

Многопоточное программирование

Иван Ганкевич

2018

Пример: пул задач

```
struct Task { virtual void operator()() = 0; }

// Пул задач
class TaskPool {
    std::mutex mtx; // взаимное исключение
    std::queue<Task> tasks;
public:
    void push(Task t) {
        mtx.lock();
        tasks.push(t);
        mtx.unlock();
    }
};
```

Mutex — (сокp.) mutual exclusion.

```
namespace std {  
    // сторожевой объект (блокировка)  
    template <class Mutex>  
    class lock_guard {  
        Mutex& mtx;  
    public:  
        lock_guard(Mutex& m): mtx(m) { mtx.lock(); }  
        ~lock_guard() { mtx.unlock(); }  
        lock_guard(const lock_guard&) = delete;  
        lock_guard& operator=(const lock_guard&) = delete;  
    };  
}
```

```
namespace std {  
    // блокировка с владельцем  
    template <class Mutex>  
    class unique_lock {  
        Mutex* mtx;  
        bool owner = false;  
    public:  
        unique_lock(Mutex& m): mtx(&m) { lock(); }  
        ~unique_lock() { unlock(); }  
        void lock() { mtx->lock(); }  
        void unlock() { mtx->unlock(); }  
    };  
}
```

```
class TaskPool {  
    std::mutex mtx;  
    std::queue<Task> tasks;  
public:  
    void push(Task t) {  
        // безопасная блокировка  
        std::lock_guard<std::mutex> lock{mtx};  
        tasks.push(t);  
    }  
};
```

Типы взаимных исключений

Класс	Методы
<code>std::mutex</code>	<code>lock</code> <code>try_lock</code> <code>unlock</code>
<code>std::recursive_mutex</code> <code>std::timed_mutex</code>	<code>try_lock_for</code> <code>try_lock_until</code>
<code>std::recursive_timed_mutex</code>	

Рекурсивную блокировку один и тот же поток может ставить несколько раз.

```
class TaskPool {  
    std::mutex mtx;  
    std::condition_variable cv; // условная переменная  
    std::queue<Task> tasks;  
public:  
    void push(Task t) {  
        std::lock_guard<std::mutex> lock(mtx);  
        tasks.push_back(t);  
        cv.notify_one(); // оповестить один из ждущих потоков  
    }  
};
```

```
class TaskPool {  
    ...  
    std::atomic<bool> stopped{false}; // состояние  
    void loop() {  
        std::unique_lock<std::mutex> lock{mtx};  
        cv.wait(lock, [this, &lock] () { // только unique_lock  
            while (!tasks.empty()) {  
                Task task = tasks.front();  
                tasks.pop();  
                unlock_guard unlock{mtx}; // lock_guard наоборот  
                task();  
            }  
            return stopped;  
        });  
    }  
};
```



```
class TaskPool {  
    ...  
    std::vector<std::thread> threads{4}; // потоки  
    void start() {  
        for (auto& thr : threads) {  
            thr = std::thread{[this] () { this->loop(); }};  
        }  
        stopped = false; // изменяем состояние  
    }  
    void stop() { stopped = true; }  
    void wait() { // ожидание завершения потоков  
        for (auto& thr : threads) {  
            if (thr.joinable()) { thr.join(); }  
        }  
    }  
};
```

Атомарные операции

```
namespace std {  
    template <class T>  
    class atomic {  
        T number; // интегральный тип или указатель  
    public:  
        atomic(T n): number(n) {}  
        operator T() { return load(); } // атомарная загрузка  
        T operator=(T n) { store(n); return n; } // сохранение  
        // ... другие операторы  
    };  
}
```

- ▶ Если атомарная операция не поддерживается процессором, то она заменяется на операцию с блокировкой.
- ▶ Отсутствие блокировки гарантируется только для `std::atomic_flag`.

Пример: счетчик

```
std::atomic<unsigned long> counter{0}; // безопасный счетчик
std::vector<std::thread> threads;
for (int i=0; i<10; ++i) {
    threads.emplace_back([&counter] () { ++counter; });
}
unsigned long value = counter;
std::cout << value << '\n'; // 10
```

Пример: циклическая блокировка

```
class SpinMutex {  
    std::atomic_flag f = ATOMIC_FLAG_INIT; // false  
public:  
    void lock() { while (f.test_and_set()); } // установка флага  
    void unlock() { f.clear(); } // сброс флага  
};
```

```
class TaskPool {  
    SpinMutex mtx;  
    std::condition_variable_any cv; // для любых блокировок  
    ...  
public:  
    void push(Task t) {  
        std::lock_guard<SpinMutex> lock{mtx};  
        ...  
    }  
    void loop() {  
        std::lock_guard<SpinMutex> lock{mtx};  
        cv.wait(mtx, ...);  
    }  
};
```

```
namespace std {  
    void condition_variable::wait(unique_lock<mutex>& mtx) {  
        // СИСТЕМНЫЕ ВЫЗОВЫ  
    }  
    template <class Pred> void  
    condition_variable::wait(unique_lock<mutex>& mtx, Pred p) {  
        while (!p) { wait(mtx); }  
    }  
}
```

```
namespace std {  
class condition_variable_any {  
    condition_variable cv;  
    mutex mtx;  
public:  
    template <class Mutex>  
    void wait(Mutex& mtx1) {  
        unique_lock<mutex> lock1{mtx}; // блокировка mtx  
        unlock_guard<Lock> unlock{mtx1}; // разблокировка mtx1  
        unique_lock<mutex> lock2{std::move(mtx)}; // нет блокировки  
        cv.wait(lock2);  
        // деструктор lock2: разблокировка mtx  
        // деструктор unlock: блокировка mtx1  
        // деструктор lock1: ничего  
    }  
};  
}
```

Поддержка сторонних блокировок реализована через...
еще одни блокировки!

Системный семафор

```
#include <semaphore.h> // системный файл (POSIX)
class Semaphore {
    sem_t sem; // целочисленный тип
public:
    Semaphore() { sem_init(&sem,0,1); } // 1 – начальное значение
    ~Semaphore() { sem_destroy(&sem); } // 0 – condition_variable
    void lock() { sem_wait(&sem); } // декремент на 1
    void unlock() { sem_post(&sem); } // инкремент на 1
    void notify_one() { unlock(); }
    void wait() { lock(); }
    template <class Mutex> void wait(Mutex& mtx) { // любой
        unlock_guard<Mutex> unlock{mtx}; wait(); // мьютекс
    }
};
```

```
class TaskPool {  
    Semaphore mtx; // или SpinMutex  
    Semaphore cv;  
  
    ...  
public:  
    void push(Task t) {  
        std::lock_guard<Semaphore> lock{mtx};  
        ...  
    }  
    void loop() {  
        std::lock_guard<Semaphore> lock{mtx};  
        cv.wait(mtx, ...);  
    }  
};
```

Преимущества Semaphore:

- ▶ `std::mutex + std::condition_variable` в одном классе.
- ▶ `wait` с любыми мьютексами (в т.ч. `std::mutex`).
- ▶ Это просто целое число.
- ▶ Синхронизация и потоков, и процессов.

Future/promise

```
void long_running_task(std::promise<int> pr) {  
    // ...  
    pr.set_value(11);  
}  
std::promise<int> promise; // для другого потока  
std::future<int> future = promise.get_future(); // для текущего  
std::thread t{long_running_task, std::move(promise)};  
std::cout << "result=" << future.get() << '\n'; // 11  
t.join();
```

Пакетное задание

```
int long_running_task() {  
    // ...  
    return 11;  
}  
  
std::packaged_task<int()> task(long_running_task);  
std::future<int> future = task.get_future(); // для текущего  
std::thread t{task};  
std::cout << "result=" << future.get() << '\n'; // 11  
t.join();
```

Асинхронный вызов

```
int long_running_task() {  
    // ...  
    return 11;  
}  
auto future = std::async(std::launch::async, long_running_task);  
std::cout << "result=" << future.get() << '\n'; // 11
```

- ▶ `std::future` — объект для получения результата асинхронных вычислений.
- ▶ `std::promise` — объект для передачи результата асинхронных вычислений.
- ▶ `std::packaged_task` — объект, содержащий функцию и результат ее работы.
- ▶ `std::async` — функция, выполняющая другую функцию асинхронно.

Пул задач без `std::packaged_task`

```
struct Task { virtual void operator()() = 0; }

// определяем задачу
struct SumTask: public Task {
    float a, b; float* result;
    void operator()() override {
        *result = a + b;
    }
};

// отправляем задачи на исполнение
TaskPool pool{4}; pool.start();
float result1, result2;
pool.push(SumTask{10,20,&result1});
pool.push(SumTask{20,40,&result2});
pool.stop(); pool.wait(); // ждать завершения обоих заданий?
std::cout << result1 << ' ' << result2 << '\n';
```


Пул задач с `std::packaged_task`

```
// перепишем метод TaskPool
typedef std::packaged_task<float()> Task;
// результат всегда float?
std::future<float> TaskPool::push(Task&& t) {
    auto f = t.get_future();
    std::lock_guard<std::mutex> lock{mtx};
    tasks.push(std::move(t));
    cv.notify_one();
    return f;
}
// отправляем задачи на исполнение
TaskPool pool{4}; pool.start();
auto result1 = pool.push(Task([] () { return 10.f + 20.f; }));
auto result2 = pool.push(Task([] () { return 20.f + 40.f; }));
std::cout << result1.get() << ' ' << result2.get() << '\n';
pool.stop(); pool.wait();
```

Попытка №2

```
// тип для хранения любых значений
```

```
// std::any в C++17
```

```
class Any {  
    void* ptr;  
    int type;  
    // ...  
};
```

```
std::future<Any> TaskPool::push(std::packaged_task<Any()> t);
```