C++

Лекция

Multithreading

Section 0

Intro

Basics

- Процессы
- Потоки
- Переключение контекста
- Concurrency/Parallelism

Multithreading motivation

- Design
- Performance

Section I

Threads

Launch Threads

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

void do_work() { std::cout << "done"; }

int main() {
    std::thread th(do_work);
    return 0;
}
```

Launch Threads

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

struct task {
    void operator()() const { std::cout << "task done"; }
};

int main() {
    std::thread th(task());
    return 0;
}
```

Launch Threads

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

struct task {
    void operator()() const { std::cout << "task done"; }
};

int main() {
    std::thread th(task());
    th.join(); // error: member reference base type
               // 'std::thread (task (*)())' is not
               // a structure or union
    return 0;
}
```

Most Vexing Parse (Click me)

Lauch Threads

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

struct task {
    void operator()() const { std::cout << "task done"; }
};

int main() {
    std::thread th{task()};
    th.join();
    return 0;
}
```

`join`

- синхронное ожидание окончания выполнения потока
- может быть вызван однажды
- после вызова: `joinable() == false`

`detach`

- запуск в фоне (daemon thread)
- после вызова: `joinable() == false`

using a function with parameters

```cpp
#include <thread>

void do_work(int i) {/*...*/}

int main() {
   std::thread th(do_work, 10);
   th.detach();
}
```

using a function with parameters

```cpp
#include <thread>

void do_work(int& i) { i = 11; }

int main() {
    int i = 10;
    std::thread th(do_work, i);
    th.join();
    assert(i == 11);
}
```

what is wrong here?

using a function with parameters

```
#include <thread>
#include <functional>

void do_work(int& i) { i = 11; }

int main() {
    int i = 10;
    std::thread th(do_work, std::ref(i));
    th.join();
    assert(i == 11);
}
```

Thread Id

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

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

int main() {
    std::thread th{do_work};
    auto id = th.get_id();
    th.join();
    std::cout << id;
}
```

```
0x70000f26f000
0x70000f26f000
```

Thread Id

`std::thread::id` — копируемый, сравнимый, хешируемый

Usage:

- Выбор ветки кода в зависимости от id
- Сохранение id в структуре данных: для определения потока, который может работать со структурой

Section II

Shared data

Shared data

- Механизм потоков легок и эффективен в использовании разделяемых данных
- Проблемы с изменяемыми данными

Issues

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

int x;
void do_work() {
    for (int i = 0; i < 100'000'000; ++i) {
        x += 1;
    }
}

int main() {
    x = 0;
    std::thread th1(do_work), th2(do_work);
    th1.join(); th2.join();
    std::cout << x;
}
```

Issues

- Race condition (problematic if violate invariants)
- Data race (UB)

Avoiding race conditions approaches

1. Only one thread can modify (and see intermediate states)
2. Lock-free
3. Transaction (STM)

Mutual exclusion

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

std::vector<int> vs;

void do_work() {
    for (int i = 0; i < 100; ++i) {
        vs.push_back(i);
    }
}

int main() {
    std::thread th1(do_work), th2(do_work);
    th1.join(); th2.join();
    std::cout << vs.size();
}
```

Any problems?

Mutual exclusion

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

std::vector<int> vs;
std::mutex vs_access;

void do_work() {
    for (int i = 0; i < 100; ++i) {
        vs_access.lock();
        vs.push_back(i);
        vs_access.unlock();
    }
}

int main() {
    std::thread th1(do_work), th2(do_work);
    th1.join(); th2.join();
    std::cout << vs.size();
}
```

Mutual exclusion

use RAII

```cpp
std::vector<int> vs;
std::mutex vs_access;

void do_work() {
    for (int i = 0; i < 100; ++i) {
        std::lock_guard<std::mutex> guard(vs_access);
        vs.push_back(i);
    }
}
```

Mutual exclusion

C++17

```
std::vector<int> vs;
std::mutex vs_access; // global variables is poor, use classes

void do_work() {
    for (int i = 0; i < 100; ++i) {
        std::scoped_lock lock(vs_access);
        vs.push_back(i);
    }
}
```

Deadlock

```cpp
std::mutex mtx1;
std::mutex mtx2;

void work1() {
    std::lock_guard lock1(mtx1);
    std::this_thread::sleep_for(std::chrono::seconds{1});
    std::lock_guard lock2(mtx2);
}

void work2() {
    std::lock_guard lock1(mtx2);
    std::this_thread::sleep_for(std::chrono::seconds{1});
    std::lock_guard lock2(mtx1);
}

int main() {
    std::thread th1(work1), th2(work2);
    th1.join(); th2.join();
}
```

Deadlock

```
#include <thread>

std::mutex mtx;

void work(int i) {
    std::lock_guard lock(mtx);
    std::this_thread::sleep_for(std::chrono::seconds{1});
    if (i > 0) work(i - 1);
}

int main() {
    std::thread th1{work, 1};
    th1.join();
}
```

Deadlock

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

struct account { size_t balance; std::mutex mtx; };

void transfer(account* from, account* to, size_t money) {
    std::lock_guard lock1(from->mtx);
    std::this_thread::sleep_for(std::chrono::milliseconds {100});
    std::lock_guard lock2(to->mtx);

    if (from->balance >= money) { from->balance -= money; to->balance += money; }
}

int main() {
    account ac1{1000};
    account ac2{1000};
    std::vector<std::thread> threads;
    for (size_t i = 0; i < 10; ++i) {
        threads.emplace_back(transfer, &ac1, &ac2, i);
        threads.emplace_back(transfer, &ac2, &ac1, i);
    }
    std::for_each(threads.begin(), threads.end(), [](auto& t) { t.join(); });

    std::cout << ac1.balance + ac2.balance;
}
```

Avoiding Deadlock

```
void transfer(account* from, account* to, size_t money) {
    std::lock(from->mtx, to->mtx);
    std::lock_guard lock1(from->mtx, std::adopt_lock);
    std::lock_guard lock2(to->mtx, std::adopt_lock);

    if (from->balance >= money) {
        from->balance -= money;
        to->balance += money;
    }
}
```

Avoiding Deadlock

C++17

```cpp
void transfer(account* from, account* to, size_t money) {
    std::scoped_lock lock(from->mtx, to->mtx);

    if (from->balance >= money) {
        from->balance -= money;
        to->balance += money;
    }
}
```

Avoiding Deadlock

- Остерегаться вложенных захватов мьютекса
- Фиксировать порядок захвата

std::unique_lock

```
void transfer(account* from, account* to, size_t money) {
    std::unique_lock lock1(from->mtx, std::defer_lock);
    std::unique_lock lock2(to->mtx, std::defer_lock);
    std::lock(lock1, lock2);

    if (from->balance >= money) {
        from->balance -= money;
        to->balance += money;
    }
}
```

`std::unique_lock`

- `lock`, `unlock`, `try_lock`
- `owns_lock`
- flexible, but larger/slower

Гранулярность блокировок

```cpp
#include <thread>
#include <iostream>
#include <list>

class T { /*...*/ };

std::list<T> cache;
std::mutex cache_access;

void reset_cache() {
    std::scoped_lock lock(cache_access);
    cache.clear();
}

int main() {
    /*...*/
    std::thread(reset_cache).detach();
    /*...*/
}
```

Гранулярность блокировок

Если мьютекс захвачен:

- Не выполнять вычисления, не связанные данными, находящимися под мьютексом
- Исключить I/O операции

Section III

Waiting for events

Waiting for ready

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

int data;
bool ready = false;

void producer(int i) {
    data = i;
    ready = true;
}

void consumer() {
    while (!ready) { }
    std::cout << "ready: " << data;
}

int main() {
    std::thread c(consumer);
    std::this_thread::sleep_for(std::chrono::seconds{1});
    std::thread p(producer, 10);
    p.join(); c.join();
}
```

Waiting for ready

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

int data;
bool ready = false;
std::mutex mtx;

void producer(int i) {
    std::scoped_lock lock{mtx};
    data = i;
    ready = true;
}

void consumer() {
    std::unique_lock lock{mtx};
    while (!ready) {
        lock.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds{100});
        lock.lock();
    }
    std::cout << "ready: " << data;
}
```

Waiting for ready

using cv

```cpp
#include <thread>
#include <iostream>
#include <condition_variable>

int data;
bool ready = false;
std::mutex mtx;
std::condition_variable cv;

void producer(int i) {
    {
        std::scoped_lock lock{mtx};
        data = i;
        ready = true;
    }
    cv.notify_one();
}

void consumer() {
    std::unique_lock lock{mtx};
    while (!ready) cv.wait(lock); // spurious wakeup
    std::cout << "ready: " << data;
}
```

Async

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

std::thread::id heavy_calculation() {
    std::this_thread::sleep_for(std::chrono::seconds{3});
    return std::this_thread::get_id();
}

int main() {
    auto future
        = std::async(std::launch::async, heavy_calculation);

    // payload
    std::this_thread::sleep_for(std::chrono::seconds{3});

    assert(future.valid());
    future.wait();
    assert(future.valid());
    auto async_thread_id = future.get();
    assert(!future.valid());

    std::cout << std::this_thread::get_id() << " "
              << async_thread_id;
}
```

std::packaged_task

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

int heavy_calculation() {
    std::this_thread::sleep_for(std::chrono::seconds{3});
    return 42;
}

int main() {
    std::packaged_task<int()> task{heavy_calculation};
    auto future = task.get_future();
    std::thread{std::move(task)}.detach();
    auto res = future.get();
    assert(res == 42);
}
```

- Может пригодиться при реализации тред-пула
- Менеджмент задач

std::promise

```cpp
#include <future>
#include <thread>

void heavy_calculation(std::promise<int> p) {
    std::this_thread::sleep_for(std::chrono::seconds{3});
    p.set_value(42);
}

int main() {
    std::promise<int> promise;
    auto future = promise.get_future();
    std::thread{heavy_calculation, std::move(promise)}.detach();
    auto res = future.get();
    assert(res == 42);
}
```

Exceptions

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

int heavy_calculation() {
    std::this_thread::sleep_for(std::chrono::seconds{3});
    throw std::runtime_error("error");
}

int main() {
    std::packaged_task<int()> task{heavy_calculation};
    auto future = task.get_future();
    std::thread{std::move(task)}.detach();

    future.wait();
    assert(future.valid());

    try {
        future.get();
        assert(false);
    } catch(const std::runtime_error& ex) {
        std::cout << ex.what();
    }
}
```

Exceptions (2)

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

void heavy_calculation(std::promise<int> p) {
    std::this_thread::sleep_for(std::chrono::seconds{3});
    try {
        throw std::runtime_error("error");
    } catch (...) {
        p.set_exception(std::current_exception());
    }
}

int main() {
    std::promise<int> promise;
    auto future = promise.get_future();
    std::thread{heavy_calculation, std::move(promise)}.detach();
    future.wait();
    assert(future.valid());
    try {
        future.get();
        assert(false);
    } catch(const std::runtime_error& ex) {
        std::cout << ex.what();
    }
}
```

*) make_exception_ptr