` — забыть про поток.

# Класс `std::thread`

Описатель потока.

- ▶ `join` — дождаться, когда поток завершится,
- ▶ `get_id` — идентификатор потока,
- ▶ `joinable` — можно ли дождаться потока,
- ▶ `detach` — забыть про поток.

Поток перестает быть `joinable`:

- ▶ после вызова `join` и завершения,
- ▶ после `detach`.

# Многопоточность в C++11

Еще пример:

```
void Do(bool& f) {
    while (!f) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "waiting thread..." << std::endl;
    }
}

int main() {
    bool f = false;
    std::thread T(Do, std::ref(f));
    while (!f) {
        int x;
        std::cin >> x;
        f = (x == 5);
    }
    T.join();
}
```



# Метод detach

Поток перестает быть joinable.

```
void Do(bool& f) {
    A a;
    while (!f) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "waiting thread..." << std::endl;
    }
}
```

```
int main() {
    bool f = false;
    std::thread T(Do, std::ref(f));
    T.detach();
    while (!f) {
        int x;
        std::cin >> x;
        f = (x == 5);
    }
}
```

Где утечка?

## Исключения внутри потоков

- ▶ Исключения, которые происходят в функции потока, обрабатываются как обычно.
- ▶ Поймать исключение другого потока внутри `main` невозможно. Исключения из других потоков не передаются.

```
void Do() {  
    // ...  
    throw std::logic_error("oh no!");  
}  
  
int main() {  
    try {  
        std::thread T(Do);  
        // ...  
    } catch (...) {  
        std::cout << "something went wrong" << std::endl;  
    }  
}
```

## Суммируем вектор в 2 потока

```
int main() {
    constexpr int size = 1000000;
    std::vector<int> a(size);
    for (size_t i = 0; i < size; ++i)
        a[i] = i;
    int sum = 0;
    std::thread t1(
        [&] {
            for (size_t i = 0; i < size / 2; ++i)
                sum += a[i];
        }
    );
    std::thread t2(
        [&] {
            for (size_t i = size / 2; i < size; ++i)
                sum += a[i];
        }
    );
    t1.join();
    t2.join();
    std::cout << sum << std::endl;
}
```

## Синхронизация: mutex

Критическая секция выделяется через lock-unlock мьютекса.

```
std::mutex m;  
// ....  
  
    m.lock();  
    sum += a[i];  
    m.unlock();
```

Последовательно написать `m.lock()`; `m.lock()`; нельзя.  
Можно это сделать в `std::recursive_mutex`.

```
// RAII  
for (int i = 0; i < size / 2; ++i) {  
    std::lock_guard<std::mutex> lock(m);  
    sum += A[i];  
}
```

# Синхронизация: атомики

`std::atomic` — шаблон для типов, которые умеют копироваться и изменяться атомарно.

```
#include <iostream>
```

```
#include <thread>
```

```
#include <vector>
```

```
int main() {  
    std::atomic<long> variable(0);  
    std::thread t1([&variable] {  
        for (size_t i = 0; i < 100000; ++i) {  
            variable += 1;  
        }  
    });  
    std::thread t2([&variable] {  
        for (size_t i = 0; i < 100000; ++i) {  
            variable += 1000000;  
        }  
    });  
    t1.join(); t2.join();  
    std::cout << variable << std::endl;  
}
```

## Алиасы для std::atomic

std::atomic_char	std::atomic<char>
std::atomic_schar	std::atomic<signed char>
std::atomic_uchar	std::atomic<unsigned char>
std::atomic_short	std::atomic<short>
std::atomic_ushort	std::atomic<unsigned short>
std::atomic_int	std::atomic<int>
std::atomic_uint	std::atomic<unsigned int>
std::atomic_long	std::atomic<long>
std::atomic_ulong	std::atomic<unsigned long>
std::atomic_llong	std::atomic<long long>
std::atomic_ullong	std::atomic<unsigned long long>
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>
std::atomic_wchar_t	std::atomic<wchar_t>

# compare\_exchange

## Функции

```
bool compare_exchange_weak(T& expected, T new_val)
```

```
bool compare_exchange_strong(T& expected, T new_val)
```

проверяют, действительно ли хранится `expected`, и если так — заменяют его на новое `new_val`, если нет — `expected` заменяется на то, которое есть.

```
void lock() {  
    unsigned int current = 0;  
    while (!Spin.compare_exchange_weak(current, 1))  
        current = 0;  
}
```

# Синхронизация: условные переменные

Поток, обрабатывающий задачи:

```
while (1) {  
    unique_lock<mutex> l(mutex);  
    if (!Tasks.empty()) {  
        auto t = Tasks.front();  
        Tasks.pop_front();  
        l.unlock();  
        t->Do();  
        // ...  
    }  
}
```

Второй поток аналогично ставит задачи в очередь.  
Какая проблема?



# Синхронизация: условные переменные

Класс `std::condition_variable`, работает в связке с `mutex`.

```
unique_lock<mutex> l(Mutex);  
while (Tasks.empty()) {  
    Condition.wait(l); // ожидание условной переменной  
}
```

- ▶ `Condition.notify_one();`
- ▶ `Condition.notify_all();`

## double checking

```
if (Object == nullptr) {  
    // *  
    unique_lock<mutex> l(Mutex);  
    if (Object == nullptr) {  
        Object = new TObject;  
    }  
}
```

# Умные указатели: потокобезопасны?

Указатель `std::shared_ptr` содержит в себе два указателя:

- ▶ указатель на данные,
- ▶ указатель на управляющий блок.

## Потокобезопасность `std::shared_ptr`

```
int main() {
    std::shared_ptr<int> sp = std::make_shared<int>();
    for (size_t i = 0; i < 5; ++i) {
        std::thread t(
            [sp] {
                std::shared_ptr<int> p(sp);
                p = std::make_shared<int>(5); // OK
            }
        );
        t.detach();
    }
}
```

## Потокобезопасность `std::shared_ptr`

```
int main() {
    std::shared_ptr<int> sp = std::make_shared<int>();
    for (size_t i = 0; i < 10; ++i) {
        std::thread t(
            [&sp] {
                sp = std::make_shared<int>(5); // UB
            }
        );
        t.detach();
    }
}
```

## Потокобезопасность `std::shared_ptr`

```
int main() {
    std::shared_ptr<int> sp = std::make_shared<int>();
    for (size_t i = 0; i < 10; ++i) {
        std::thread t(
            [&sp] {
                auto p = std::make_shared<int>(5);
                std::atomic_store(&sp, p); // OK
            }
        );
        t.detach();
    }
}
```