# Современные средства разработки ПО

## Пул потоков и другие патерны параллельного проектирования.

**Фетисов Михаил Вячеславович**
**fetisov.michael@bmstu.ru**

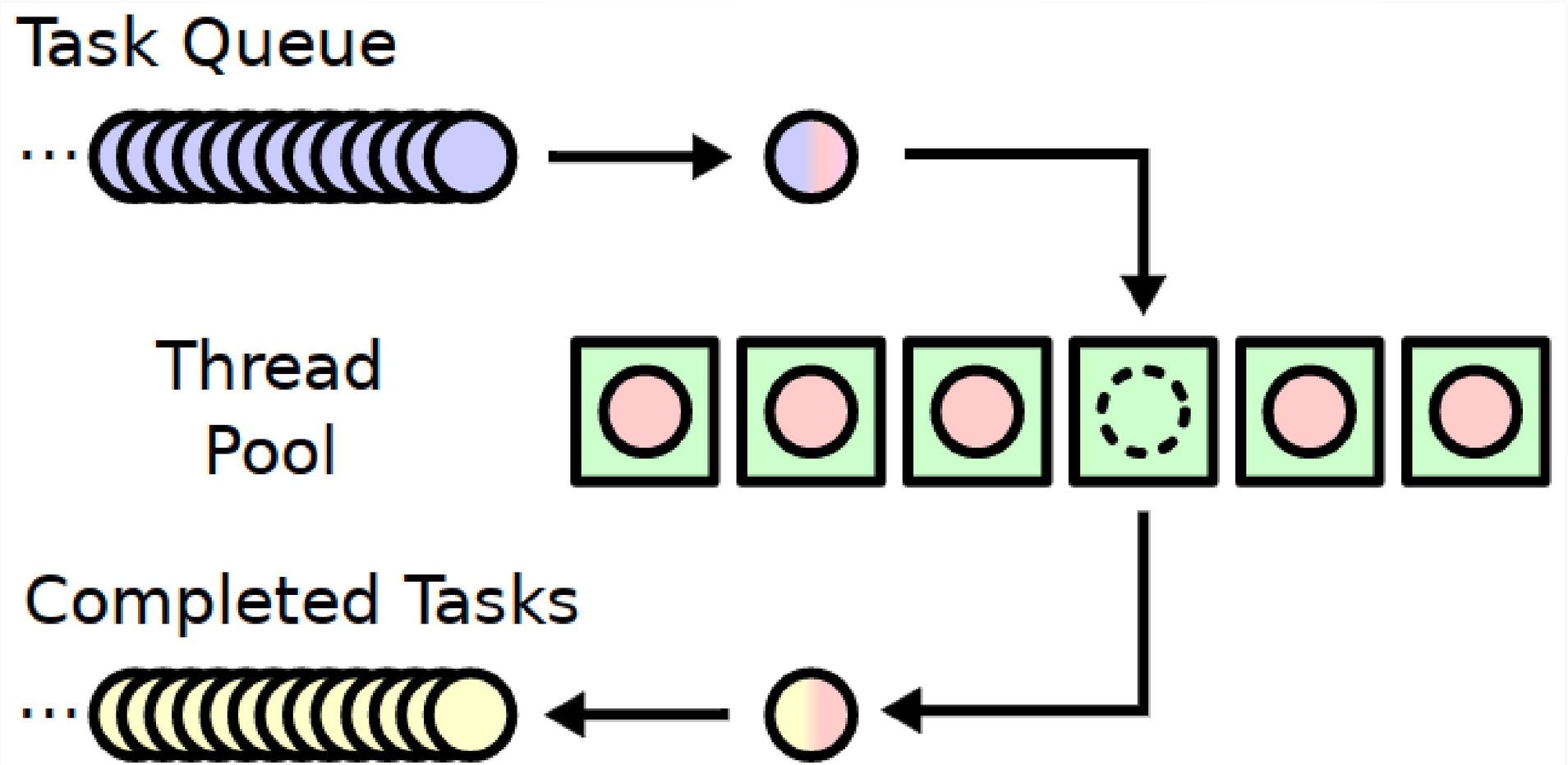
# Пул потоков (Thread pool)
# Задача

- Проблемы с потоками:
  - Создание и уничтожение потока и связанных с ним ресурсов может быть дорогостоящим процессом с точки зрения времени.
  - Одновременная работа большого количества потоков может снижать производительность программы.
- Задача:
  - Необходимо снизить накладные расходы на создание и уничтожение потоков для большого количества слабозависимых задач.

# Пул потоков (Thread pool)
## Определения

- **Пул потоков** (анг. thread pool) — шаблон параллельного программирования, который поддерживает несколько потоков, ожидающих выделения управляющей программой задач для одновременного выполнения.
- **Размер пула потоков** — это количество потоков, выделенных для выполнения задач.

# Пул потоков (Thread pool)
## Схема



# Пул потоков (Thread pool) Характеристики

- Выбор оптимального размера пула потоков имеет решающее значение для оптимизации производительности.
- Пул потоков часто совмещают с очередью задач.
- Могут быть реализованы сложные пулы потоков, которые поддерживают асинхронность, работу с сопрограммами, несколько пулов для прерываемых (ждущих) и не прерываемых задач (см. http://userver.tech) и др.

# Пул потоков (Thread pool)
# Преимущества

- Накладные расходы на создание и уничтожение потоков ограничены начальным созданием пула, что может привести к повышению производительности и стабильности системы.
- Обеспечивает управление параллелизмом и синхронизацией потоков на более высоком уровне абстракции, чем это можно сделать при ручном управлении потоками. В этом случае преимущества производительности от использования могут быть второстепенными.

# Пул потоков (Thread pool)
# Недостатки

- При активном использовании синхронизации между задачами в пуле потоков, он может стать менее эффективным.

# Пул потоков (Thread pool)
# Пример (объявление класса)

```
class ThreadPool
{
    int                         _number_of_threads;
    std::atomic_bool            _necessary_to_stop;
    std::condition_variable     _waiting_condition;
    std::mutex                  _waiting_mutex;
    std::vector<std::thread>    _threads;

    ThreadsafeQueue<Task_interface *> _task_queue;

    void worker();

public:
    ThreadPool(int number_of_threads=-1);

    ~ThreadPool();
    void submit(Task_interface * task);
    void start();
    size_t queue_length() const { return _task_queue.size(); }
};
```

# Пул потоков (Thread pool)
# Пример (конструктор и деструктор — RAII)

```
ThreadPool::ThreadPool(int number_of_threads)
    : _number_of_threads(number_of_threads)
    , _necessary_to_stop(false)
{
    if (_number_of_threads < 0)
        _number_of_threads = std::thread::hardware_concurrency();
}
```

```
ThreadPool::~ThreadPool()
{
    _necessary_to_stop = true;
    _waiting_condition.notify_all();

    for(size_t i=0; i < _threads.size(); ++i)
        _threads[i].join();
}
```

# Пул потоков (Thread pool)
# Пример (создание потоков и начало работы)

```
void ThreadPool::start()
try
{
    if (_number_of_threads == 0)
        return;

    _threads.reserve(_number_of_threads);

    for(int i=0; i < _number_of_threads; ++i)
        _threads.push_back(std::thread(&ThreadPool::worker,this));

    _waiting_condition.notify_all();
}
catch(...) {
    _necessary_to_stop = true;
    _waiting_condition.notify_all();
    throw;
}
```

# Пул потоков (Thread pool)
# Пример (помещение задачи на исполнение)

```
void ThreadPool::submit(Task_interface * task)
{
    if (_number_of_threads == 0) {
        task->work();
        delete task;
    }
    else {
        _task_queue.push(task);
        _waiting_condition.notify_one();
    }
}
```

# Пул потоков (Thread pool)
# Пример (метод-обработчик)

```
void ThreadPool::worker()
{
    while(!_necessary_to_stop || !_task_queue.empty()) {
        if (_task_queue.empty()) {
            std::unique_lock<std::mutex> locker(_waiting_mutex);
            _waiting_condition.wait(locker, [this]{ return !_task_queue.empty() || _necessary_to_stop; });
        }

        Task_interface * task = nullptr;

        if(_task_queue.try_pop(task)) {
            assert(task);
            task->work();
            delete task;
        }
    }
}
```

# Пул потоков (Thread pool)
# Пример (интерфейс задачи)

```
class Task_interface
{
public:
    virtual ~Task_interface() = default;

    /**
     * @brief Метод вызывается пулом потоков tp::ThreadPool,
     * когда доходит очередь до выполнения переданной в пул потоков реализации интерфейса
     * tp::Task_interface
     *
     */
    virtual void work() = 0;
};
```

# Пул потоков (Thread pool)
# Пример (реализация задачи из ЛР4)

```
class Application : public tp::Task_interface
{
    ItemCollector &                 _col;
    const std::vector<std::string> _args;
    const IOutput &                 _out;

public:
    Application() = delete;
    Application(const Application &) = delete;

    Application & operator=(const Application &) = delete;

    Application(ItemCollector & col, const std::vector<std::string> & args, const IOutput & out)
        : _col(col)
        , _args(args)
        , _out(out)
    {}

    virtual void work() override;
};
```

# Пул потоков (Thread pool)
# Пример (работа с пулом потоков)

```
tp::ThreadPool tp(number_of_threads);

tp.start();
```

```
    tp.submit(new Application(col,args,out));
```

# Пул потоков (Thread pool)
# Пример в проекте

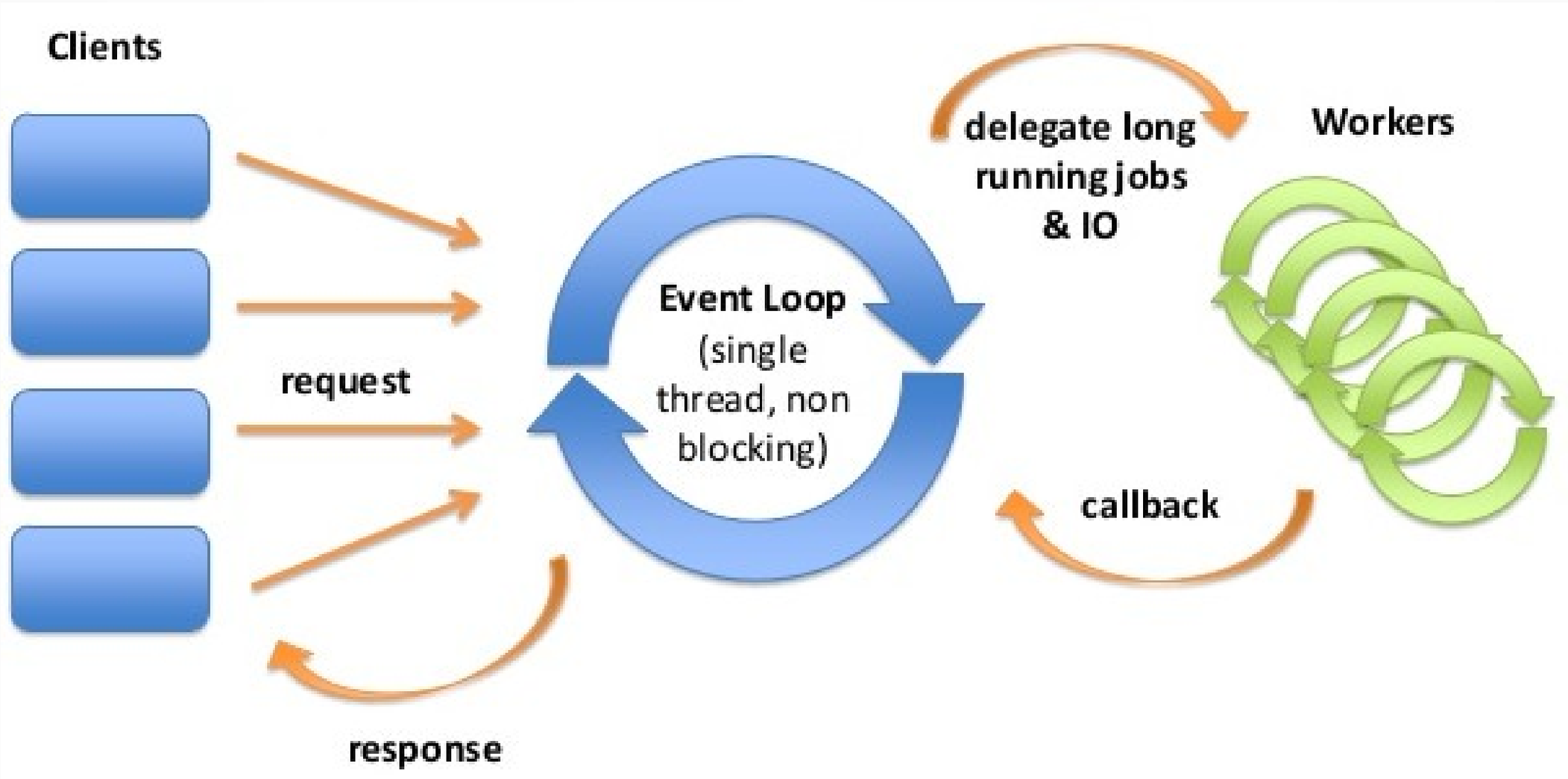
# Реактор (Reactor)
# Задача

- Проблемы с потоками:
  - Часто алгоритм можно выполнить в отдельном потоке, но результат его работы передать в синхронный (не многопоточный алгоритм).
  - Такое разделение особенно было бы полезно, если результат обрабатывается независимым алгоритмом.
  - Использование шаблона promice-future не годится, т. к. только первая часть алгоритма знает, какой обработчик необходим.
- Задача:
  - Нужно сделать так, чтобы формирование данных выполнялось параллельно, и использование — в одном потоке (без контроля гонки). Обработчик задаётся формируемым алгоритмом.

# Реактор (Reactor)
# Определения

- **Реактор** (англ. Reactor) — предназначен для синхронной передачи запросов сервису от одного или нескольких источников.
- Шаблон проектирования реактора представляет собой шаблон обработки событий для обработки запросов на обслуживание, передаваемых одновременно обработчику услуг одним или несколькими входами. Обработчик сервиса затем демультиплексирует входящие запросы и отправляет их синхронно связанным обработчикам запросов.

# Реактор (Reactor)
# Схема



# Реактор (Reactor)
# Структура

- Ресурсы:
  - Любой ресурс, который может обеспечить ввод или потребление выходных данных из системы.
- Синхронный демультиплексор событий:
  - Использует цикл событий для блокировки всех ресурсов. Демультиплексор отправляет ресурс диспетчеру, когда можно запустить синхронную операцию на ресурсе без блокировки (пример: синхронный вызов read () будет блокироваться, если нет данных для чтения. Демультиплексор использует select () на ресурс, который блокируется до тех пор, пока ресурс не будет доступен для чтения. В этом случае синхронный вызов read () не будет блокироваться, а демультиплексор может отправить ресурс диспетчеру.)
- Диспетчер:
  - Обрабатывает регистрацию и отмена регистрации обработчиков запросов. Отправляет ресурсы из демультиплексора в соответствующий обработчик запросов.
- Обработчик запросов:
  - Обработчик обработанного приложения и связанный с ним ресурс.

# Реактор (Reactor)
# Пример в проекте

# Шаблоны параллельного программирования*

- **Пул потоков** (Thread pool)
  - поддерживает несколько потоков, ожидающих выделения задач для одновременного выполнения управляющей программой.
- **Реактор** (англ. Reactor)
  - предназначен для синхронной передачи запросов сервису от одного или нескольких источников.
- **Очередь задач** (Task queue)
  - позволяет развязать основной алгоритм и другие шаблоны параллельного программирования (часто применяется с пулом потоков).
- **Блокиратор** (Locker)
  - безопасное использование блокировки (как правило на основе мьютекса) с применением идиомы RAII в C++.
- **Активный объект** (Active object)
  - шаблон проектирования, который отделяет поток выполнения метода от потока, в котором он был вызван (см. модель акторов).
- **Блокировка с двойной проверкой** (Double checked locking)
  - предназначен для уменьшения накладных расходов, связанных с получением блокировки.
- **Монитор** (Monitor)
  - объект, предназначенный для безопасного использования более чем одним потоком.
- **Планировщик** (Scheduler)
  - обеспечивающий механизм реализации политики планирования, но при этом не зависящий ни от одной конкретной политики.

# Вопросы?

Фетисов Михаил Вячеславович
fetisov.michael@bmstu.ru