

Модульность и управляемая многопоточность.

Трудности, проблемы, решения. Пример реализации на C++.

Think of it this way: threads are like salt, not like pasta.
You like salt, I like salt, we all like salt. But we eat more pasta.
– Larry McVoy

A computer is a state machine.
Threads are for people who can't program state machines.
– Alan Cox

Конференция C++ CoreHard Spring 2017, Минск
Василий Вяжевич

Предисловие



О чем поговорим?

- Пример из жизни
 - Функция `std::async`
 - Последствия
 - Есть ли выход?
- Подходы
 - Как это уже давно работает?
- Строим архитектуру
- Выводы
- Вопросы

Пример - АСУ гаражные ворота



- Открыть/заккрыть ворота
- Световая сигнализация

Как будем “моргать”?

```
bool isDoorMoving = false;
unsigned blinkLampInterval_ms = 500;
std::future<void> blinkFuture;
void openTheDoor()
{
    startDoorMotorFwd();
    isDoorMoving = true;
    blinkFuture = std::async(std::launch::async, blinkLamp);
}
void blinkLamp()
{
    do
    {
        bool lampIsOn = getLampStatus();
        triggerLamp(!lampIsOn);
        std::this_thread::sleep_for(std::chrono::milliseconds(blinkLampInterval_ms));
    } while (isDoorMoving || lampIsOn);
}
```


Заглянем в будущее.



- Сколько потоков потребуется?
- Сколько мьютексов и семафоров?
- А вообще возможна отладка?

С чем мы можем столкнуться?

- Как производить отладку “по шагам” (debug)?
- Стек вызовов отсутствует
- Как отлаживать работу примитивов синхронизации?
- Как осуществлять модульное тестирование (unit testing)?
- Магические `wait()` и `sleep()`.
- Как часто необходимо распараллеливание?

Сложность отладки многопоточных приложений

- 2 потока - вполне возможно
- 3 потока - не просто
- 4 и более потока - практически невозможно!

Когда возникает желание использовать потоки?

- Исключение блокировки процесса при ожидании.
 - Отзывчивость графического интерфейса.
 - Сетевой обмен.
 - Ввод/вывод (файлы, устройства).
- Скорость и удобство разработки, лямбда выражения.
- Параллельные процессы или события, паузы и ожидание?

```
auto callback=[](SomeType res) {  
    m_someWorkResult = res;  
};  
std::thread myThread = std::thread([](int a, int b) { //lambda  
    SomeType res = doSomeWork(a,b);  
    callback(res);  
}, 5, 6); //parameters
```

Как это уже давно работает?

```
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR pCmdLine, int
nCmdShow) {
    ...
    while (GetMessage(&msg, NULL, , )) {
        DispatchMessage(&msg);
    }
    return (int) msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    switch(msg) {
        case WM_CLOSE:
            DestroyWindow(hwnd);
            break;
        case WM_DESTROY:
            PostQuitMessage();
            break;
        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
}
```

Алгоритм на основе событий

```
bool isDoorMoving = false;
unsigned blinkLampInterval_ms = 500;
void openTheDoor()
{
    startDoorMotorFwd();
    isDoorMoving = true;
    blinkLamp();
}
void blinkLamp()
{
    bool lampIsOn = getLampStatus();
    triggerLamp(!lampIsOn);
    if (isDoorMoving || lampIsOn) // instead of sleep()
        addEventToMessageLoop(&blinkLamp, blinkLampInterval_ms);
}
```

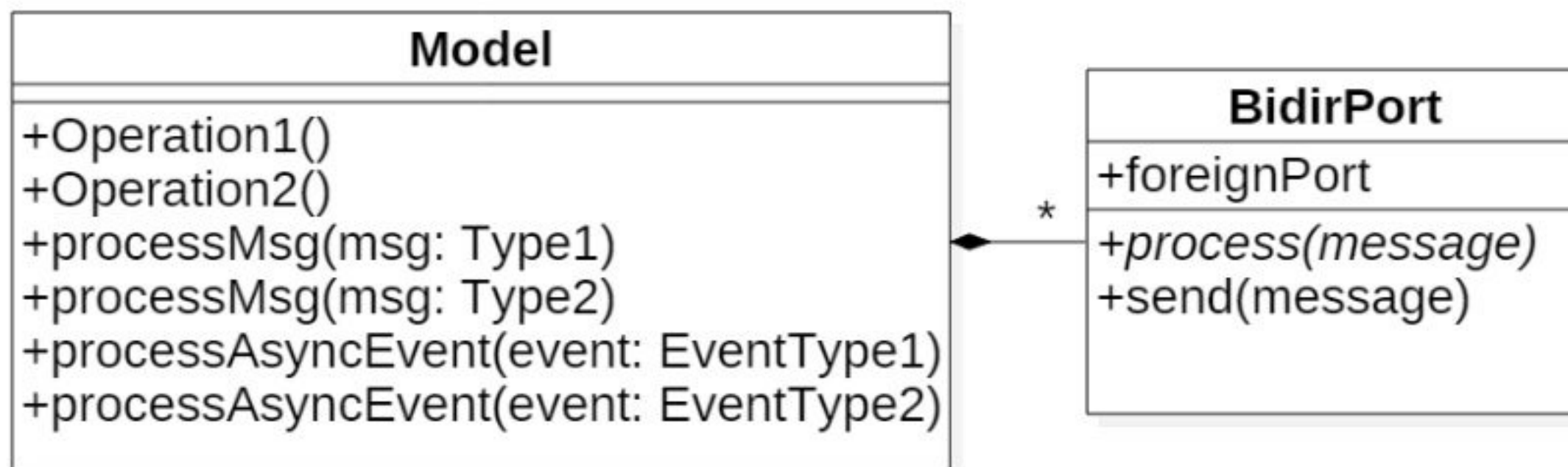
Строим архитектуру - требования

- Логика не зависит от потоков
- Возможность тестирования отдельных модулей
- Возможность распараллеливания
- Стек вызовов

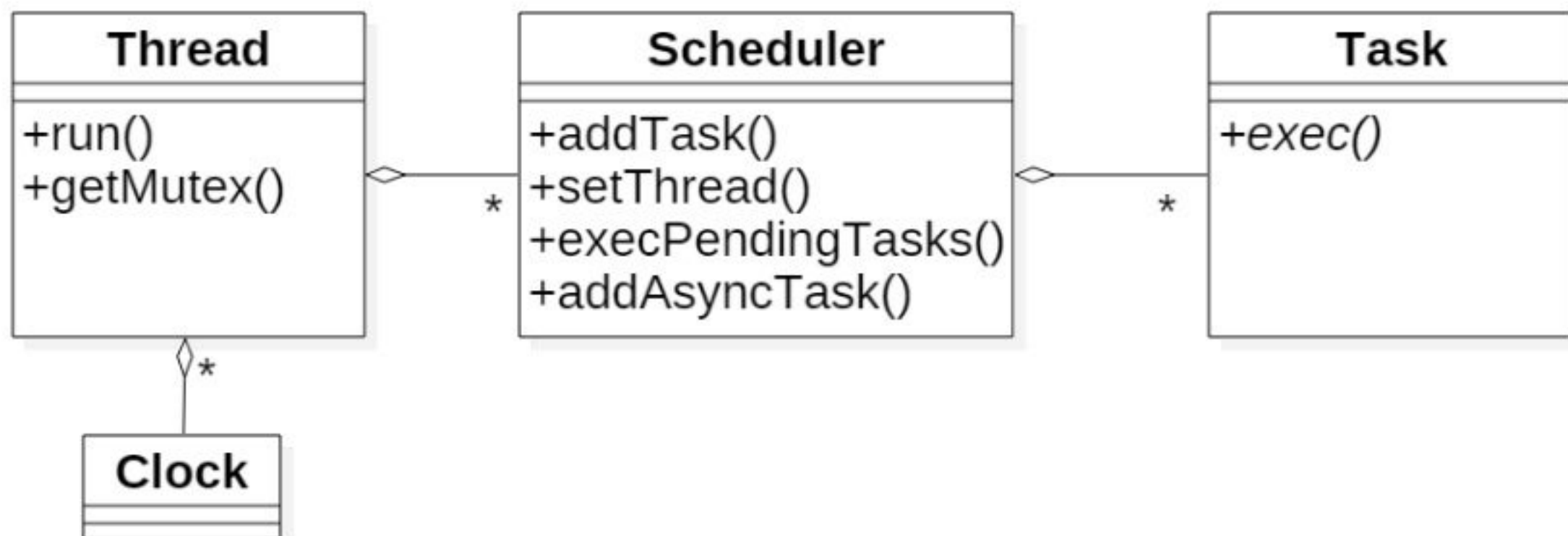
Определяем сущности

- Модели (Машины состояний и алгоритмы)
- Порты (Коммуникационные каналы)
- Планировщик задач
- Поток

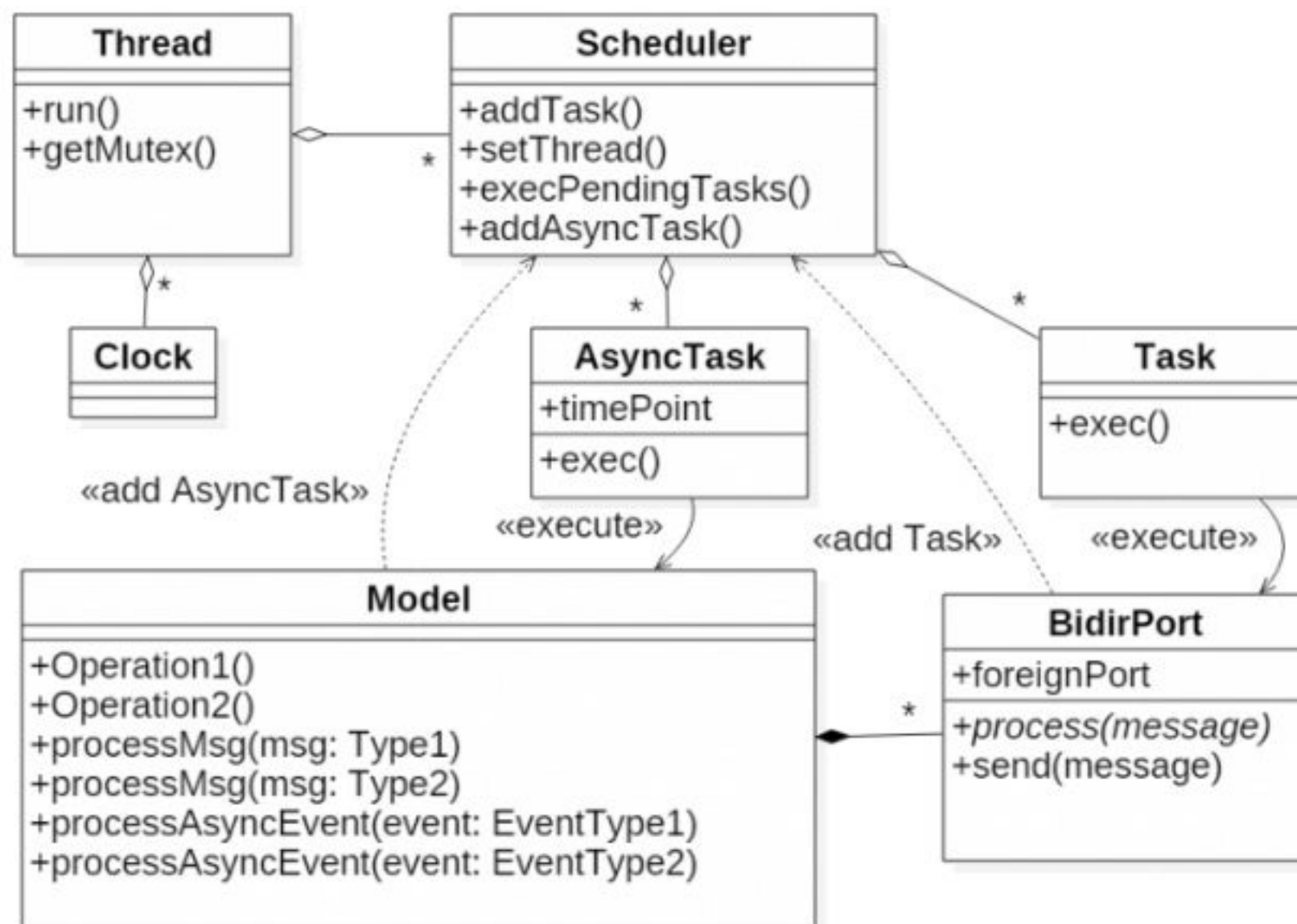
Взаимодействие моделей



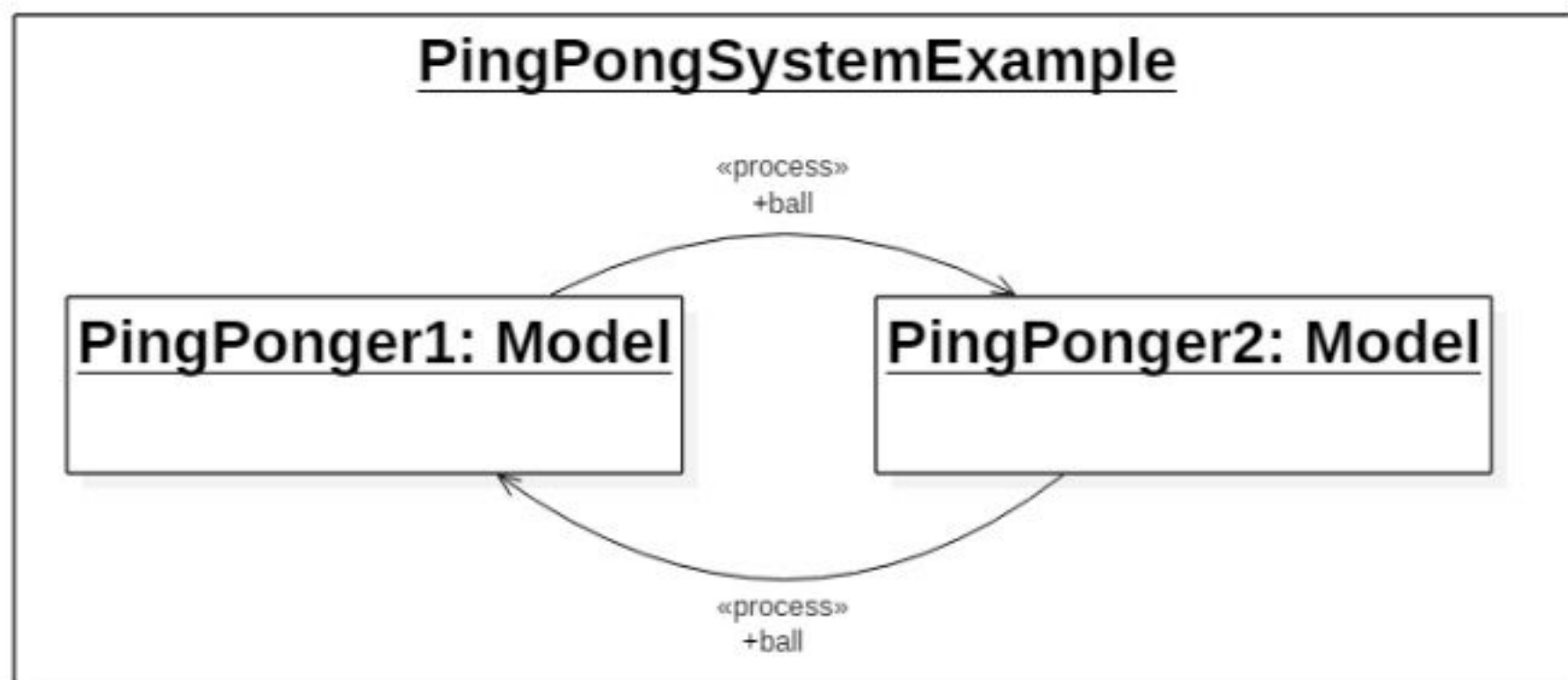
Планирование задач



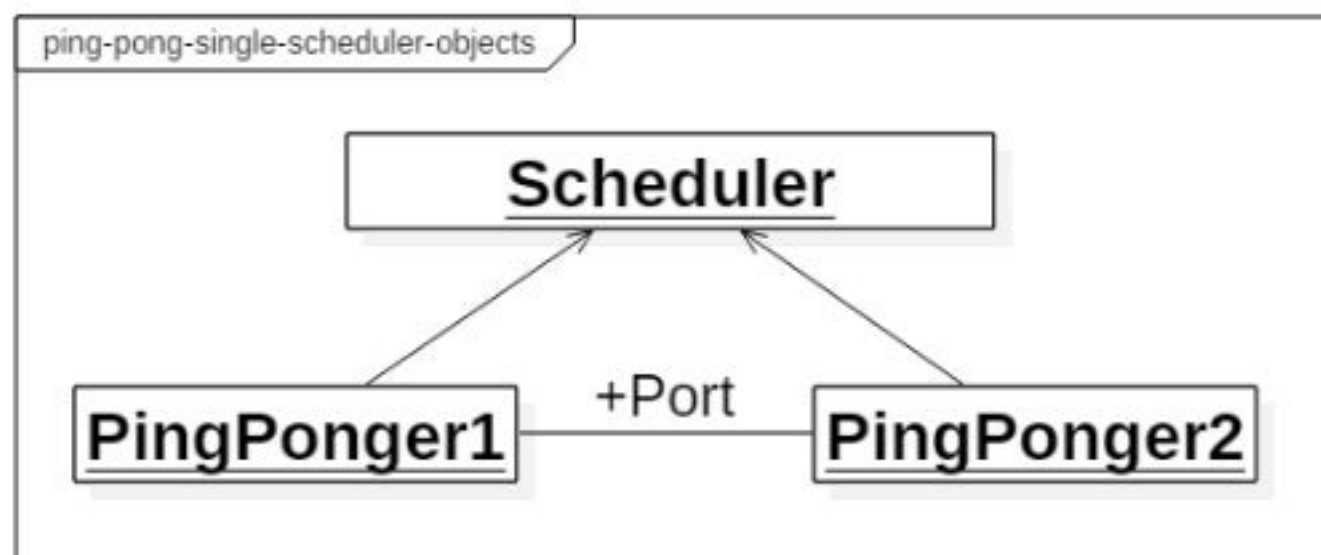
Планирование задач и взаимодействие



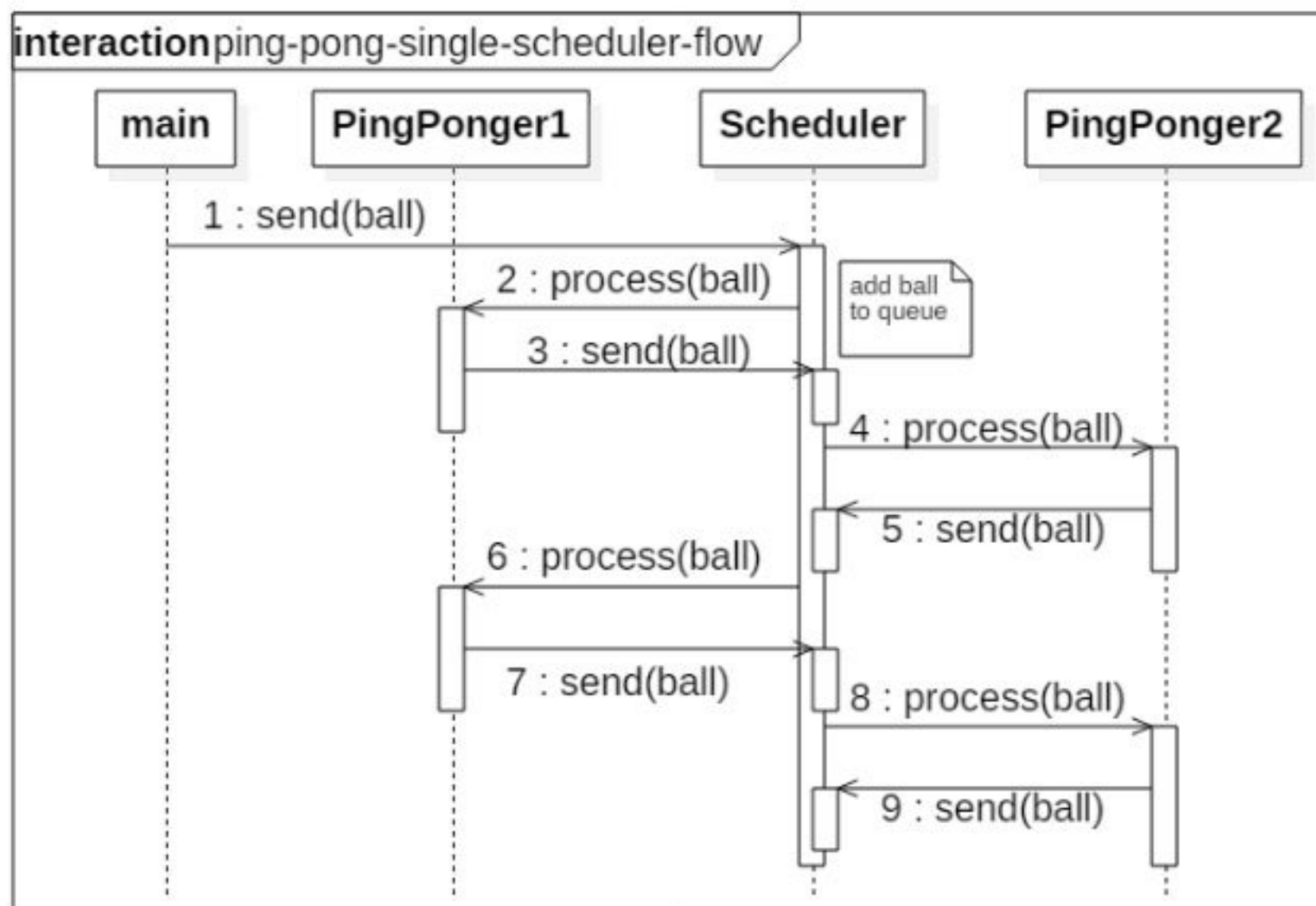
ПИНГ-ПОНГ



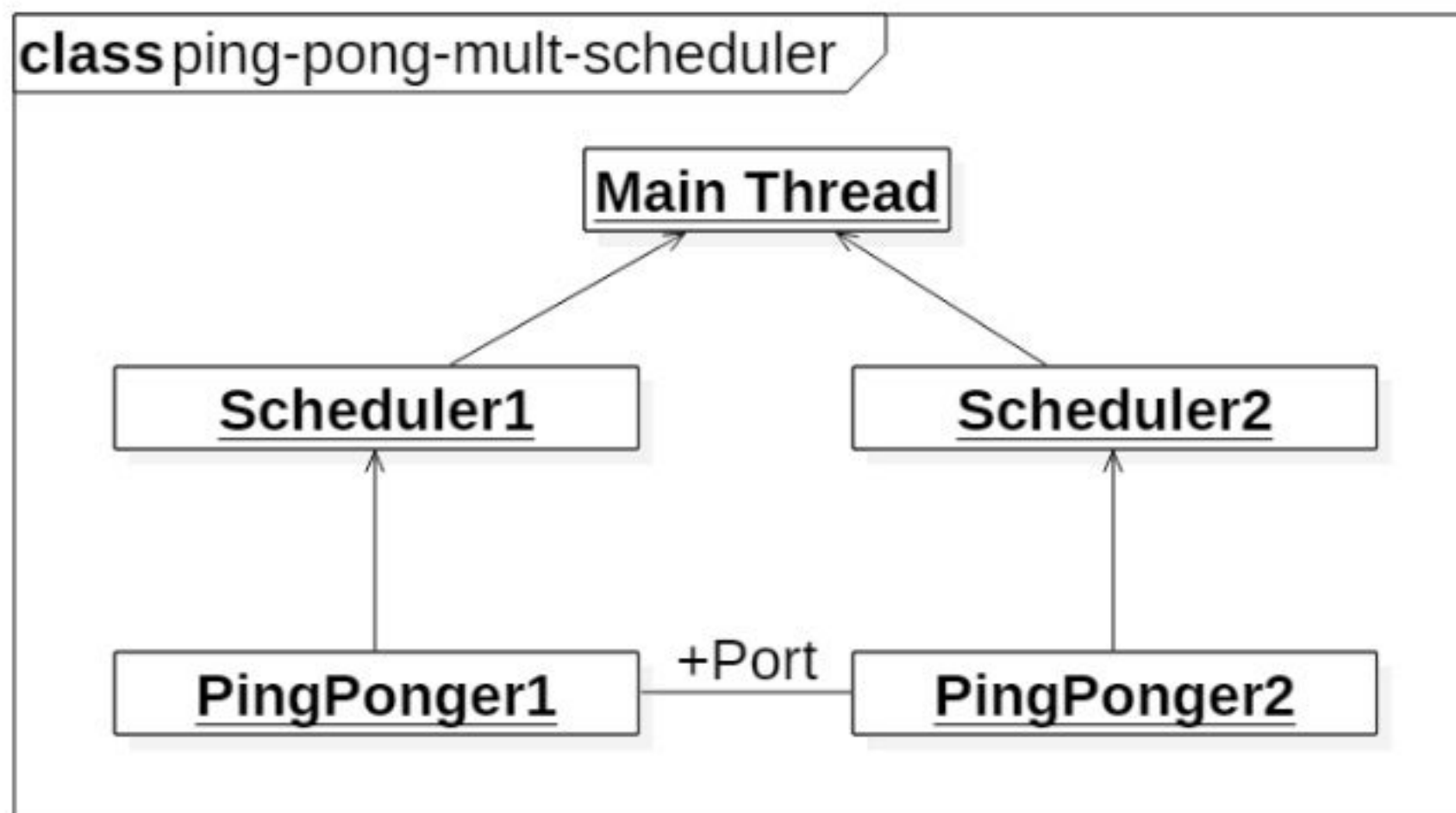
Модель игры с одним планировщиком



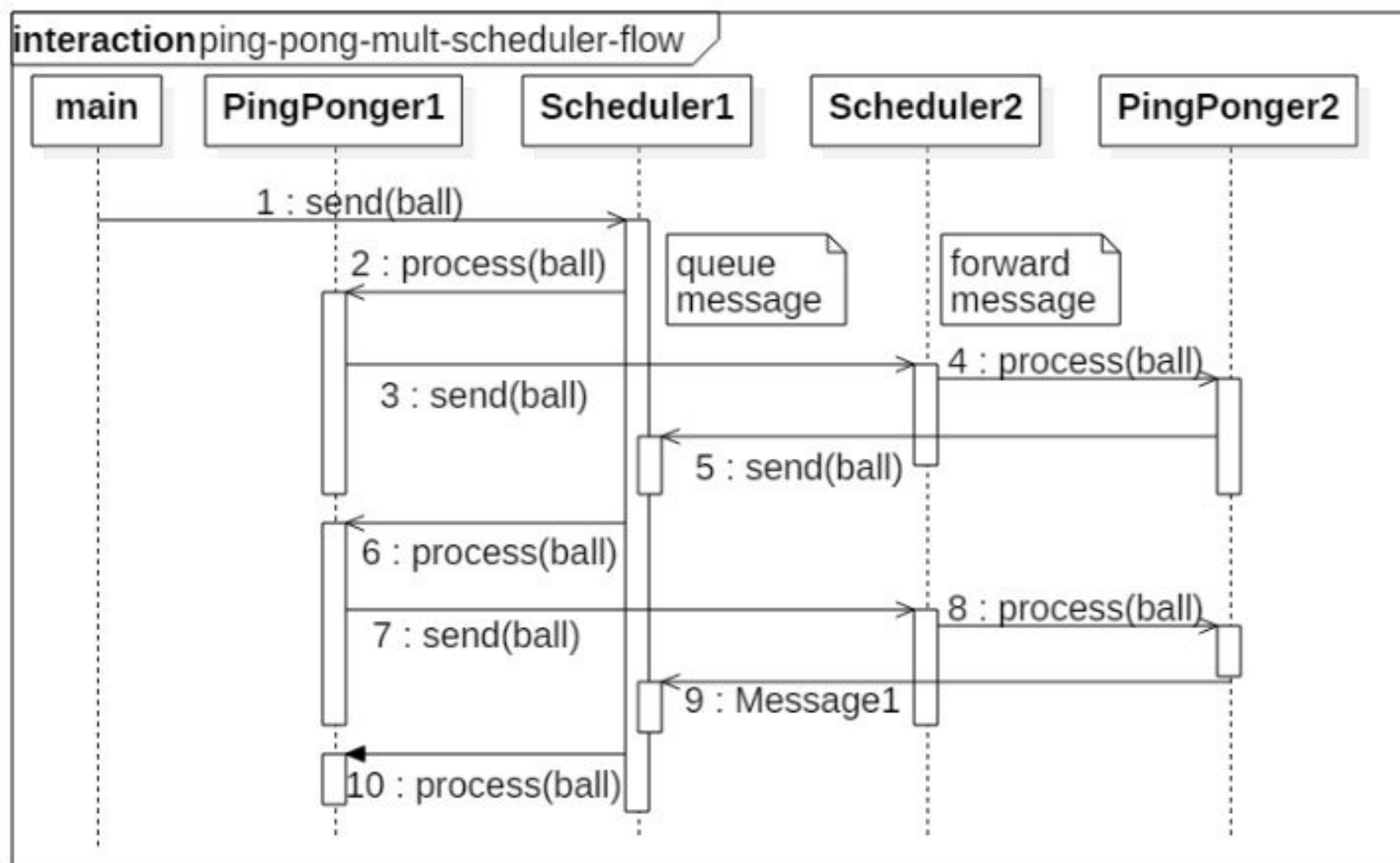
ПИНГ-ПОНГ С ОДНИМ ПЛАНИРОВЩИКОМ



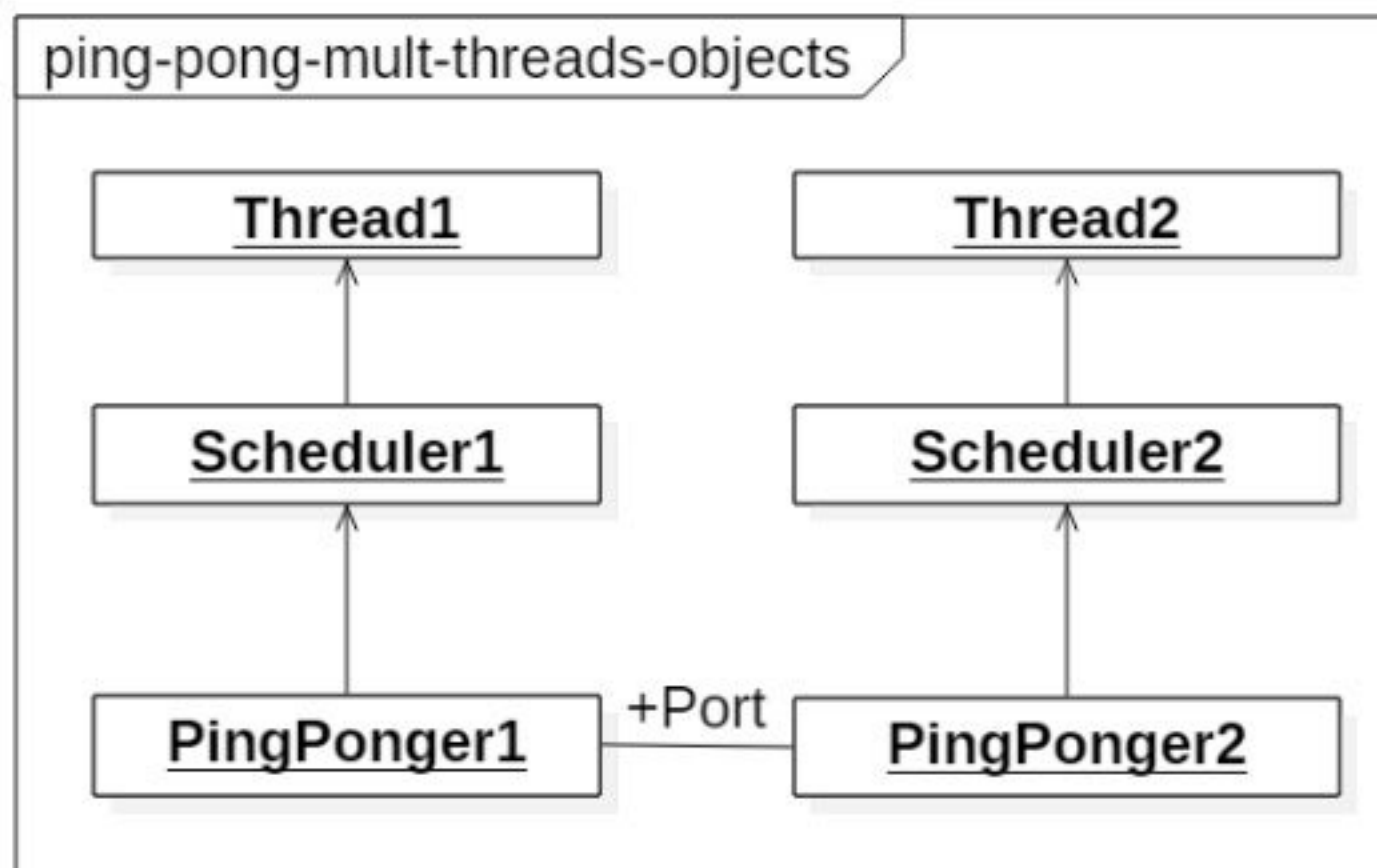
Модель игры с двумя планировщиками



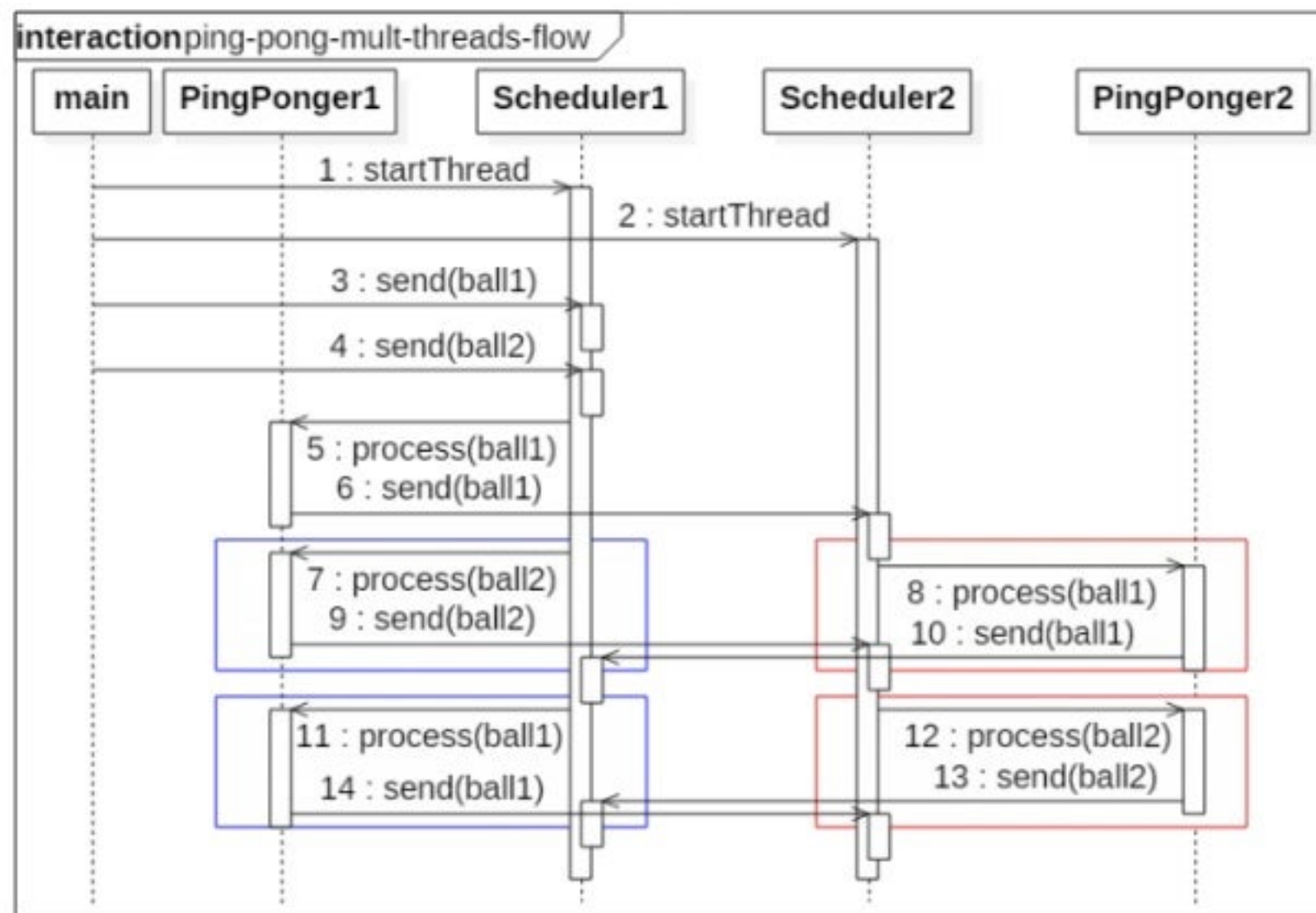
Пинг-понг с двумя планировщиками



Модель игры с двумя планировщиками и потоками



Пинг-понг с двумя потоками



Исходный код - игрок (PingPonger)

```
struct PingPonger : sms::BidirPort<Ball, Ball>
{
    void process(const Ball& msg) override
    {
        challengeTask();
        if (_count++<_maxCount)
            send(msg);
        else
            _finished = true;
    }
    uint32_t _maxCount = 1000000;
    uint32_t _count = 0;
    bool _finished = false;
};
```

Исходный код - создание объектов

//Общий планировщик

```
sms::QueuedScheduler scheduler;  
PingPonger pingPonger1(scheduler);  
PingPonger pingPonger2(scheduler);
```

//Каждому по планировщику

```
sms::QueuedScheduler scheduler1;  
sms::QueuedScheduler scheduler2;  
PingPonger pingPonger1(scheduler1);  
PingPonger pingPonger2(scheduler2);
```

//Каждому по потоку

```
sms::Thread thread1;  
sms::Thread thread2;  
sms::QueuedScheduler scheduler1(thread1);  
sms::QueuedScheduler scheduler2(thread2);  
PingPonger pingPonger1(scheduler1);  
PingPonger pingPonger2(scheduler2);
```

Исходный код - заброс шарика

```
//Общий планировщик
```

```
pingPonger1.process(Ball());
```

```
//Каждому по планировщику
```

```
pingPonger1.process(Ball());
```

```
//Каждому по потоку
```

```
thread1.start();
```

```
thread2.start();
```

```
pingPonger1.process(Ball());
```

```
//pingPonger1.process(Ball());
```

```
while (!pingPonger1._finished)
```

```
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
```

```
thread1.terminate();
```

```
thread2.terminate();
```

Результаты тестов - Linux

Linux kernel 4.4, Intel Core i7 2.4GHz, GCC

	Один планировщик	Два планировщика	Два потока
Один мяч, мкс	0.53	0.54	2,19+
Два мяча, мкс	0.52	0.56	0.85
Один мяч и вычисления, мкс	4.4	4,4	6,8+
Два мяча и вычисления, мкс	4.4	4.4	3.3

Результаты тестов - Windows

Intel Core 2 Duo CPU 3GHz, Windows 8.1 x64, Visual C++

	Один планировщик	Два планировщика	Два потока
Один мяч, мкс	1.1	1.2	20.0...40.0
Два мяча, мкс	1.3	1.2	4.0...20.0
Один мяч и вычисления, мкс	5.8	5.9	15.0...25.0
Два мяча и вычисления, мкс	5.7	5.9	5.9...10

Результаты тестов - Linux (ARM)

Linux Kernel 2.6, ARM926, GCC

	Один планировщик	Два планировщика	Два потока
Один мяч	26 мкс	28 мкс	18 мс
Два мяча	27 мкс	28 мкс	11 мс
Один мяч и вычисления, мкс	353 мкс	353 мкс	28 мс
Два мяча и вычисления, мкс	354 мкс	355 мкс	12,5 мс

Результаты тестов

- При использовании 2 потоков обработки сообщений результат не стабилен
- Потоки становятся стабильными, если время вычисления превышает 10...100 мкс.
- Windows и компилятор Visual C++ хуже справляются с задачей, чем Linux и GCC

Подведем итог

- Использование потоков ускоряет разработку небольших приложений на начальном этапе
- Активное использование потоков затрудняет отладку и тестирование
- Потоки позволяют ускорить работу “длинных” задач за счет распараллеливания ($>0,1..1$ мс)
- Для коротких задач цикл выполнения задач лучше, чем `std::thread`
- Существующие решения
 - Модель акторов - SObjectizer, C++ Actor Framework (CAF)
 - Qt - QEventLoop, QTimer, QSocketNotifier

Что не вошло в демо?

- Асинхронные задачи AsyncTask и планировщик AsyncScheduler
- Синхронный порт SyncPort и удаленные вызовы RPC
- Сериализатор JSON
- Мост и маршрутизатор
- Модульное тестирование

Что еще предстоит?

- Сериализация (рефлексия) (JSON, ProtoBuf и т. п.)
- Передача данных (TCP/UDP, Shared Memory)

Вопросы?

Тема: Модульность и управляемая многопоточность.
Трудности, проблемы, решения. Пример реализации на C++.

Исходный код: <https://github.com/vasviazh/sms>

Докладчик: Василий Вяжевич

Компания: Klika Tech

email/skype: vviazhevich@klika-tech.com

email/skype: vasviazh@gmail.com / vas_viazh