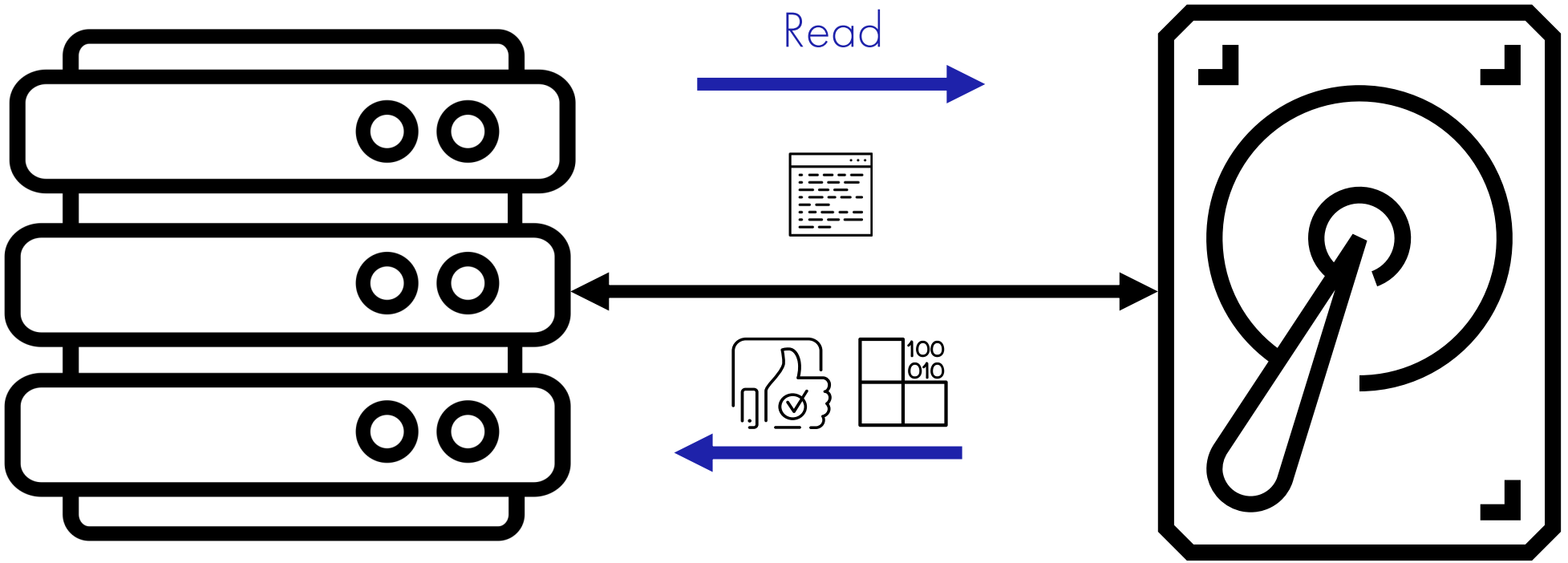
В Asio за корутинами

Илья Казаков

Старший инженер по разработке ПО

- Страдаю на C++ пятый год
- А также страдаю с Linux kernel
- И все это для СХД (Систем хранения данных)

Самое точное схематичное представление СХД

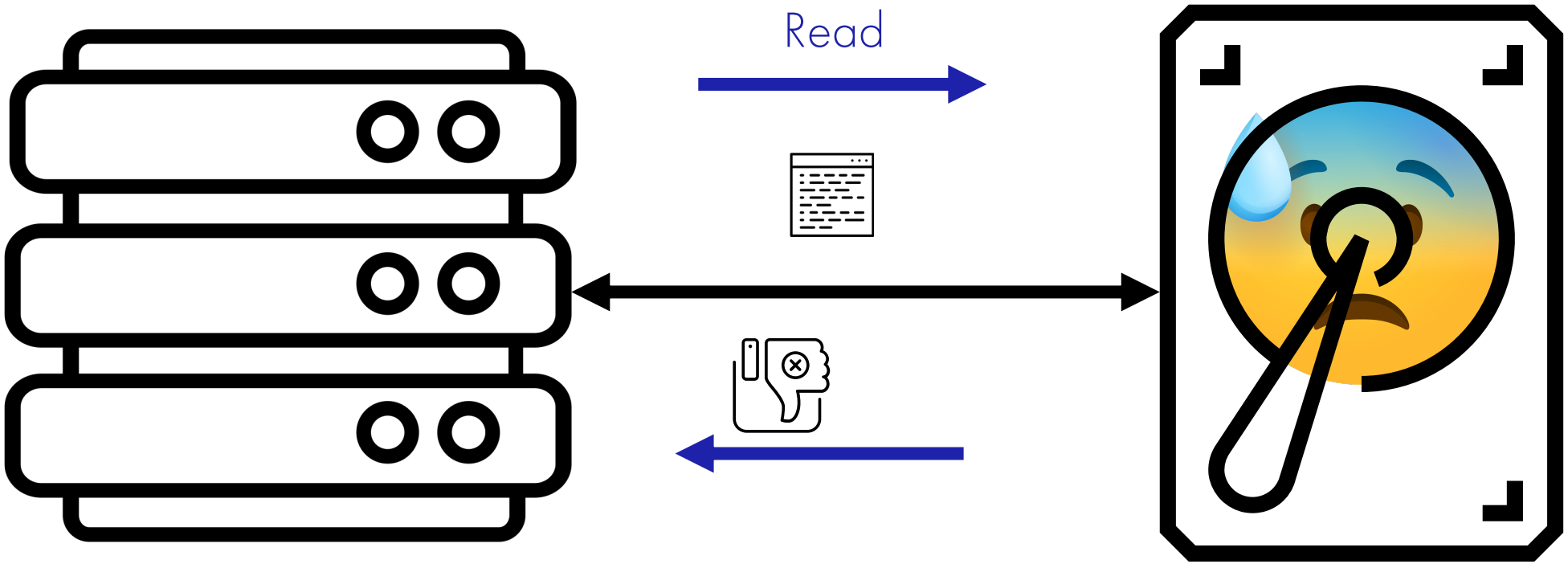


Сервер

Диск

(Иногда больше одного)

Самое точное схематичное представление СХД



Сервер

Диск

(Иногда больше одного)

Акт 1

Команды

```
struct Command {
      virtual void execute() = 0;
      virtual ~Command() = default;
      /* ... */
};
```

Команды

```
class ErrorCommand : public CommandDelegate {
      void setError();

public:
      void execute() override {
            setError();
            // skip actual command execution
      }
      /* ... */
};
```

Команды

```
class ReadCorruptCommand : public ReadCommandDelegate {
      void corruptData();

public:
      void execute() override {
            // Do actual read
            command_->execute();
            // Corrupt data
            corruptData();
      }
      /* ... */
};
```

Команды

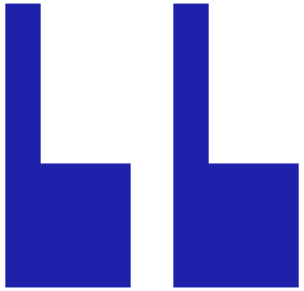
```
class DelayCommand : public CommandDelegate {
      std::chrono::milliseconds delay_;

public:
      void execute() override {
            std::this_thread::sleep_for(delay_);
            command_->execute();
      }
      /* ... */
};
```

Команды

```
class DelayCommand : public CommandDelegate {
    std::chrono::milliseconds delay_;

public:
    void execute() override {
        std::this_thread::sleep_for(delay_);
        command_->execute();
    }
    /* ... */
};
```

Чаще всего блокировать поток – не лучшая идея

Что обещали



Корутины это как синхронные функции, только асинхронные

Любое введение в корутины (с)

Что обещали, то и получили (почти)

```
struct Command {
      virtual void execute() = 0;
      virtual ~Command() = default;
      /* ... */
};
```

Что обещали, то и получили (почти)

```
struct Command {
      virtual awaitable<void> execute() = 0;
      virtual ~Command() = default;
      /* ... */
};
```

Что обещали, то и получили (почти)

```
class ReadCorruptCommand : public ReadCommandDelegate {
     void corruptData();

public:
     void execute() override {
          // Do actual read
          command_->execute();
          // Corrupt data
          corruptData();
     }
     /* ... */
};
```

Что обещали, то и получили (почти)

```
class ReadCorruptCommand : public ReadCommandDelegate {
      void corruptData();

public:
      awaitable<void> execute() override {
            // Do actual read
            co_await command_->execute();
            // Corrupt data
            corruptData();
      }
      /* ... */
};
```

Что обещали, то и получили (почти)

```
class ErrorCommand : public CommandDelegate {
      void setError();

public:
      void execute() override {
            setError();
            // skip actual command execution
      }
      /* ... */
};
```

Что обещали, то и получили (почти)

```
class ErrorCommand : public CommandDelegate {
      void setError();

public:
      awaitable<void> execute() override {
            setError();
            // skip actual command execution
            co_return;
      }
      /* ... */
};
```

Что обещали, то и получили (почти)

```
15c16,20
< void execute(Command& command) { command.execute(); }
---
> void execute(Command& command) {
>     auto ctx = io_context{1};
>     co_spawn(ctx.get_executor(), command.execute(), detached);
>     ctx.run();
> }
```

Запуск корутин

```
void execute(Command& command) {
    auto ctx = io_context{1};
    co_spawn(ctx.get_executor(), command.execute(), detached);
    ctx.run();
}
```

Запуск корутин

```
void execute(Command& command) {
    auto ctx = io_context{1};
    co_spawn(ctx.get_executor(), command.execute(), detached);
    ctx.run();
}
```

Запускаем корутину

Где выполняться
execution_context/executor

Что сделать после
completion_token

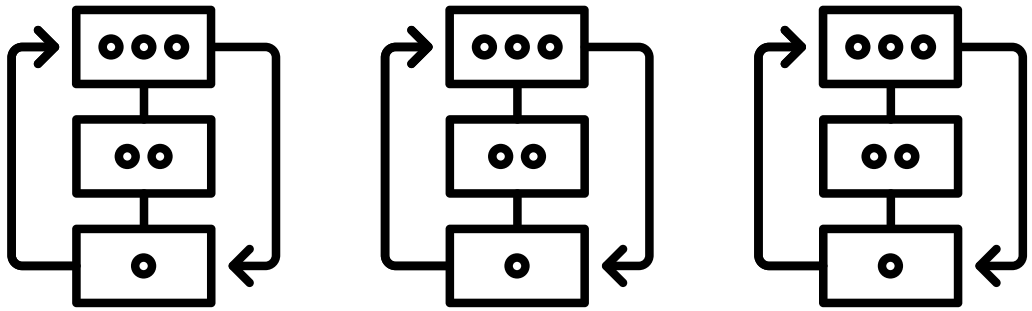
Segmentation Fault Alert

```
co_await awaitable<void>{}; // SegFault
// e.g
EXPECT_CALL(*commandMock, execute()).Times(1); // SegFault

// Use
EXPECT_CALL(*commandMock, execute()).WillOnce([] -> awaitable<void> {
      co_return;
});

// Or
testing::DefaultValue<awaitable<void>>::SetFactory(
      [] -> awaitable<void> {
            co_return;
});
EXPECT_CALL(*commandMock, execute()).Times(1);
```

Контексты и экзекьюторы

```
auto ctx = io_context{N};
```

```
ctx.run();  ctx.run();  ctx.run();
```



Thread 1 Thread 2 Thread N

- N – concurrency hint
  - 1 – выключает внутренние мьютексы

Кроме io_context:

- thread_pool
- system_executor
  - Глобальный thread_pool. Но есть нюанс ☺

Мини заключение 1

- Корутины с Asio – просто. awaitable + co_spawn
- Asio предоставляет три основных экзекьютора/контекста
  - io_context
  - thread_pool
  - system_executor

Акт 2

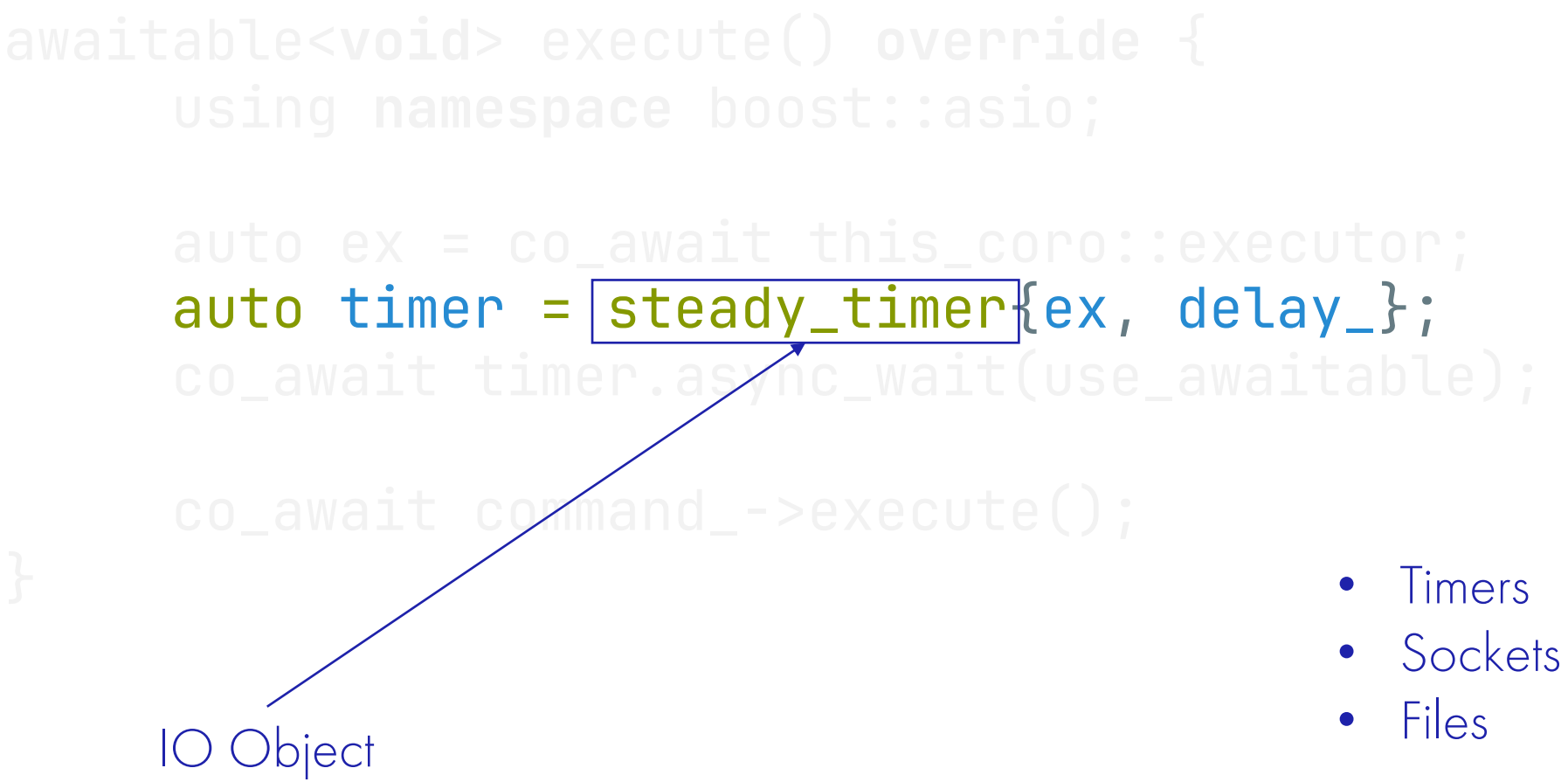
Что там с DelayCommand?

```
void execute() override {
     std::this_thread::sleep_for(delay_);
     command_->execute();
}
```

Что там с DelayCommand?

```
awaitable<void> execute() override {
      using namespace boost::asio;

      auto ex = co_await this_coro::executor;
      auto timer = steady_timer{ex, delay_};
      co_await timer.async_wait(use_awaitable);

      co_await command_->execute();
}
```

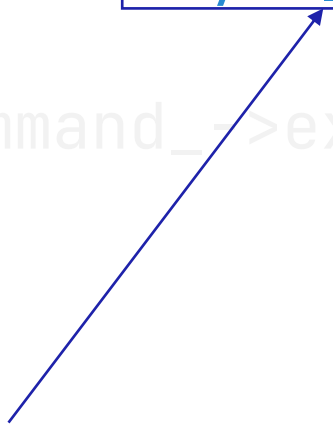
Что там с DelayCommand?



- Timers
- Sockets
- Files

Что там с DelayCommand?

```
awaitable<void> execute() override {
    using namespace boost::asio;

    auto ex = co_await this_coro::executor;
    auto timer = steady_timer{ex, delay_};
    co_await timer.async_wait(use_awaitable);

    co_await command_->execute();
}
```



- Async wait
- Async read
- Async write

this_coro::executor

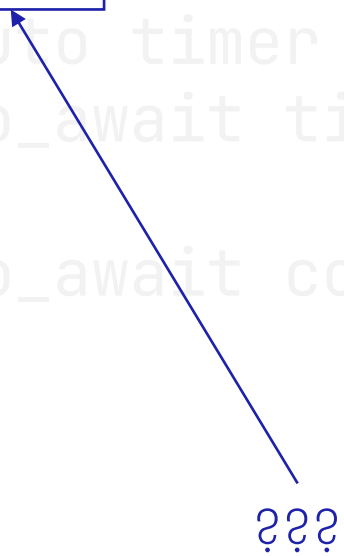
```
awaitable<void> execute() override {
    using namespace boost::asio;

    auto ex = co_await this_coro::executor;
    auto timer = steady_timer{ex, delay_};
    co_await timer.async_wait(use_awaitable);

    co_await command_->execute();
}
```

this_coro::executor

```
auto await_transform(this_coro::executor_t) noexcept
{
    struct result {
        bool await_ready() const noexcept {
            return true;
        }
        auto await_resume() const noexcept {
            return /* executor */
        }
        /* ... */
    };

    return result{this};
}
```

Что там с DelayCommand?

```
awaitable<void> execute() override {
    using namespace boost::asio;

    auto ex = co_await this_coro::executor;
    auto timer = steady_timer{ex, delay_};
    co_await timer.async_wait(use_awaitable);

    co_await command_->execute();
}
```



- io_context::executor?
- system_executor?

any_io_executor

- io_context::executor?
- system_executor?

any_io_executor

```
awaitable<void, any_io_executor> execute() override {
    using namespace boost::asio;

    any_io_executor ex = co_await this_coro::executor;
    auto timer = steady_timer{ex, delay_};
    co_await timer.async_wait(use_awaitable);

    co_await command_->execute();
}

EXPECT_EQ(ex.target<system_executor>(), nullptr);
EXPECT_EQ(*ex.target<io_context::executor_type>(),
          ctx.get_executor());
```

any_io_executor

```
awaitable<void, any_io_executor> execute() override {
    using namespace boost::asio;

    any_io_executor ex = co_await this_coro::executor;
    auto timer = steady_timer{ex, delay_};
    co_await timer.async_wait(use_awaitable);

    co_await command_->execute();
}

void foo(execution::executor auto ex) {}
```

Если позднее связываение (runtime polymorphism) Вам не подходит, можно воспользоваться концептом **execution::executor**

Мини заключение 2

- Работа с Asio это композиция асинхронных операций над I/O объектами
  - Поведение I/O объектов зависит от экзекьюторов
- Кроме базовых экзекьюторов бывают «адаптеры»
  - any_io_executor
- Asio корутины предоставляют удобный интерфейс для работы с экзекьютором в контексте которого они выполняются

Акт 3

Порядок выполнения

```
class Worker {
      virtual std::shared_ptr<Command> getNextCommand() = 0;

public:
      void work(any_io_executor ex, std::stop_token token) {
            while (!token.stop_requested()) {
                  auto command = getNextCommand();
                  auto a = command->execute();
                  co_spawn(ex, std::move(a), detached);
            }
      }
};
```

Порядок выполнения

```
class Worker {
    virtual std::shared_ptr<Command> getNextCommand() = 0;

public:
    void work(any_io_executor ex, std::stop_token token) {
        while (!token.stop_requested()) {
            auto command = getNextCommand();
            auto a = command->execute();
            co_spawn(ex, std::move(a), detached);
        }
    }
};
```

Порядок выполнения

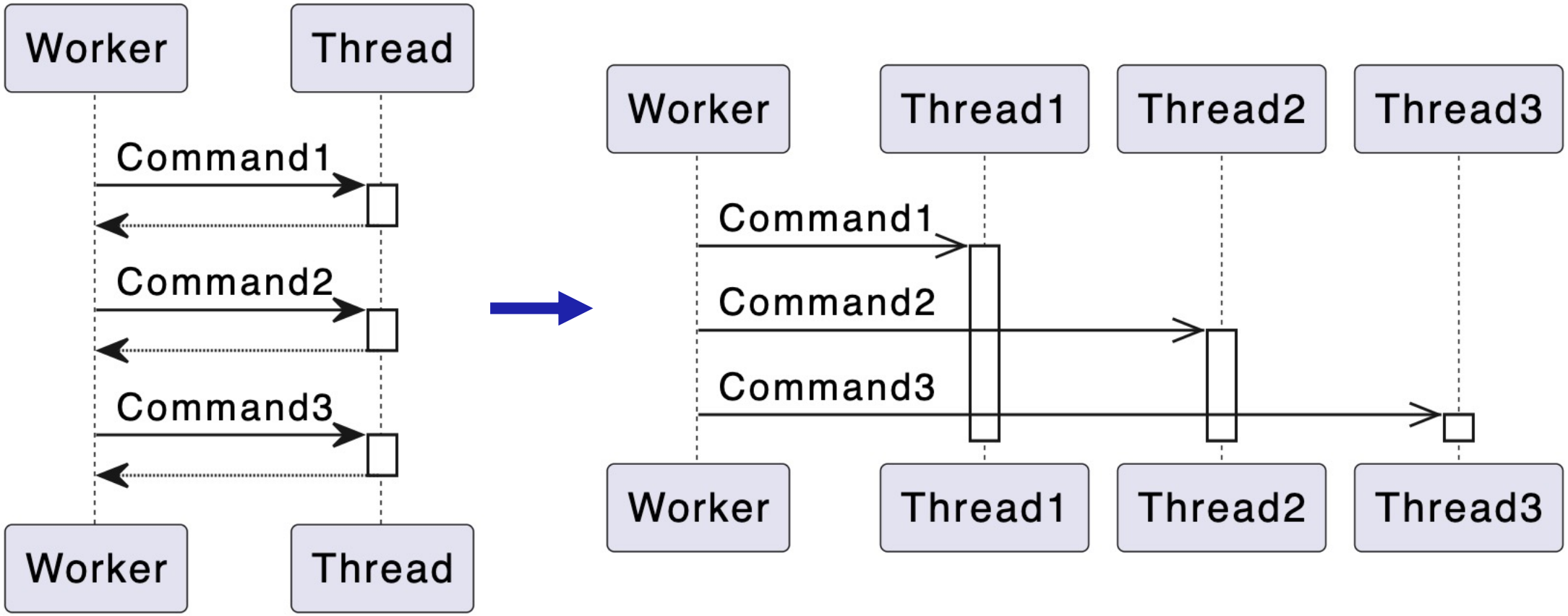
```
class Worker {
    virtual std::shared_ptr<Command> getNextCommand() = 0;

public:
    void work(any_io_executor ex, std::stop_token token) {
        while (!token.stop_requested()) {
            auto command = getNextCommand();
            auto a = command->execute();
            co_spawn(ex, std::move(a), detached);
        }
    }
};
```

Порядок выполнения

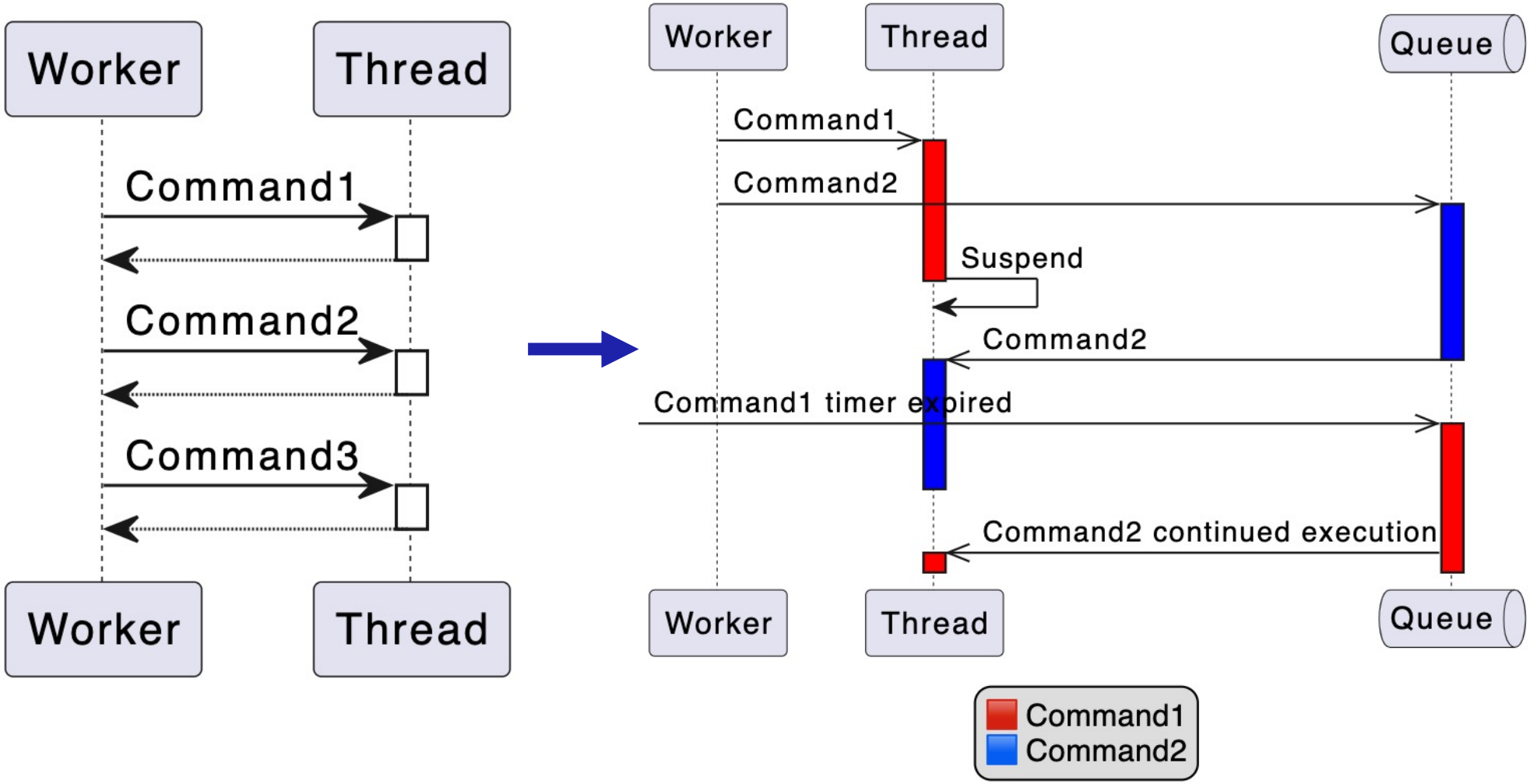
Порядок выполнения

```
class Worker {
    virtual std::shared_ptr<Command> g                    0;

public:
    void work(any_io_executor ex, std:                    ) {
        while (!token.stop_requested(
            auto command = getNextCommand();
            auto a = command->execute();
            co_spawn(ex, std::move(a), detached);
        }
    }
};
```

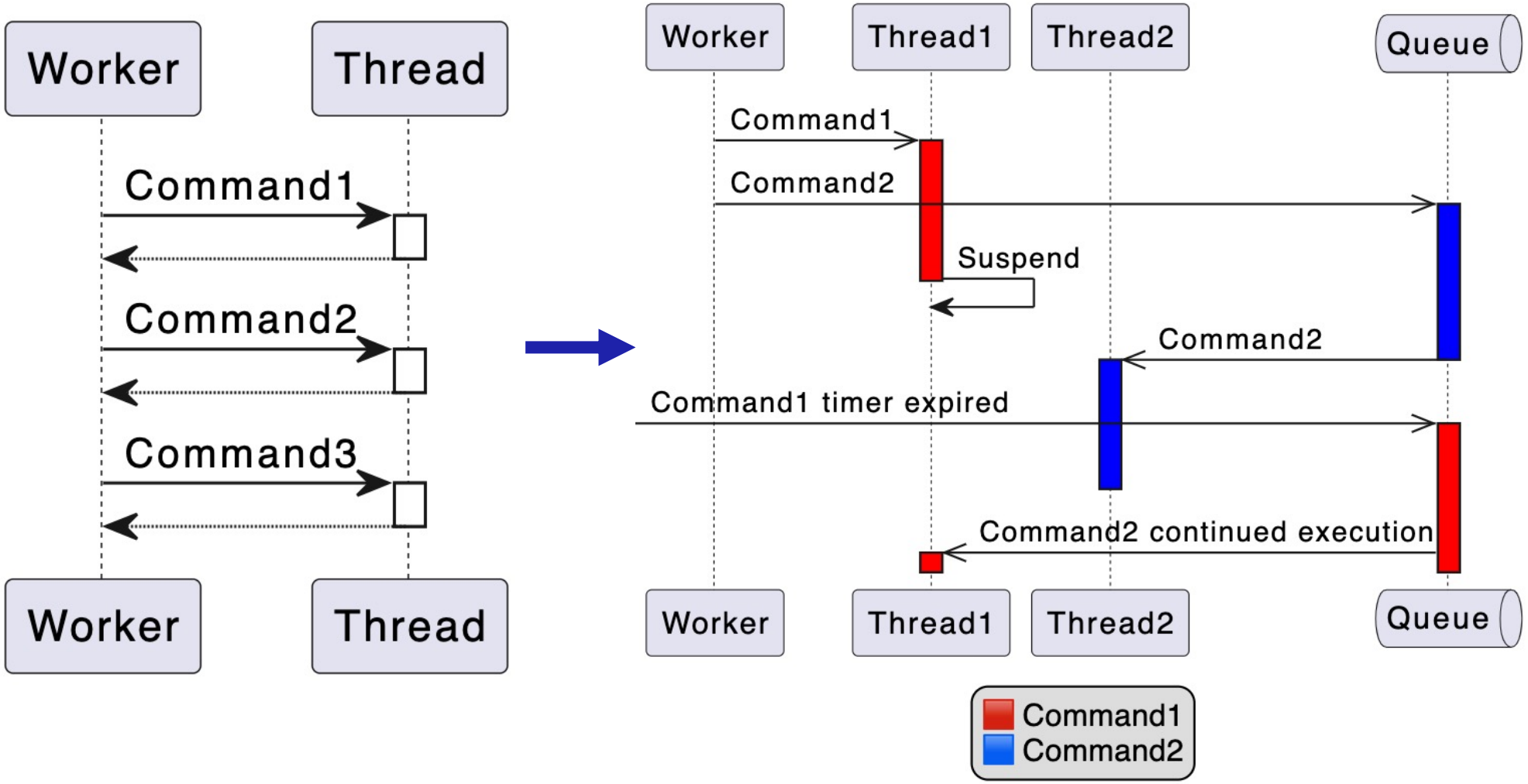
Порядок выполнения



Порядок выполнения



Порядок выполнения

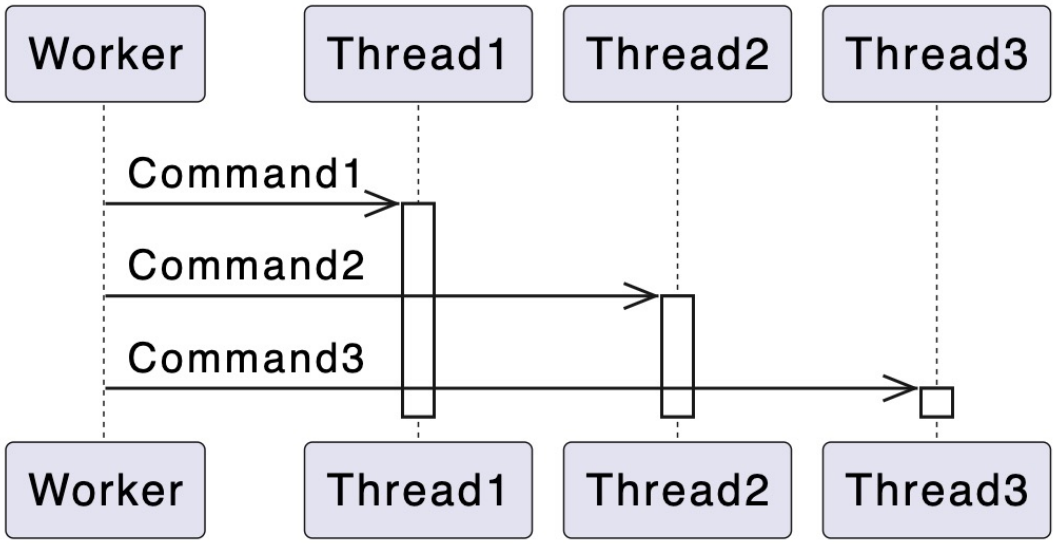


Strand

```
void work(any_io_executor ex, std::stop_token token) {
    auto samPorterBridges
        = make_strand(std::move(ex));
    while (!token.stop_requested()) {
        auto command = getNextCommand();
        auto a = command->execute();
        co_spawn(
            samPorterBridges, std::move(a), detached);
    }
}
```

Порядок выполнения

```
awaitable<void> execute() override {
      std::this_thread::sleep_for(delay_);
      co_await command_->execute();
}
```



- Command1 – 10ms
- Command2 – 10ms
- Command3 – 10ms
- Команды запускаются без задержек

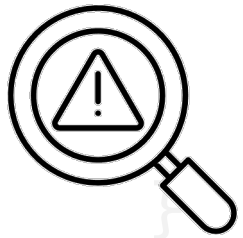
Порядок выполнения

```
{
    // Thread pool
    worker.work(pool.get_executor(), std::move(stopToken));
    f.wait();
} // ? ms
{
    // System executor
    worker.work(system_executor{}, std::move(stopToken));
    f.wait();
} // ? ms
```

Порядок выполнения

```
{
    // Thread pool
    worker.work(pool.get_executor(), std::move(stopToken));
    f.wait();
} // ~10 ms
{
    // System executor
    worker.work(system_executor{}, std::move(stopToken));
    f.wait();
} // ~30 ms
```

Порядок выполнения



```
// May block!
co_spawn(ex, std::move(a), detached);
```

Executor properties: query

```
ASSERT_EQ(
      query(system_executor{}, execution::blocking),
      execution::blocking.possibly
);
```

- `execution::blocking.possibly`
- `execution::blocking.always`
- `execution::blocking.never`

Executor properties: query

```
// Allocator
ASSERT_EQ(
      query(system_executor{}, execution::allocator),
      std::allocator<void>{}
);
```

Executor properties: query

```
// Execution context
auto ctx = io_context{};
auto ex = any_io_executor{ctx.get_executor()};
ASSERT_THAT(
      query(ex, execution::context), // execution_context&
      Ref(ctx)
);

// Thread count (not for io_context)
auto res = query(system_executor{}, execution::occupancy);
EXPECT_EQ(res, std::thread::hardware_concurrency());
```

Executor properties: require

```
void work(any_io_executor ex, std::stop_token token) {
      auto neverBlocks
            = require(ex, execution::blocking.never);
      while (!token.stop_requested()) {
            auto command = getNextCommand();
            auto a = command->execute();
            co_spawn(neverBlocks, std::move(a), detached);
      }
}
```

Executor properties: require

```
// Allocator
auto ctx = io_context{};
auto alloc = std::pmr::polymorphic_allocator{};
auto ex
     = require(ctx.get_executor(),
               execution::allocator(alloc));
```

Executor properties: require

```
// Allocator
auto ctx = io_context{};
auto alloc = std::pmr::polymorphic_allocator{};
auto ex
     = require(ctx.get_executor(),
               execution::allocator(alloc));

auto anyEx = any_io_executor{ctx.get_executor()};
auto anyExAlloc
     = require(anyEx,
               execution::allocator(alloc));
```

Executor properties: require


```
// Allocator
auto ctx = io_context{};
auto alloc = std::pmr::polymorphic_allocator{};
auto ex
     = require(ctx.get_executor(),
               execution::allocator(alloc));

auto anyEx = any_io_executor{ctx.get_executor()};
auto anyExAlloc
     = require(anyEx,
               execution::allocator(alloc));
```

Executor properties: require

```
auto ctx = io_context{};
auto ex = require(ctx.get_executor(),
     execution::outstanding_work.tracked);

ctx.run();
std::unreachable();
// ------------------------------------
// Better use make_work_guard
auto ctx = io_context{};
auto guard = make_work_guard(ctx);

ctx.run();
std::unreachable();
```

Executor properties: require

```
void work(any_io_executor ex, std::stop_token token) {
    auto neverBlocks
        = require(ex, execution::blocking.never);
    auto samPorterBridges = make_strand(neverBlocks);
    while (!token.stop_requested()) {
        auto command = getNextCommand();
        auto a = command->execute();
        co_spawn(samPorterBridges, std::move(a),
                 detached);
    }
}
```

Запрет на блокирование потока

Запрет на параллельное выполнение операций

Мини заключение 3

- Экзекьюторами можно управлять
  - make_strand
  - require
  - make_work_guard
- Свойства (property) экзекьюторов (require/prefer/query) у каждого экзекьютора свои
  - any_io_executor не имеет свойства allocator
  - io_context не имеет свойства occupancy
  - ...

Акт 4

Completion Tokens

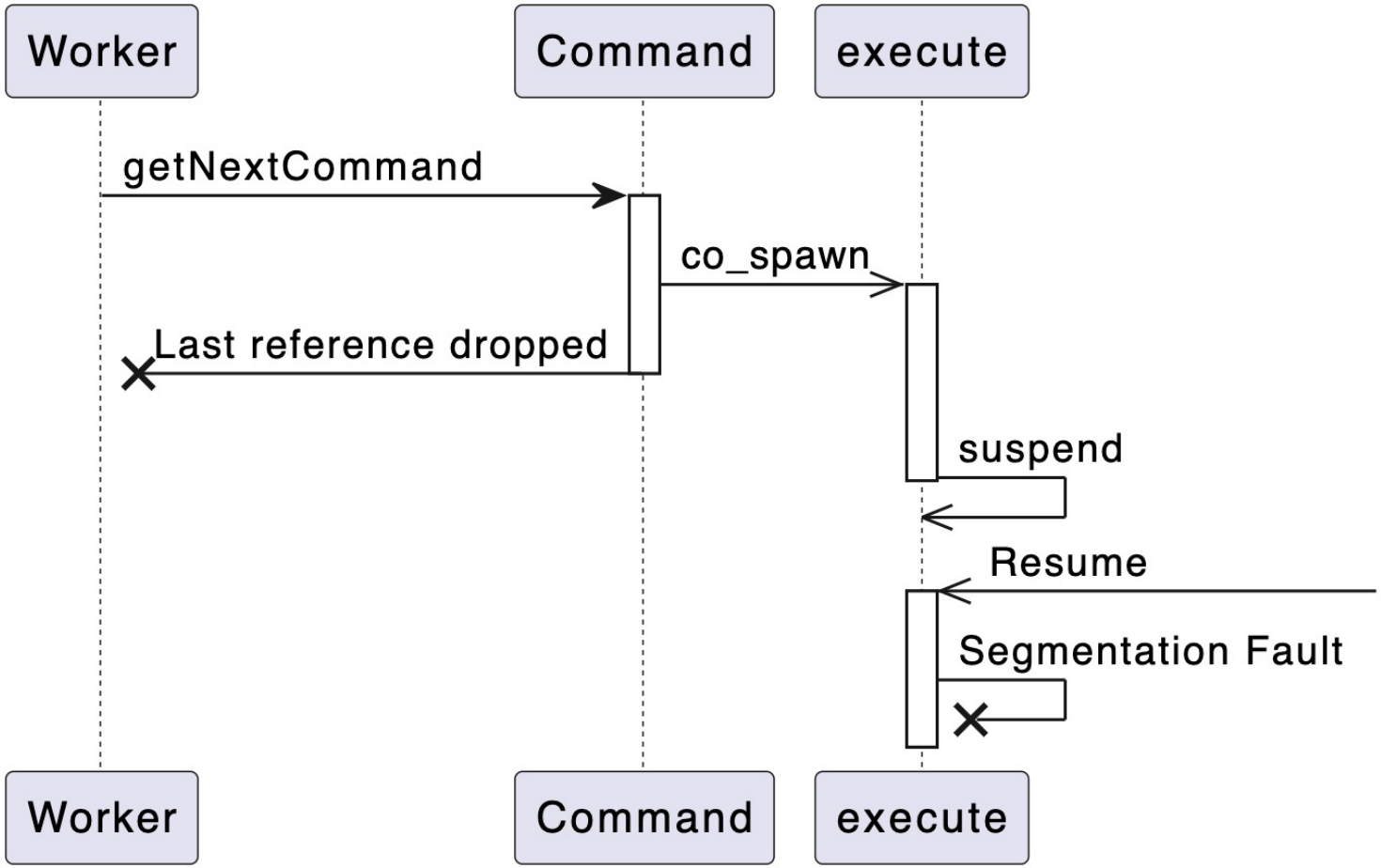
```
auto command1 = std::make_shared<DelayCommand>(commandMock,
                                               command1Delay);
EXPECT_CALL(worker, getNextCommand())
    .WillOnce(Return(command1))
```

+ std::move

```
auto command1 = std::make_shared<DelayCommand>(commandMock,
                                               command1Delay);
EXPECT_CALL(worker, getNextCommand())
    .WillOnce(Return(std::move(command1)))
```

= Segmentation Fault 🤗

Completion tokens



Если корутина является методом это не дает никаких гарантий относительно времени жизни объекта у которого вызвали этот метод

Completion tokens: lambdas

```
auto command = getNextCommand();
auto a = command->execute();
co_spawn(samPorterBridges, std::move(a),
      [command](...) {});
```

Completion tokens: lambdas

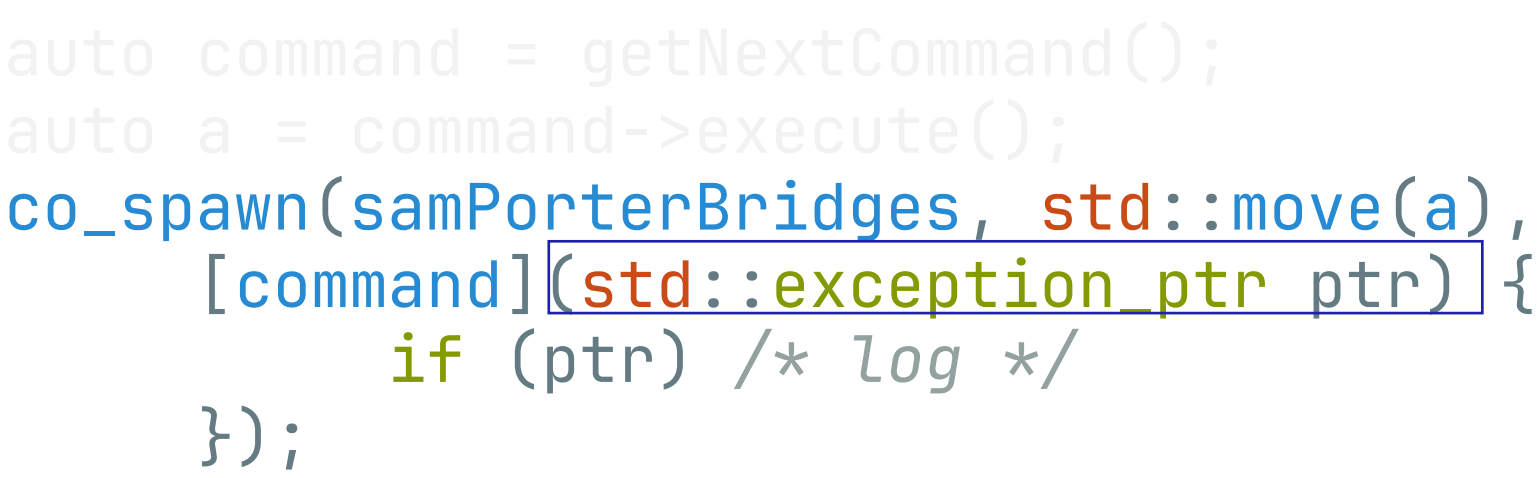
```
auto command = getNextCommand();
auto a = command->execute();
co_spawn(samPorterBridges, std::move(a),
     consign(detached, command));
```

Completion tokens: Completion Signature

```
EXPECT_CALL(*commandMock, execute())
      .WillOnce([] -> awaitable<void> {
            throw std::runtime_error{"Command error"};
            co_return;
      })
```

Исключение будет потеряно

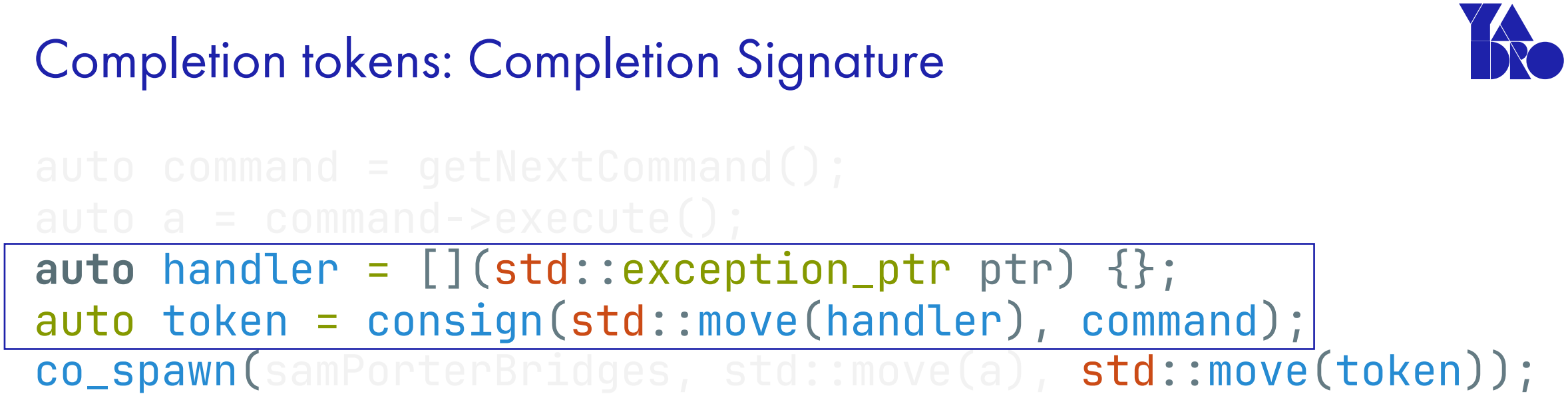
Completion tokens: Completion Signature



```
auto command = getNextCommand();
auto a = command->execute();
co_spawn(samPorterBridges, std::move(a),
      [command](std::exception_ptr ptr) {
            if (ptr) /* log */
      });
```

- Результат выполнения асинхронной операции передается в CompletionHandler
  - Параметры зависят от CompletionSignature
- Для awaitable<void> будет передан лишь указатель на исключение
  - bool(ptr) == false если исключения не произошло
  - CompletionSignature void(std::exception_ptr)
- К примеру, для awaitable<int> CompletionSignature == void(std::exception_ptr, int)

Completion tokens: Completion Signature



```
auto command = getNextCommand();
auto a = command->execute();
auto handler = [](std::exception_ptr ptr) {};
auto token = consign(std::move(handler), command);
co_spawn(samPorterBridges, std::move(a), std::move(token));
```

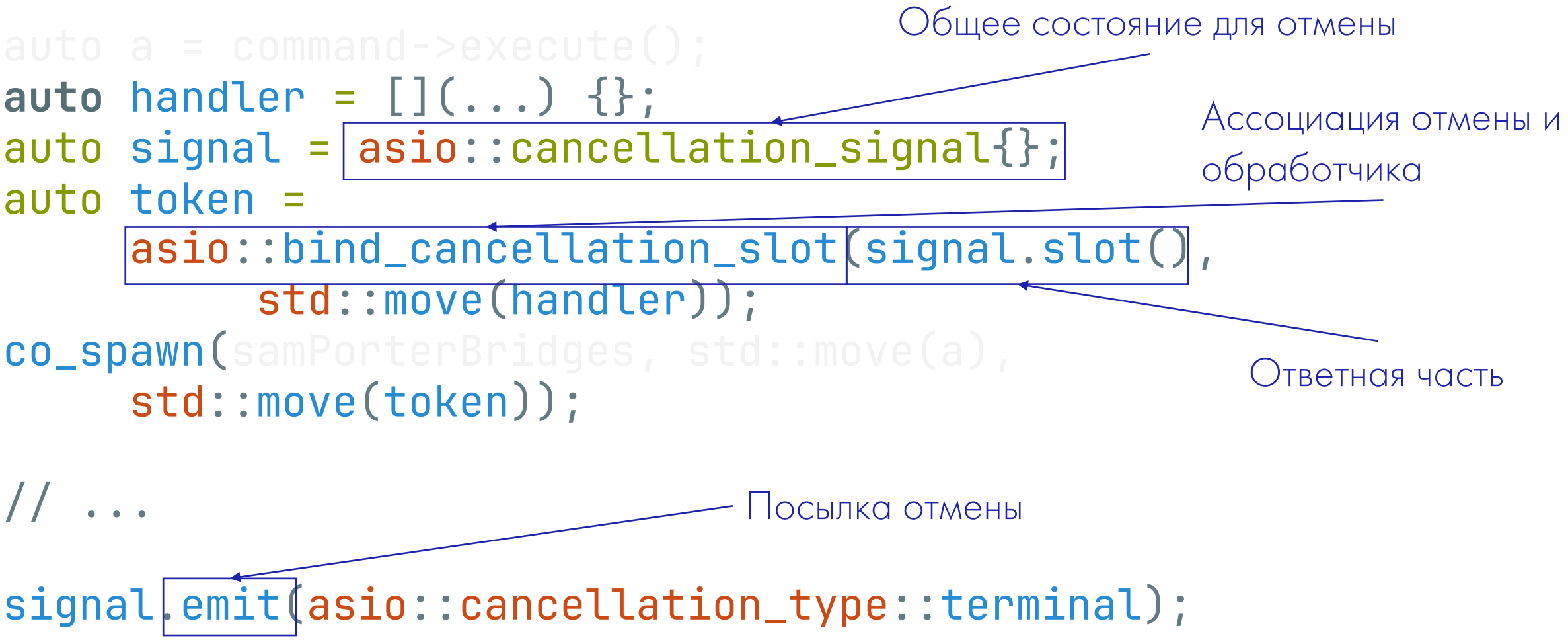
- consign также может работать и с lambda функциями
- И с обычными функциями
- И с другими токенами

Completion tokens: append, prepend

```
auto command = getNextCommand();
auto a = command->execute();
auto handler = [](std::exception_ptr ptr,
     std::shared_ptr<Command> command) {};
auto token = append(std::move(handler), command);
co_spawn(samPorterBridges, std::move(a), std::move(token));
```

Для того чтобы получить доступ к значению в CopmletionHandeler можно использовать append или prepend

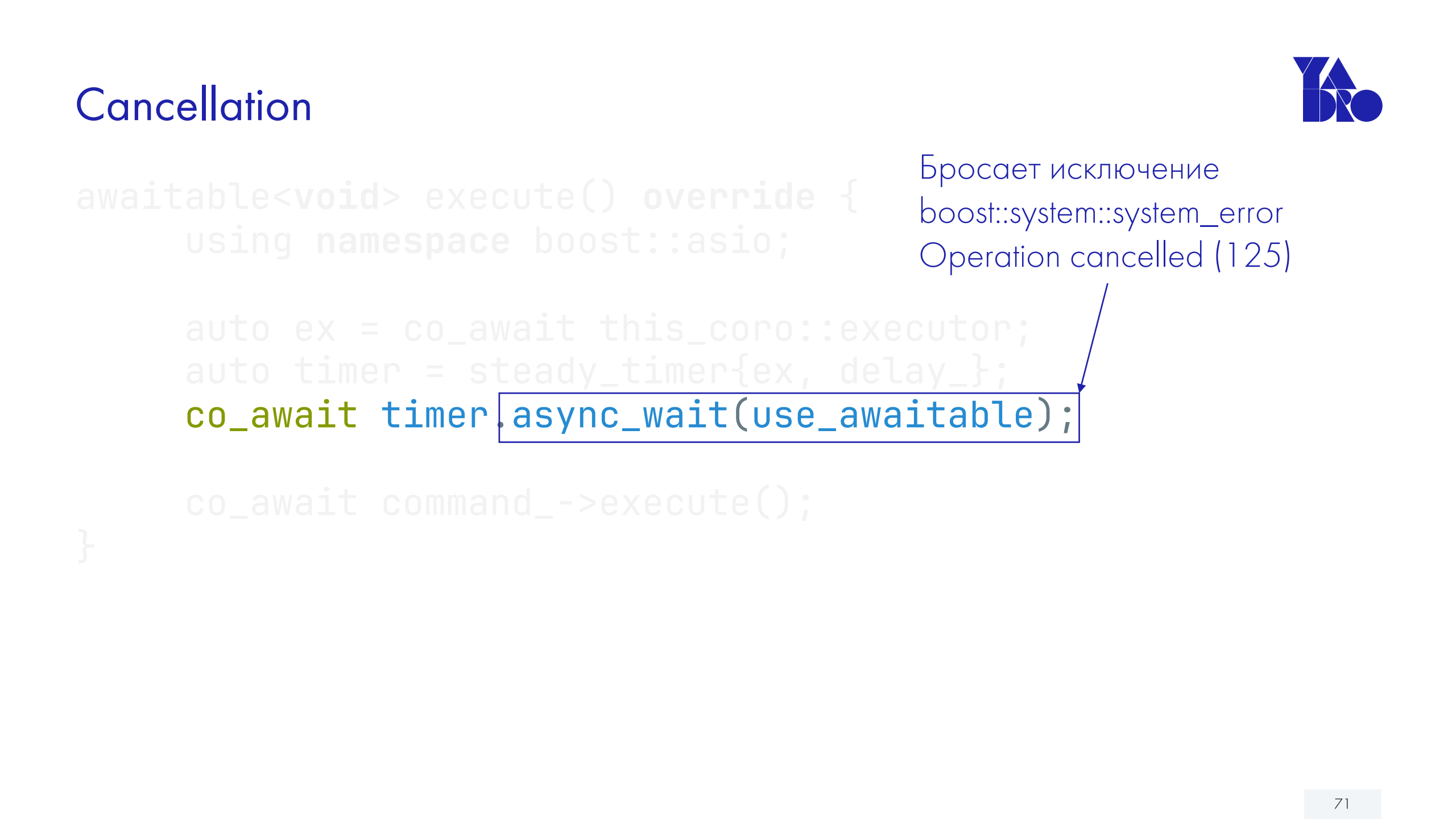
Cancellation



Cancellation

```
awaitable<void> execute() override {
      using namespace boost::asio;

      auto ex = co_await this_coro::executor;
      auto timer = steady_timer{ex, delay_};
      co_await timer.async_wait(use_awaitable);

      co_await command_->execute();
}
```

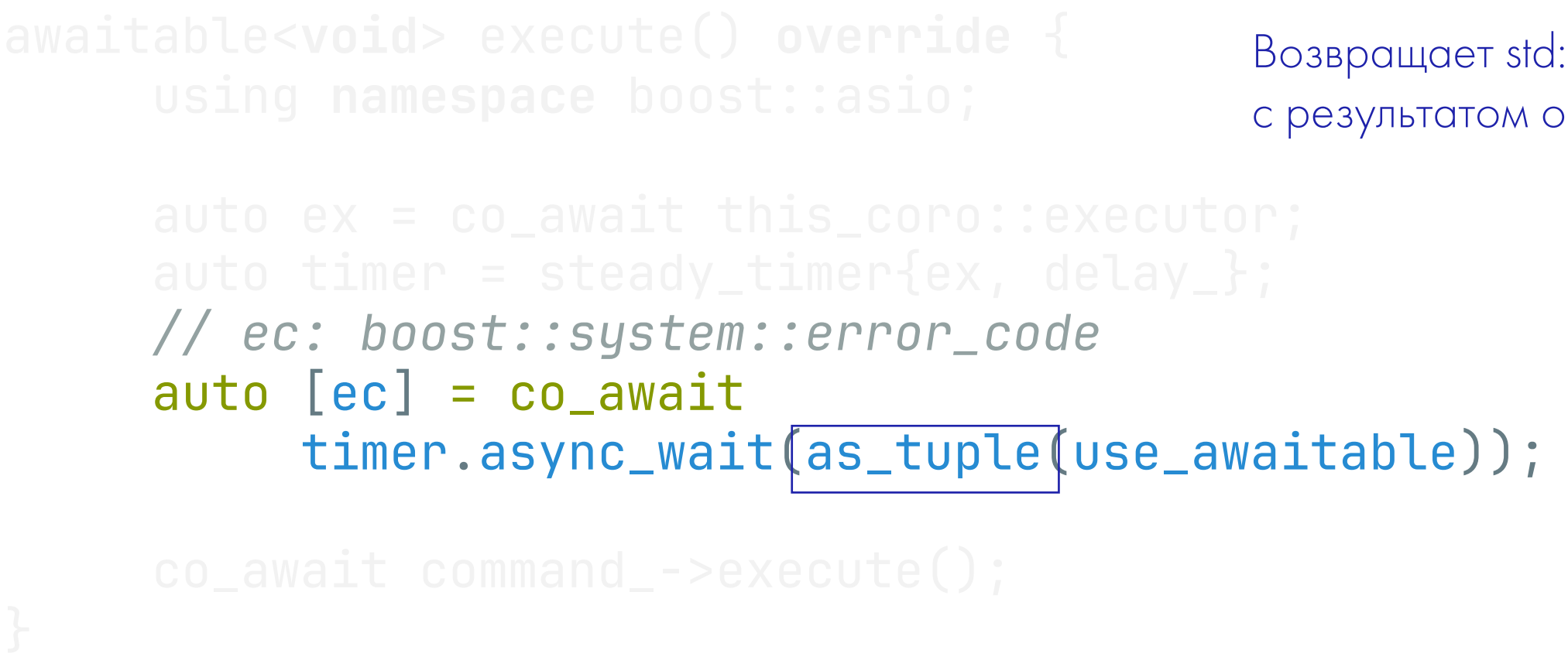
Cancellation



```
awaitable<void> execute() override {
    using namespace boost::asio;

    auto ex = co_await this_coro::executor;
    auto timer = steady_timer{ex, delay_};
    co_await timer.async_wait(use_awaitable);

    co_await command_->execute();
}
```

Бросает исключение
boost::system::system_error
Operation cancelled (125)

Cancellation



```
awaitable<void> execute() override {
    using namespace boost::asio;

    auto ex = co_await this_coro::executor;
    auto timer = steady_timer{ex, delay_};
    // ec: boost::system::error_code
    auto [ec] = co_await
        timer.async_wait(as_tuple(use_awaitable));

    co_await command_->execute();
}
```

Возвращает std::tuple
с результатом операции

- Если бы это был async_write то as_tuple вернул бы еще количество прочитанных байт кроме кода ошибки **auto [ec, len]**

Allocations

Аллоцирует дважды:
- execution_context
- io_context(int)

```
auto ctx = asio::io_context{1};
```

```
auto a = command->execute();
```

Аллоцирует awaitable_frame.
awaitable_frame == std::coroutine_traits<awaitable>::promise_type

```
co_spawn(ctx, std::move(a), detached);
```

- Аллоцирует awaitable_frame<coro_entry_point>
- Аллоцирует место для CompletionHandler (detached)
- Аллоцирует место для executor_op (единица исполнения)

Allocations

```
auto ctx = asio::io_context{1};

auto a = command->execute();

co_spawn(ctx, std::move(a), detached);
```

Аллоцирует дважды:
- execution_context
- io_context(int)

Аллоцирует awaitable_frame.
awaitable_frame == std::coroutine_traits<awaitable>::promise_type

- Аллоцирует awaitable_frame<coro_entry_point> ← Не избежать
- Аллоцирует место для CompletionHandler (detached)
- Аллоцирует место для executor_op (единица исполнения)

Allocator binding

```
auto ctx = io_context{};
auto ex = require(ctx.get_executor(),
     execution::allocator(exAlloc));
auto token = bind_allocator(handlerAlloc, detached);

auto a = command->execute();
co_spawn(ex, std::move(a), std::move(token));
```

Мини заключение 4

- Не важно, Asio или нет, корутина-метод не гарантирует время жизни объекта
  - У Asio для этого есть специальный CompletionToken
- Управлять асинхронными операциями позволяет CompletionToken
  - use_future
  - use_awaitable
  - lambda
  - ...
- CompletionToken тоже можно «адаптировать»
  - consign/append/prepend
  - as_tuple
- CompletionToken можно ассоциировать с аллокаторами/экзекьюторами
- Отмена корутины значит отмена следующей асинхронной операции в корутине

Источники

- Документация Asio
  - https://think-async.com/Asio/
- Статьи про корутины
  - https://lewissbaker.github.io/
- Про cancellation в Asio
  - https://www.youtube.com/@talkingasync4596
- Кастомизируем Asio
  - https://youtu.be/QAdp-icrvos

YADRO
yadro.com