Rubius Academy

Курс-интенсив

Программирование на C++

Параллельный C++
Часть 2

academy.rubius.com

konstantin.dobrychev@rubius.com

Константин Добрычев

Ещё раз про блокировки

- std::lock_guard
- std::unique_lock
- std::shared_lock
- std::scoped_lock

std::lock_guard

```
std::lock_guard lock(mutex);
```

↓

```
mutex.lock();
// critical section
mutex.unlock();
```

std::unique_lock

```
std::unique_lock lock(mutex);
```

- Уникальное владение с перемещением
- Поведение как у захваченного объекта
- Дополнительная гибкость
- Чуть больше накладных расходов

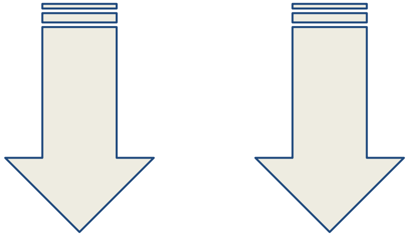
std::shared_lock

```
std::shared_lock lock(mutex);
```

```
mutex.lock_shared();
// critical section
mutex.unlock_shared();
```

std::scoped_lock

```
std::scoped_lock lock(m1, m2);
```



```
std::lock(m1, m2);
std::lock_guard lk1(m1, std::adopt_lock);
std::lock_guard lk2(m2, std::adopt_lock);

std::unique_lock lk1(m1, std::defer_lock);
std::unique_lock lk2(m2, std::defer_lock);
std::lock(lk1, lk2);
```

Атомарные операции

https://en.cppreference.com/w/cpp/atomic/atomic

```
#include <atomic>

std::atomic<int> index = 42;
const int value = index;

index.fetch_add(3);
index.fetch_sub(5);

index.store(12);
const int value = index.load(std::memory_order_seq_cst);

const int prev = index.exchange(11);

int expected = 11;
while(!index.compare_exchange_weak(expected, 34));

index.compare_exchange_strong(expected, 34);
```

std::atomic_flag

```
class SpinLock final
{
public:
  SpinLock() : flag_(ATOMIC_FLAG_INIT) {}

  void lock() {
    while(flag_.test_and_set(std::memory_order_acquire));
  }

  void unlock() { flag_.clear(std::memory_order_release); }

private:
    std::atomic_flag flag_;
};
```

lock-free структуры

```
template<typename T>
class Stack
{
    struct Node {
        std::shared_ptr<T> value;
        std::shared_ptr<Node> next;

        explicit Node(T&& value)
            : value(std::make_shared<T>(std::forward<T>(value)))
       {}
    };

    void push(T&& value);
    std::shared_ptr<T> pop()
};
```

lock-free структуры

```
template<typename T>
void Stack::push(T&& value) {
    auto node = std::make_shared<Node>(std::forward<T>(value));
    node->next = std::atomic_load(&head_);
    while (!std::atomic_compare_exchange_weak(&head_, &node->next, node));
}


template<typename T>
std::shared_ptr<T> Stack::pop() {
    auto node = std::atomic_load(&head_);
    while (node && !std::atomic_compare_exchange_weak(&head_, &node, node->next));

    return node ? node->value : nullptr;
}
```

Чего ждём?

```
while (!device.isReady()) {
    // waiting ...
}
```

```
std::mutex mutex;
std::unique_lock lock(mutex);

while (!device.isReady()) {
    lock.unlock();
    std::this_thread::sleep_for(100ms);
    lock.lock();
}
```

Условные переменные

```
#include <condition_variable>

struct Device {
  std::mutex mutex;
  std::condition_variable cv;

  bool isReady() const;
  ByteArray read();
  void write(ByteArray data);
};


void writerThread(Device& device)
{
    std::lock_guard lock(device.mutex);
    device.write(std::move(get_data()));
    device.cv.notify_one();
}
```

```
void readerThread(Device& device)
{
  while (true) {
    std::unique_lock lock(device.mutex);

    device.cv.wait(lock, [&device] {
      return device.isReady();
    });

    ByteArray data = device.read();

    lock.unlock();

    process(data);
  }
}
```

Фоновые задачи

```
void collectInfo(const Person& preson);

#include <future>

// блокируем поток
collectInfo(person);

// не блокируем поток
auto future = std::async(
    collectInfo, person
);
```

```
std::future<ByteArray> asyncRead();

auto future = asyncRead();
ByteArray data = future.get();
```

```
std::future<void> future = std::async(
    [person] {
        collectInfo(person);
    }
);

// теперь блокируем
future.wait();

// проверяем готовность
if (future.valid())
```

```
std::vector<std::future<Result>> tasks;

for (const auto& id : ids()) {
    tasks.push_back(runTask(id))
}
```

std::async

```
// запуск в новом потоке
std::async(std::launch::async, runTask);


// отложенный запуск
std::async(std::launch::deferred, &Device::read, &device);


// на усмотрение реализации
std::async(std::launch::async | std::launch::deferred, [] {
    // do something
});
```

std::packaged_task

```
std::packaged_task<bool(int, std::string)> task(
    [=](int, std::string) {
        // do something
        return success;
    }
);                      auto future = task.get_future();

task(42, "Hello");      std::thread thread(task, 42, "Hello");

taskQueue.enqueue(task);

// ...

while (!taskQueue.empty()) {
    execute(taskQueue.dequeue(), 42, "Hello");
}
```

std::promise

```
std::promise<int> promise;
std::future<int> future = promise.get_future();

std::thread([promise = std::move(promise)]() mutable {
    // ...

    try {
        promise.set_value(findTheAnswer());
    } catch (...) {
        promise.set_exception(std::current_exception());
    }
}).detach();

std::cout << future.get() << "\n";
```

std::promise

```
void aio_readline(aio_socket fd, aio_cb callback, aio_cb_data data);
```

```
namespace aio
{
    class socket
    {
        // ...
        std::future<std::string> readline();

    private:
        aio_socket fd;
    };
};
```

std::promise

```
std::future<std::string> socket::readline()
{
  auto callback = [](const char* line, aio_cb_data data) noexcept {
    auto promise = static_cast<std::promise<std::string>*>(data);
    promise->set_value(line);
    delete promise;
  };

  auto* promise = new std::promise<std::string>;
  aio_readline(fd, callback, promise);

  return promise->get_future();
}

aio::socket socket;
std::cout << socket.readline().get() << "\n";
```

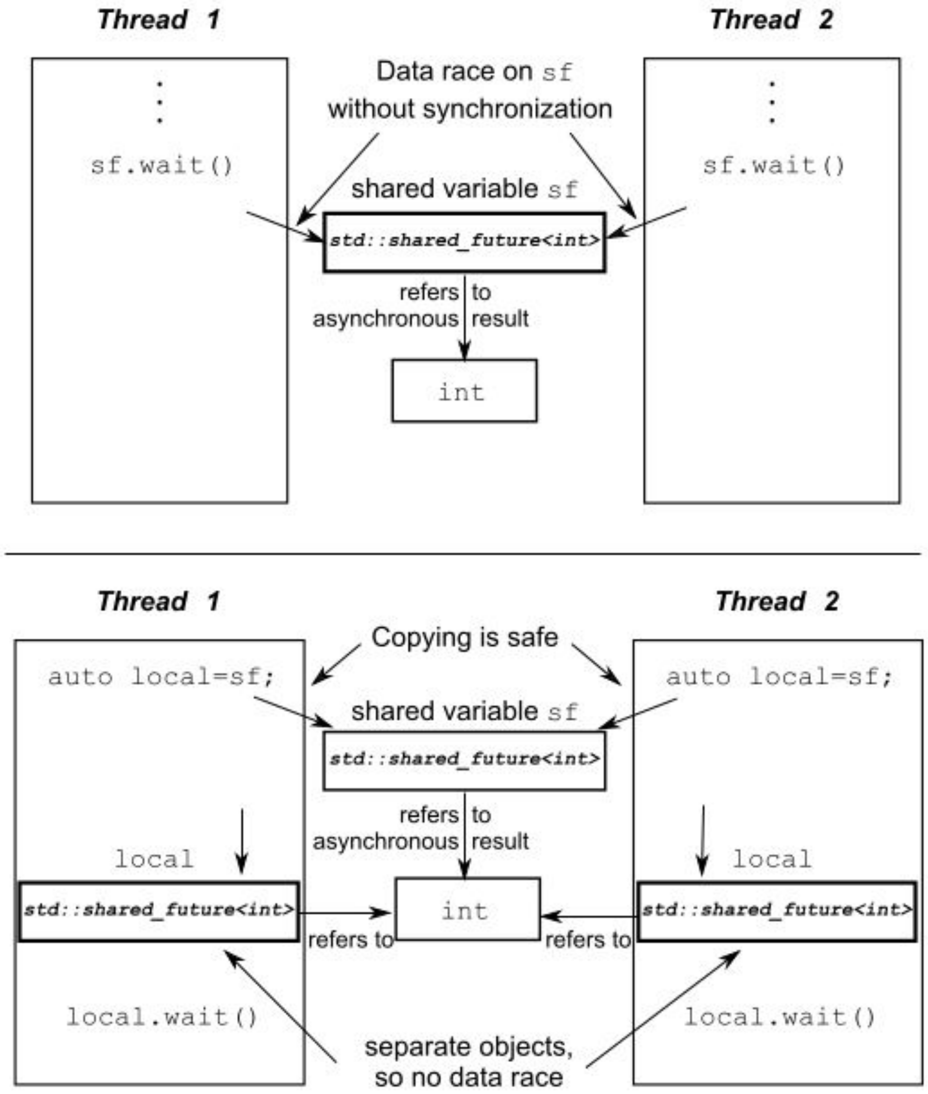
std::shared_future

```
std::future<int> future = findTheAnswer();

std::async([future = future.share()] {
    const int answer = future.get();
    // do something
});

std::async([future = future.share()] {
    const int answer = future.get();
    // do another thing
});
```

std::shared_future



boost::future

```
#include <boost/thread/future.hpp>

auto future = boost::async([] { sendRequest("GET_VALUE"); })
    .then([](auto) { return getValue(); })
    .then([](auto future) { return std::pow(future.get(), 3); })
    .then(
       [](auto future) {
            const int value = future.get();
            sendRequest("SET_VALUE", value);
            return value;
       }
    );

std:: cout << future.get() << "\n";
```

Сопрограммы

https://en.cppreference.com/w/cpp/language/coroutines

```
while (true) {
    Connection connection = co_await server.connection();

    auto request = co_await connection.readline();
    auto response = co_await process(connection, request);
    co_await connection.send(response);
}
```

```
Task<int> findTheAnswer()
{
    co_await think();
    co_await thinkAgain();

    co_return 42;
}
```

```
Generator<int> numbers(int begin) {
    while(true) {
        co_yield ++begin;
    }
}

for (int number : numbers()) { … }
```

Stackful coroutines

Boost.Coroutine2

https://www.boost.org/doc/libs/1_71_0/libs/coroutine2/doc/html/index.html

Boost.Fiber

https://www.boost.org/doc/libs/1_71_0/libs/coroutine2/doc/html/index.html

Куда копать дальше?

Параллельное программирование на C++ в действии
Энтони Уильямс
Практика разработки многопоточных программ
ДМК
MANNING
ozon.ru

Rubius Academy

Курс-интенсив

Программирование на C++

Параллельный С++ Часть 2

academy.rubius.com

konstantin.dobrychev@rubius.com

Константин Добрычев