

Яндекс



# Асинхронность и сопрограммы

Григорий Демченко, разработчик Y.T, 2017

# План

- Введение
- Сопрограммы
- Планировщики
- Синхронизация
- Примеры
- Заключение

# Введение



# Синхронный многопоточный сервер

```
void multithreadedServer() {  
    Acceptor acceptor{80};           // слушаем 80 порт  
    while (true) {  
        auto socket = acceptor.accept(); // 1-й блокирующий вызов  
        thread([socket] {  
            auto request = socket.read(); // 2-й блокирующий вызов  
            auto response = handleRequest(request);  
            socket.write(response);       // 3-й блокирующий вызов  
        }).detach();  
    }  
}
```

# Асинхронный сервер

```
void asynchronousServer() {  
    Acceptor acceptor{80};  
    acceptor.onAccept([] (auto socket) {           // 1-й асинхронный вызов  
        socket.onRead([socket](auto request) {    // 2-й асинхронный вызов  
            auto response = handleRequest(request);  
            socket.onWrite(response, []() {        // 3-й асинхронный вызов  
                // завершение  
            });  
        });  
        // продолжаем принимать соединения...  
    });  
}
```



# Реальный асинхронный сервер

```
void realAsynchronousServer() {
    Acceptor acceptor{80};
    Handler accepting = [&acceptor, &accepting] {
        struct Connection {
            explicit Connection(Socket sock) : socket(std::move(sock)) {}

            void onConnect() {
                if (error)
                    return onError(error);
                socket.onRead([this](const Request& request, const Error& error) {
                    if (error)
                        return onError(error);
                    response = handlerRequest(request);
                    socket.onWrite(response, [this](const Error& error) {
                        if (error)
                            return onError(error);
                        onDone();
                    });
                });
            }

private:
            void onError(const Error& error) {
                delete this;
            }

            void onDone() {
                delete this;
            }

            Response response;
            Socket socket;
        };

        acceptor.onAccept([&accepting](Socket socket, const Error& error) {
            if (!error) {
                (new Connection(socket))->onConnect();
                accepting();
            }
        });
    };
}
```

## Сравните:

```
void asynchronousServer() {
    Acceptor acceptor{80};
    acceptor.onAccept([] (socket) {
        socket.onRead([socket](request) {
            response = handlerRequest(request);
            socket.onWrite(response, []() {
                // завершение
            });
        });
        // продолжаем принимать соединения
    });
}
```

# Асинхронный сервер: обсуждение

## Плюсы:

- › Производительность
- › Автоматическая параллелизация исполнения

## Минусы:


- › Сложность:

1. Нелинейный рост сложности и проблем
2. Явная передача контекста исполнения
3. Обработка ошибок
4. Время жизни объектов
5. Отладка



# Что бы хотелось?

- Использовать **эффективность** асинхронного подхода
- Использовать **простоту** синхронного подхода



**Решение: использовать  
сопрограммы**

Сопрограммы



# Сопрограммы

- *Подпрограмма*: результат доступен сразу после завершения.
- *Сопрограмма*: результат будет доступен позже. Точка вызова и точка получения результата разнесены

Примеры сопрограмм:

- Генераторы на языке Python.
- Async/await C++ proposal.

Метод сохранения контекста исполнения:

- *Stackful сопрограммы*.

# Реализация сопрограмм

## Проблема

*Сопрограммы* пока еще не являются частью языка  
Необходимо применять низкоуровневые примитивы.

Решение: `boost.context`:

Эффективное решение: использование ассемблера  
Десктопные платформы: Windows, Linux, MacOSX  
Мобильные платформы: Windows Phone, Android, IOS  
Процессоры: x86, ARM, Spark, Spark64, PPC, PPC64, MIPS  
Компиляторы: GCC, Clang, Visual Studio



# boost.context

Работа с контекстом:

- `make_fcontext`: создает контекст

- `jump_fcontext`: переключает контекст

Контекст исполнения:

- `execution_context<Args...>`: создает контекст исполнения

Конструктор принимает сигнатуру:

- `execution_context(execution_context ctx, Args... args)`

# Сопрограммы в действии

```
void coroFun() {  
    log << "2";  
    yield();  
    log << "4";  
}
```

```
log << "1";  
Coro c{coroFun};  
log << "3";  
c.resume();  
log << "5";
```

Вывод:

12345

# Использование сопрограммы

Преобразовать асинхронный вызов:

```
async(completionCallback);
```

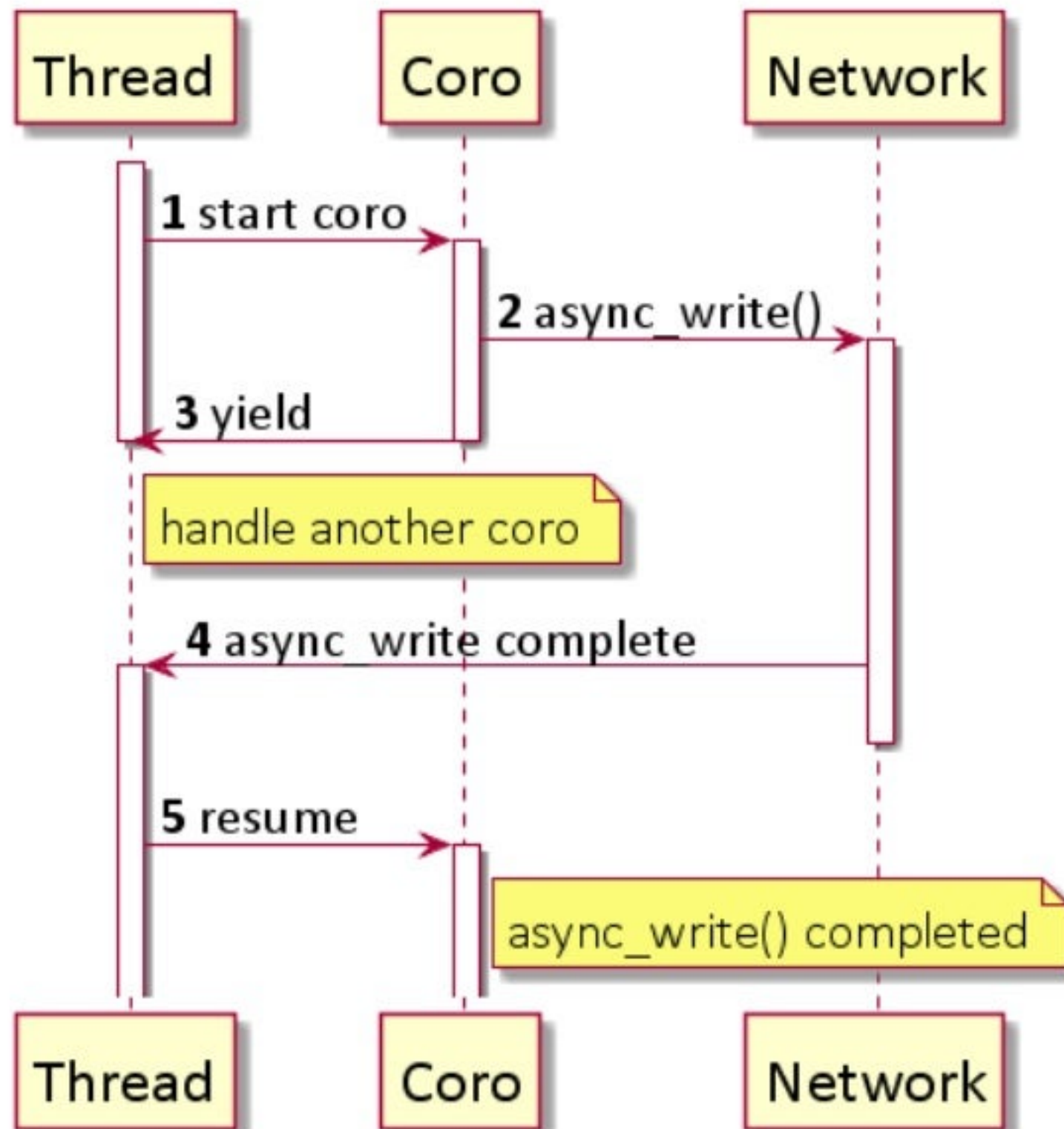
В синхронный:

```
synca(); // async -> synca
```

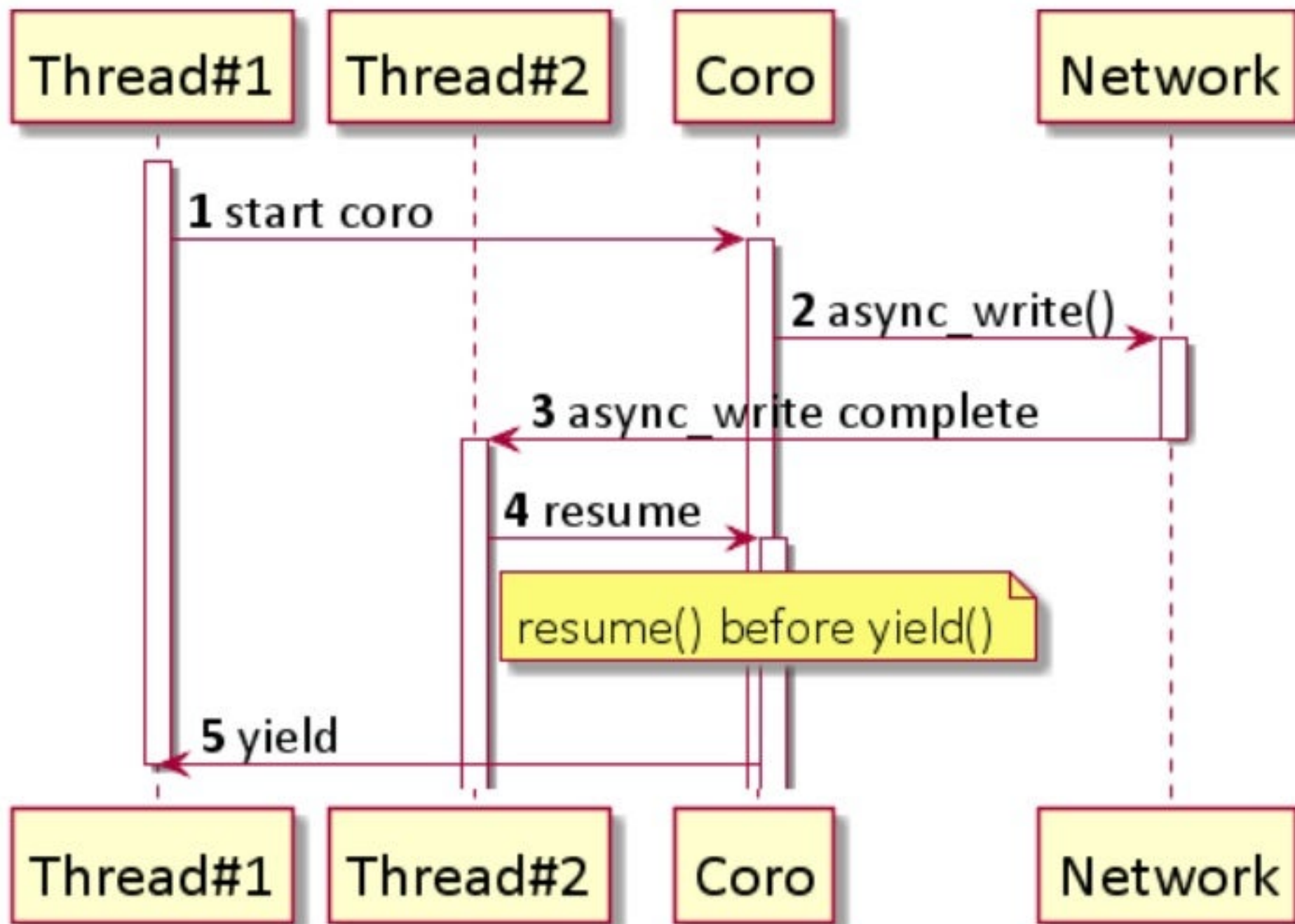
Используя следующий подход:

```
void synca() {  
    auto& coro = currentCoro();  
    async([&coro] {  
        coro.resume();  
    });  
    yield();  
}
```

# Sequence Diagram



# Проблема: состояние гонки





# Возможные решения

- `std::mutex`

- `boost::asio::strand`

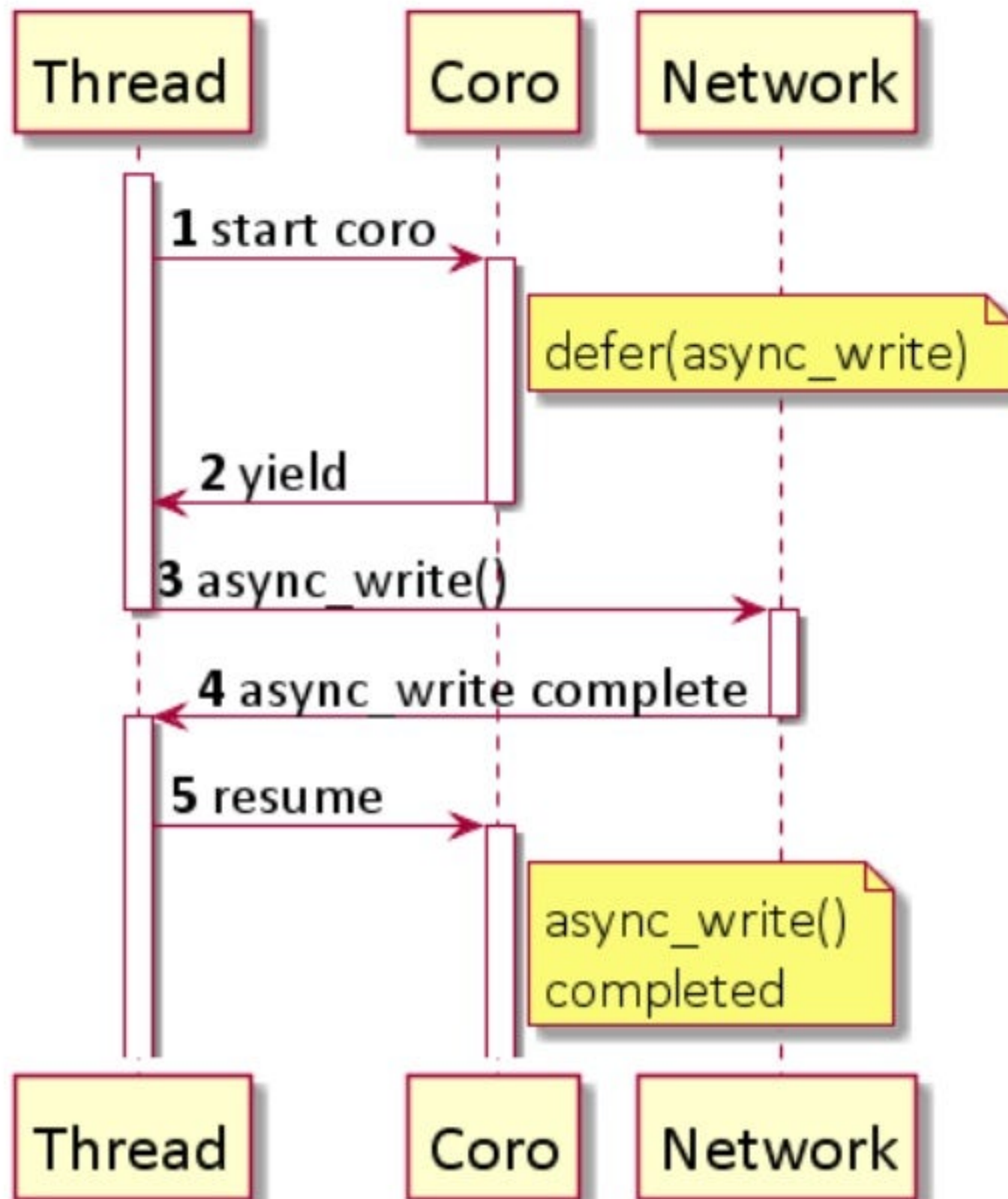
- `defer`

# Использование defer

■ defer откладывает асинхронный вызов:

```
using Handler = std::function<void()>;  
void defer(Handler handler);
```

# Решение: defer



# Использование: defer

```
void synca() {  
    auto& coro = currentCoro();  
    async([&coro] {  
        coro.resume();  
    });  
    yield();  
}
```



```
void synca() {  
    auto& coro = currentCoro();  
    defer([&coro] {  
        async([&coro] {  
            coro.resume();  
        });  
    });  
}
```

# Абстрагирование от сопрограмм

```
void synca() {  
    auto& coro = currentCoro();  
    defer([&coro] {  
        async([&coro] {  
            coro.resume();  
        });  
    });  
}
```



```
void synca() {  
    deferProceed([](Handler proceed) {  
        async(proceed);  
    });  
}
```



# Synca сервер

```
void syncaServer() {  
    Acceptor acceptor{80};  
    while (true) {  
        auto socket = acceptor.accept();  
        go([socket] { // стартуем новую программу  
            auto request = socket.read();  
            auto response = handleRequest(request);  
            socket.write(response);  
        });  
    }  
}
```

Планировщики



# Планировщик

Интерфейс планировщика:

```
struct IScheduler {  
    virtual void schedule(Handler) = 0;  
};
```

Планировщик запускает обработчики.

# Пул потоков

```
struct ThreadPool : IScheduler {  
    explicit ThreadPool(size_t threads);  
  
    void schedule(Handler handler) {  
        service.post(std::move(handler));  
    }  
private:  
    boost::asio::io_service service;  
};
```

# Сопрограммы и планировщик

```
struct Journey {  
    void defer(Handler handler) {  
        deferHandler = std::move(handler);  
        yield();  
    }  
    void proceed() {  
        scheduler->schedule([this] {  
            coro.resume();  
            deferHandler();  
        });  
    }  
private:  
    IScheduler* scheduler;  
    Coro coro;  
    Handler deferHandler;  
};
```



# Телепортация

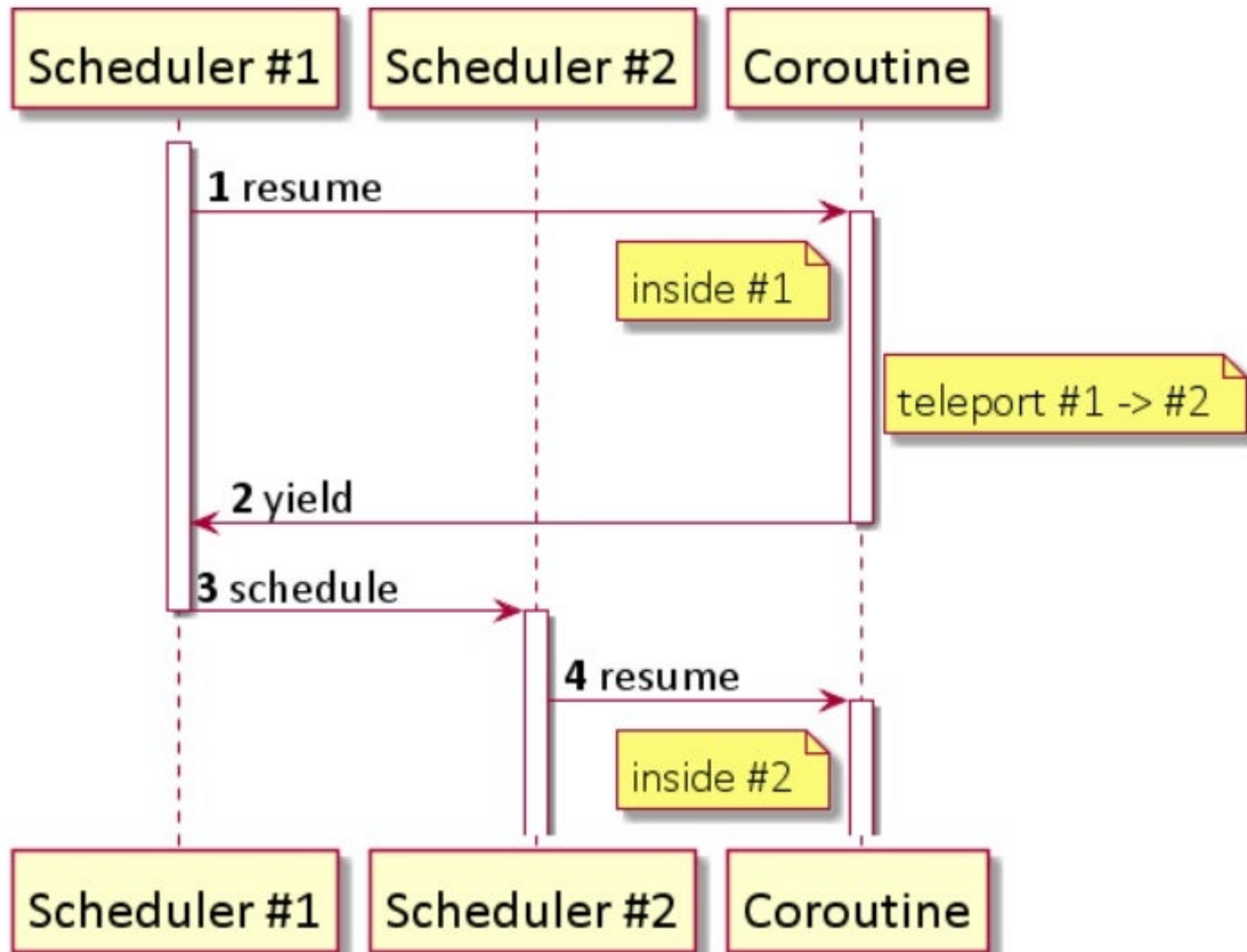
Давайте перепрыгнем на другой планировщик:

```
go([&] {  
    log << "Inside 1st thread pool";  
    teleport(tp2);  
    log << "Inside 2nd thread pool";  
}, tp1);
```

Реализация:

```
void Journey::teleport(IScheduler& s) {  
    scheduler = &s;  
    defer([this] {  
        proceed();  
    });  
}
```

# Телепортация: Sequence Diagram



# Порталы


Портал – это RAII телепортация:

```
struct Portal {  
    Portal(IScheduler& destination)  
        : source{currentScheduler()} {  
        teleport(destination);  
    }  
    ~Portal() {  
        teleport(source);  
    }  
private:  
    IScheduler& source;  
};
```



# Использование порталов

```
struct Network {  
    void handleNetworkEvents() {  
        // действия внутри сетевого пула потоков  
    }  
};  
  
ThreadPool commonPool{4};  
ThreadPool networkPool{2};  
  
portal<Network>().attach(networkPool);  
go([] {  
    log << "Inside common pool";  
    portal<Network>()->handleNetworkEvents();  
    log << "Inside common pool";  
}, commonPool);
```



**Портал – абстракция  
среды исполнения**



Синхронизация



# Alone

```
struct Alone : IScheduler {  
    void schedule(Handler handler) {  
        strand.post(std::move(handler));  
    }  
private:  
    boost::asio::io_service::strand strand;  
};
```

strand гарантирует, что ни один обработчик не будет запущен параллельно с другим

**Alone – это  
неблокирующая  
синхронизация  
без дедлоков**

Примеры



# Интегральный пример: инициализация

```
struct DiskCache/MemCache {  
    optional<string> get(const string& key);  
    void set(const string& key, const string& val);  
};  
struct Network {  
    string performRequest(const string& key);  
};
```

```
ThreadPool commonPool{3};           //    общий пул операций  
ThreadPool diskPool{2};              //    дисковый пул операций  
ThreadPool netPool{1};               //    сетевой пул операций  
Alone memAlone{commonPool};          //    MemCache синхронизация
```

```
portal<DiskCache>().attach(diskPool);  
portal<MemCache>().attach(memAlone);  
portal<Network>().attach(netPool);
```



# Интегральный пример: получение значения

```
string obtainValue(const string& key) {  
    auto res = portal<MemCache>()->get(key);  
    if (res)  
        return *res;  
    res = portal<DiskCache>()->get(key);  
    if (res)  
        return *res;  
    auto val = portal<Network>()->performRequest(key);  
    go([key, val] {  
        portal<MemCache>()->set(key, val);  
        portal<DiskCache>()->set(key, val);  
    });  
    return val;  
}
```

# Свойства портала

- Работает с исключениями
- Абстрагирует контекст исполнения

Заключение



# Теорема

Любая асинхронная задача может быть реализована через *сопрограммы*

1. Нет кода после вызова:

```
// код до  
async(params..., cb);  
// отсутствует код
```



```
// код до  
sync(params...);  
cb();
```

2. Есть код после вызова:

```
// код до  
async(..., cb);  
// код после
```



```
// код до  
go { async(..., cb); };  
// код после
```



```
// код до  
go { sync(...); cb(); };  
// код после
```

# Выводы

Синхронный подход: **простота**

Асинхронный подход : **эффективность**

Неблокирующая синхронизация без дедлоков

Прозрачно для использования

Работает с исключениями

Работа с вложенными таймаутами

Отмена операций

Принципиально новые подходы и паттерны

Это прикольно!



# Асинхронность и сопрограммы



Григорий Демченко, Разработчик YT



[gridem.blogspot.com](http://gridem.blogspot.com)



[github.com/gridem](https://github.com/gridem)



[habrahabr.ru/users/gridem](https://habrahabr.ru/users/gridem)



[bitbucket.org/gridem](https://bitbucket.org/gridem)