Технологии программирования

Лекция 5 | Многопоточность

Лектор: Киселев Станислав Дмитриевич

Работа приложения

Основные понятия процесса, ядра, потока

Мы здесь

Однопоточное -> многопоточное ПО

И синхронное –> асинхронное

Основные проблемы многопоточности

Гонки данных, условия гонки и deadlock’и

Способы синхронизации данных

Как решение проблем многопоточности

Управление памятью и ресурсами в C++

Основные советы и примеры

Сравнение принципов языков C++, Rust и Go

В контексте управления ресурсами

Заключение

Подведение итогов о реализации управления ресурсами

Базовая схема работы приложения

ПРИЛОЖЕНИЕ

Базовая схема работы приложения

Процесс – это программа в стадии выполнения

Взаимодействие с операционной системой

Когда вы пишете программу на C++ и запускаете её, **операционная система создает процесс**, который включает в себя:

- код программы
- данные (переменные и другие структуры данных)
- контекст выполнения (регистры процессора, состояние процессора)
- стек для управления функциями и их вызовами

Операционная система управляет процессами через механизм планировщика процессов:

1. Создает и уничтожает процессы
2. Распределяет процессорное время между процессами
3. Переключает выполнение процессов (контекст переключения)
4. Синхронизирует процессы и передает данные между ними

Основные характеристики процесса

Изолированность

У каждого процесса своя память, которую другие процессы напрямую не могут изменять

Собственное виртуальное адресное пространство,

которое операционная система отображает на физическую память

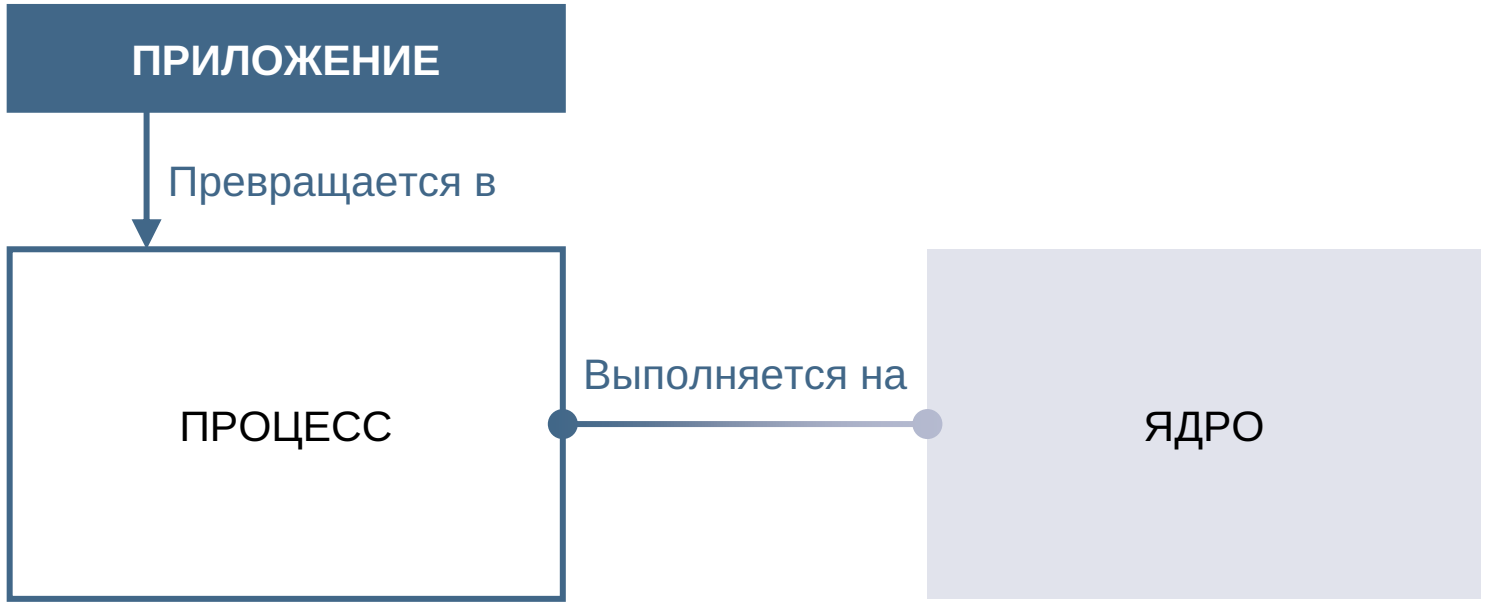
Уникальный идентификатор процесса (PID)

Используется ОС для отслеживания состояния процесса, распределения ресурсов и коммуникации

Ресурсы процесса, выделенные ОС

Такие как процессорное время, память и файловые дескрипторы

Базовая схема работы приложения



Ядро – основная рабочая единица центрального процессора (CPU), которая выполняет инструкции программ

Основные характеристики ядра процессора

Исполнение инструкций

Ядро обрабатывает инструкции программ, выполняя команды, записанные в машинном коде

Такие как арифметические операции, управление потоками и обращение к памяти

Многозадачность и многоядерные процессоры

Каждое ядро может одновременно выполнять свой поток или процесс. Это создаёт физическую возможность для параллелизма – когда несколько операций выполняются одновременно

Гиперпоточность (Hyper-Threading)

Некоторые процессоры поддерживают технологию гиперпоточности, которая позволяет одному физическому ядру одновременно обрабатывать два или более потока

Базовая схема работы приложения

Поток – единица выполнения внутри процесса

Один процесс может содержать несколько потоков, которые разделяют общее адресное пространство, но выполняются параллельно или последовательно

Различие между «процессом» и «потоком»

Поток – подмножество процесса, и его можно рассматривать как «легковесный» процесс

Потоки могут разделять ресурсы с другими потоками:

Общие ресурсы для потоков внутри одного процесса	Индивидуальные ресурсы для каждого потока
1. Код программы	1. Стек для локальных переменных и вызовов функций
2. Глобальные данные и статические переменные	2. Регистры процессора, такие как указатель стека и счетчик команд
3. Кучу (heap) – динамически выделенную память	

Потоки позволяют процессу выполнять несколько задач одновременно и требуют меньше ресурсов для создания и управления по сравнению с процессами

Взаимодействие потоков с ОС

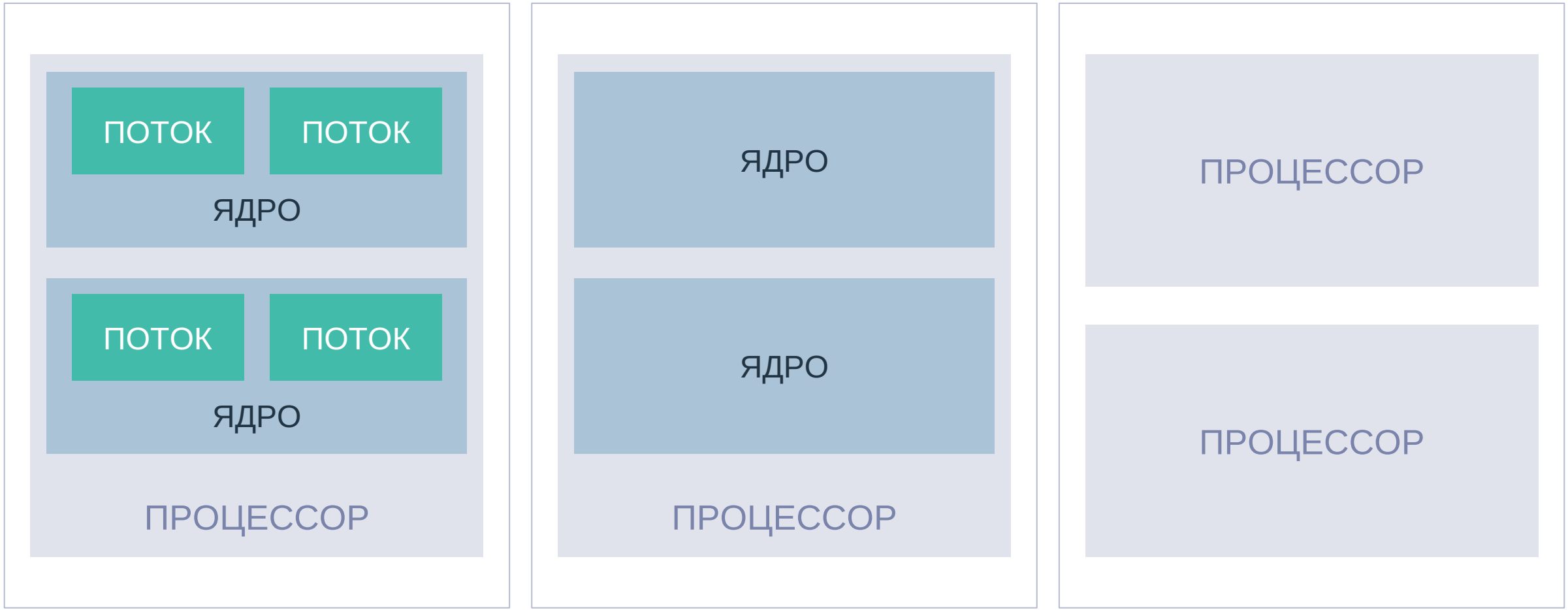
Потоки управляются как на уровне пользовательского пространства (с помощью библиотек), так и на уровне ядра операционной системы (с помощью механизмов ОС)

На уровне ядра ОС отвечает за переключение между потоками (контекстное переключение), управление временем выполнения и синхронизацию

В пользовательском пространстве программы могут управлять потоками с помощью **библиотек потоков**, например, `std::thread` в C++

В некоторых случаях роль «потока» на себя берет ядро или процессор

Взаимодействие потоков с ОС

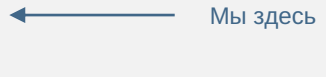


Работа приложения

Основные понятия процесса, ядра, потока

Однопоточное -> многопоточное ПО

И синхронное –> асинхронное



Основные проблемы многопоточности

Гонки данных, условия гонки и deadlock’и

Способы синхронизации данных

Как решение проблем многопоточности

Управление памятью и ресурсами в C++

Основные советы и примеры

Сравнение принципов языков C++, Rust и Go

В контексте управления ресурсами

Заключение

Подведение итогов о реализации управления ресурсами

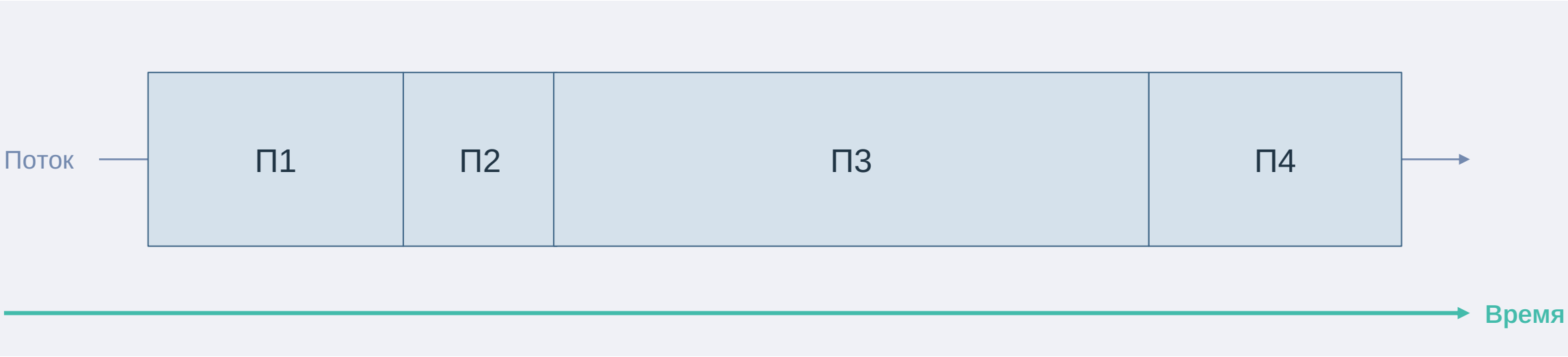
Базовая схема работы приложения

При такой схеме работы приложение будет называться **однопоточным**

Однопоточное синхронное программирование

Система в одном потоке работает со всеми задачами, выполняя их поочерёдно

- Потоку назначается одна задача, и начинается её выполнение
- Заняться следующей задачей можно только тогда, когда завершится выполнение первой
- Эта модель не предполагает приостановку одной задачи, чтобы выполнить другую

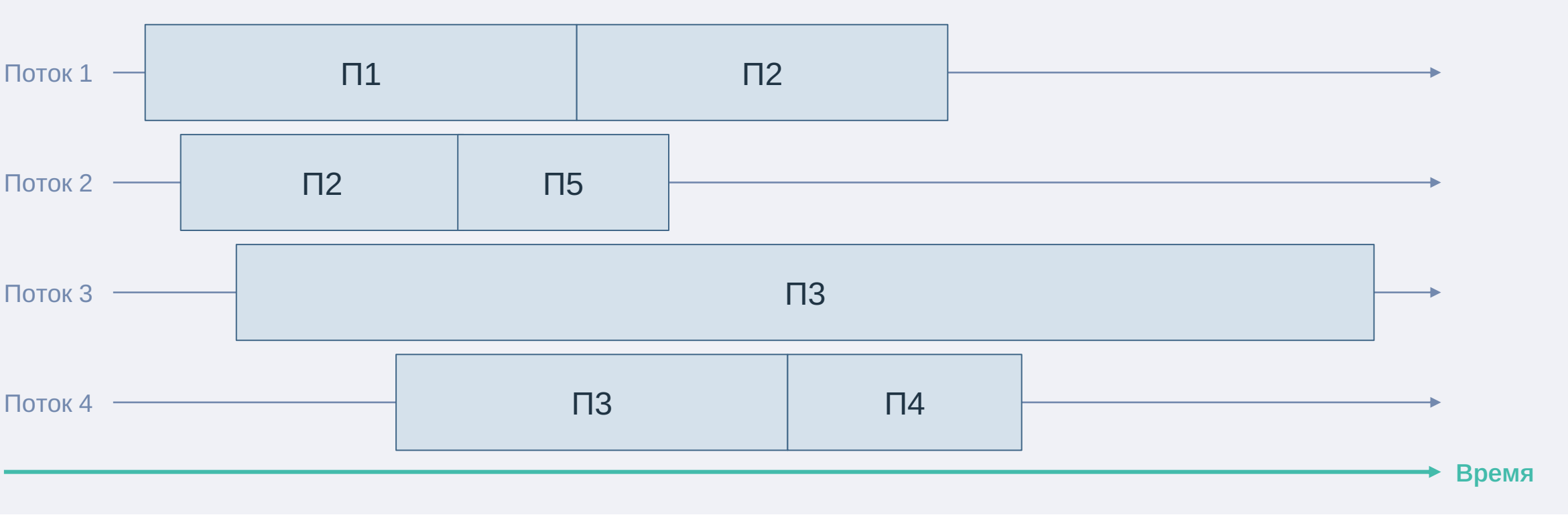


Добавляем потоки

Есть несколько потоков, которые мы можем использовать
При такой схеме можно говорить про **многопоточное приложение**

Многопоточное синхронное программирование

Выполнение задач идет одновременно на нескольких потоках и независимо друг от друга

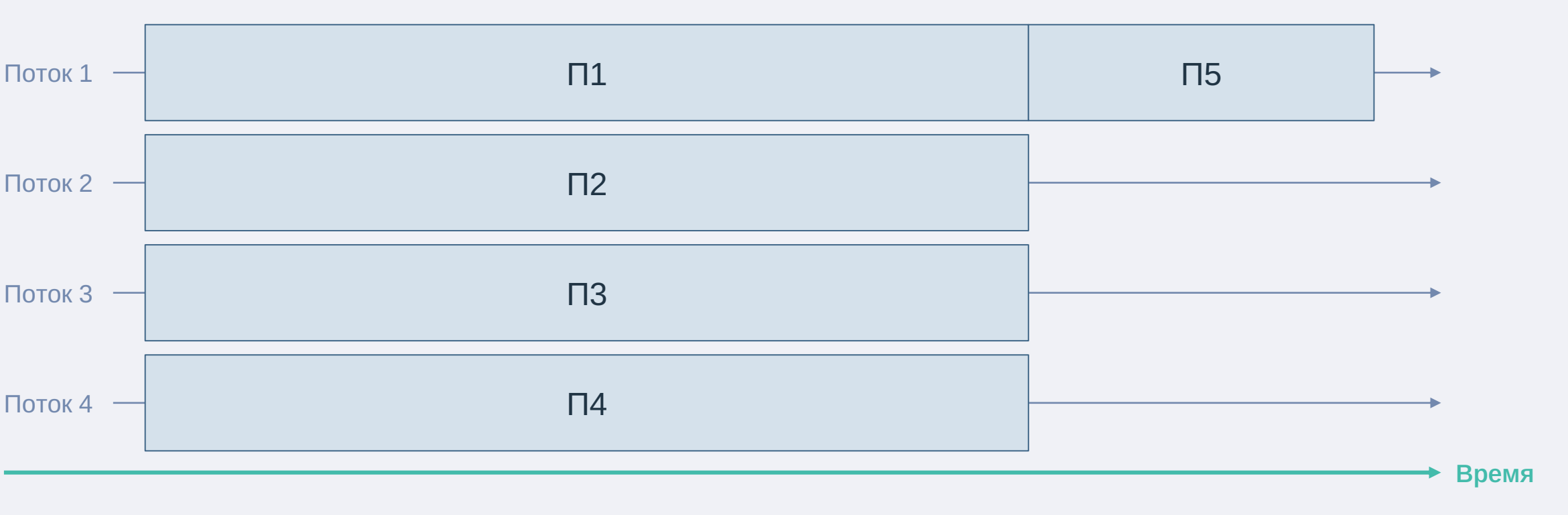


Применение многопоточности

Однопоточное ПО	Многопоточное ПО
Области применения	
• Простые задачи • Приложения без длительных операций ввода-вывода или вычислений	• Серверные приложения, обрабатывающие множество запросов • Приложения с разделением задач (например, интерфейс и обработка данных)
Ограничения и особенности	
• Неэффективно использует ресурсы многоядерных систем • Может быть неотзывчивым при выполнении длительных задач	• Каждый поток может выполнять свою задачу независимо • Необходимость синхронизации при доступе к общим ресурсам
Консольная утилита для парсинга данных	Веб-сервер – каждый входящий запрос обрабатывается в отдельном потоке

Параллелизм

Обработка независимых задач на нескольких потоках с последующим объединением результатов



Параллелизм в реальной жизни

```
void processChunk(int start, int end, std::vector<int>& data) {
    for (int i = start; i < end; ++i) data[i] = compute(data[i]);
}

 int main() {
    std::vector<int> data(1000000); // Инициализация данных

    int numThreads = std::thread::hardware_concurrency();
    std::vector<std::thread> threads;
    int chunkSize = data.size() / numThreads;

    for (int i = 0; i < numThreads; ++i) {
        int start = i * chunkSize;
        int end = (i == numThreads - 1) ? data.size() : start + chunkSize;
        threads.emplace_back(processChunk, start, end, std::ref(data));
    }

    for (auto& t : threads)
        t.join();
    return 0;
 }
```

Обработка больших данных
Например, параллельное умножение матриц в научных вычислениях

Графические вычисления
Использование GPU для параллельной обработки пикселей или векторов

NVIDIA CUDA
Технология для программ, использующих возможности GPU для параллелизма

Распараллеливание циклов
Разделение циклов на части, выполняемые в разных потоках

Асинхронное программирование

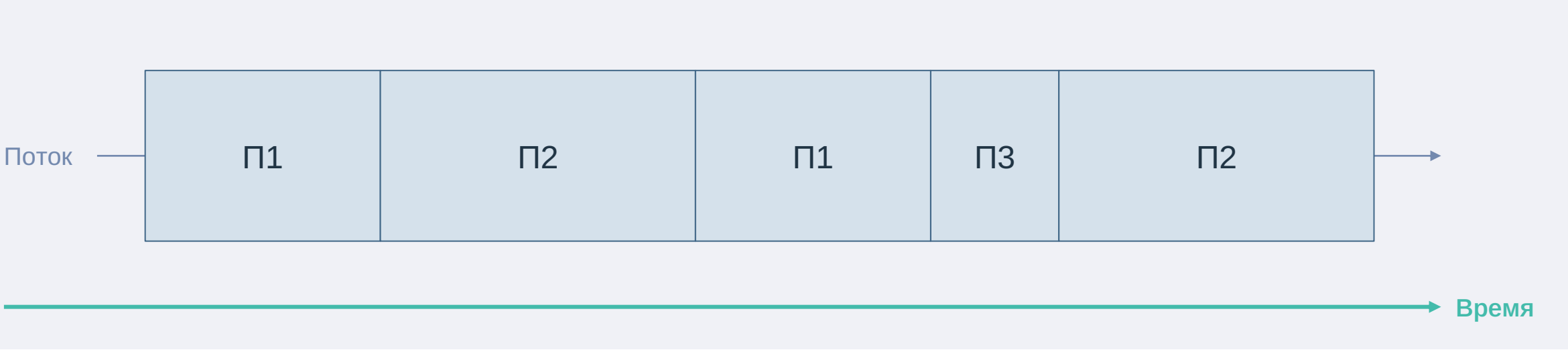
Многопоточность позволяет нам физически выполнять несколько задач одновременно на одном ядре процессора

Но, чтобы еще более эффективно использовать ресурсы одного потока, не блокируя его выполнение на длительных операциях, применяют **асинхронное программирование**

Асинхронное программирование

Задачи обрабатываются вместе, при этом поток переключается между задачами по необходимости

- Мы не ждем завершения одной операции перед началом другой
- Избегаем простоев потока с помощью логики ПО



JavaScript – пример однопоточного асинхронного кода

```
console.log('Start');

setTimeout(() => {
  console.log('Timeout');
}, 1000);

console.log('End');

// Вывод:
// Start
// End
// Timeout
```

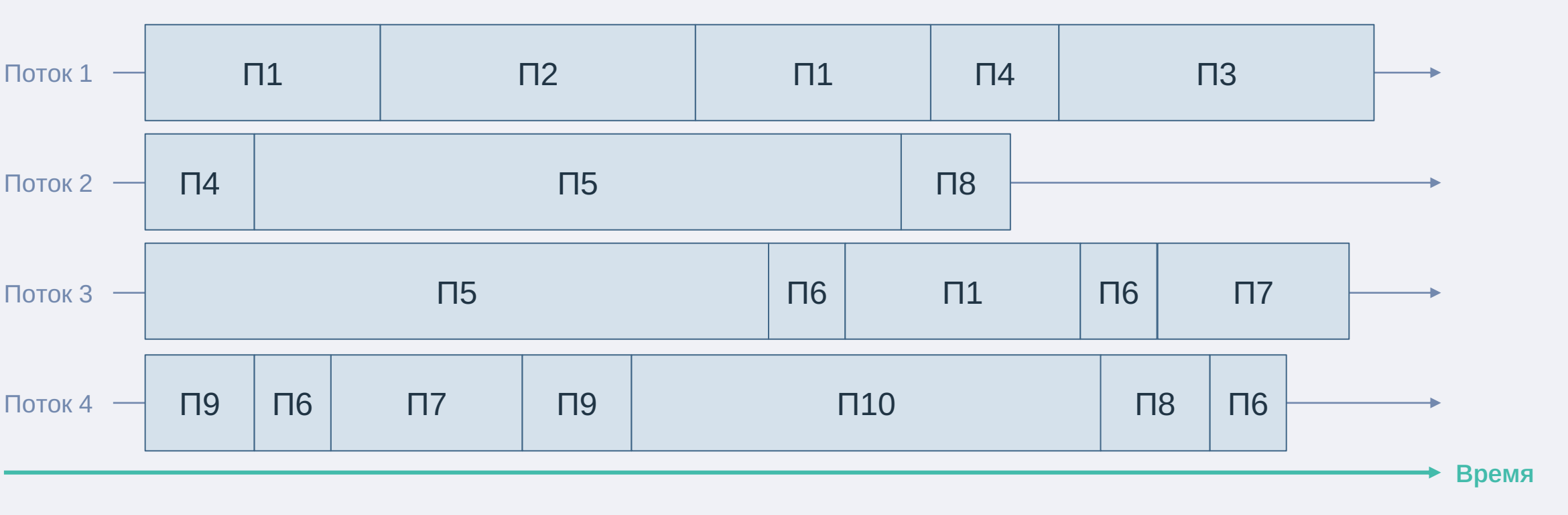
Событийный цикл
Управляет выполнением задач и обработкой событий

Доступен только 1 поток

Асинхронные операции
Выполняются через коллбеки, промисы или async/await, не блокируя основной поток

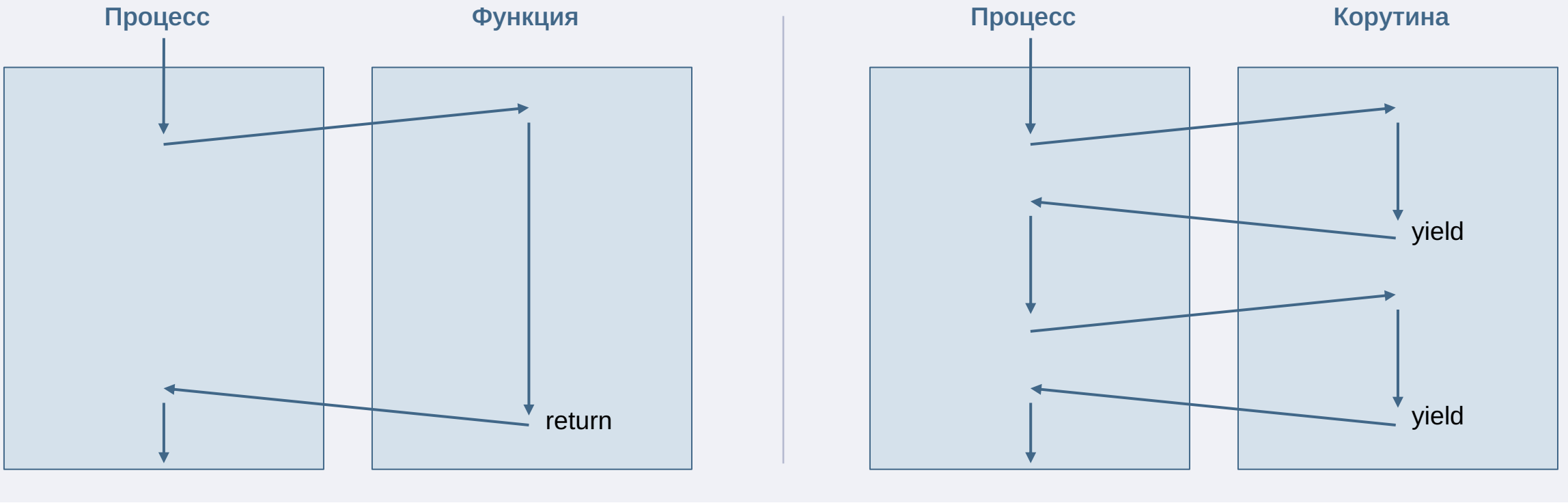
Многопоточное асинхронное приложение

Комбинирует все преимущества подходов, используется в высоконагруженных системах



Корутины

Это механизм, позволяющий функции приостанавливать своё выполнение и возобновлять его позже, сохраняя при этом своё состояние



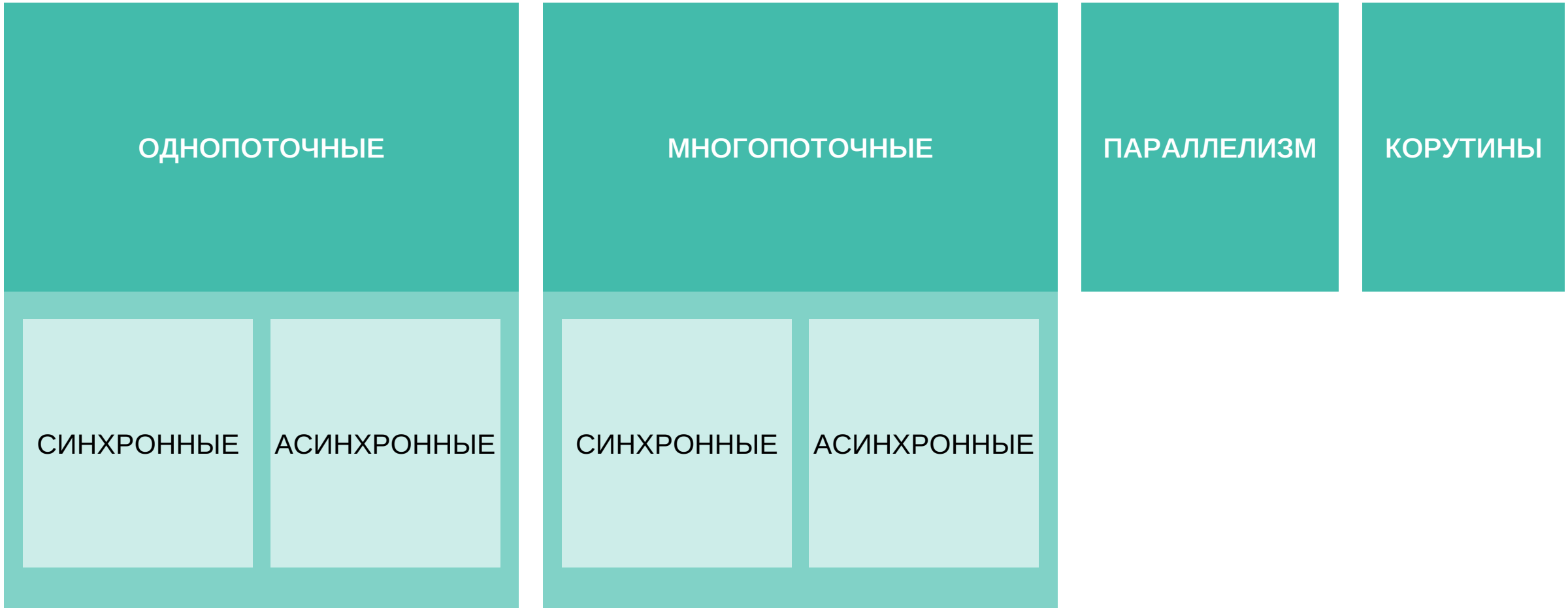
Корутины

```
struct task {
    struct promise_type {
        task get_return_object() {
            return {std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle_;
};

// Coroutine function
task async_wait(std::chrono::milliseconds delay) {
    std::cout << "Coroutine is about to be suspended..." << std::endl;
    co_await simple_awaiter{delay}; // Suspend the coroutine
    std::cout << "Coroutine has been resumed!" << std::endl;
    co_return; // End the coroutine
}
```

Виды работы приложения



Работа приложения

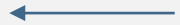
Основные понятия процесса, ядра, потока

Однопоточное -> многопоточное ПО

И синхронное –> асинхронное

Основные проблемы многопоточности

Гонки данных, условия гонки и deadlock’и



Способы синхронизации данных

Как решение проблем многопоточности

Управление памятью и ресурсами в C++

Основные советы и примеры

Сравнение принципов языков C++, Rust и Go

В контексте управления ресурсами

Заключение

Подведение итогов о реализации управления ресурсами

Гонки данных (Data Races)

```
int counter = 0;

void increment() {
    for (int i = 0; i < 1000; ++i) {
        ++counter;
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();
    std::cout << "Counter: " << counter << std::endl;
}
```

Это ситуация, когда два или более потока **одновременно обращаются** к одной и той же области памяти, и по крайней мере один из потоков **изменяет данные**

Результат работы программы становится непредсказуемым, поскольку порядок выполнения потоков может меняться

Решение – использование Mutex

О нем позднее ⟶

Тупики (Deadlocks)

```
std::mutex mtx1, mtx2;

void task1() {
    std::lock_guard<std::mutex> lock1(mtx1);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock2(mtx2);
}

void task2() {
    std::lock_guard<std::mutex> lock2(mtx2);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock1(mtx1);
}

int main() {
    std::thread t1(task1);
    std::thread t2(task2);
    t1.join();
    t2.join();
}
```

Это ситуация, когда два или более потока **блокируют друг друга**, ожидая освобождения ресурсов, которые удерживаются другими потоками

В результате программы «зависают», и выполнение не может продолжаться

Решение – использование try_lock:

```
if (std::try_lock(mtx1, mtx2) == -1) {
    // Оба мьютекса захвачены
} else {
    // Один из мьютексов не захвачен
}
```

Условия гонки (Race Conditions)

```
bool ready = false;
int data = 0;

void producer() {
    data = 42;      // Производим данные
    ready = true;   // Сигнализируем, что данные готовы
}

void consumer() {
    while (!ready) {       // Ждём, пока данные будут готовы
    }
    std::cout << "Data: " << data << std::endl;
}

int main() {
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join();
    t2.join();
}
```

Возникают, когда конечный результат выполнения программы зависит от того, **в каком порядке** выполняются потоки

Хотя на первый взгляд может казаться, что все потоки выполняются правильно, результат работы может оказаться неверным из-за того, что один поток **успевает выполнить операцию быстрее другого**

Решение – std::condition_variable

О нем позднее ⟶

Условия гонки (Race Conditions) | Решение

```
std::mutex mtx;
std::condition_variable cv;
bool ready = false;
int data = 0;

void producer() {
    std::unique_lock<std::mutex> lock(mtx);
    data = 42;
    ready = true;
    cv.notify_one();  // Оповещаем потребителя
}

void consumer() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, []{ return ready; });  // Ждём, пока producer не
оповестит
    std::cout << "Data: " << data << std::endl;
}
```

Для правильной синхронизации потоков можно использовать такие механизмы, как **условные переменные**, которые позволяют потокам «ожидать» сигнала от других потоков перед выполнением задач

В C++ это std::condition_variable

Работа приложения

Основные понятия процесса, ядра, потока

Однопоточное -> многопоточное ПО

И синхронное –> асинхронное

Основные проблемы многопоточности

Гонки данных, условия гонки и deadlock’и

Способы синхронизации данных

Как решение проблем многопоточности

Управление памятью и ресурсами в C++

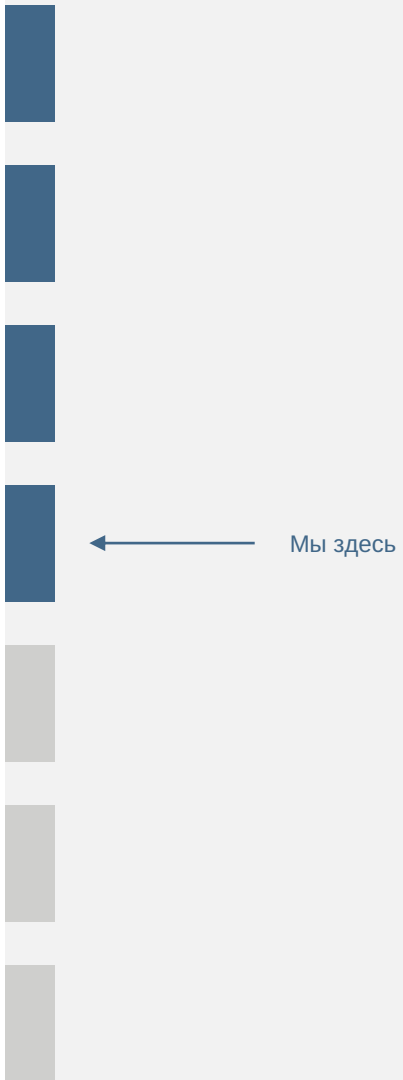
Основные советы и примеры

Сравнение принципов языков C++, Rust и Go

В контексте управления ресурсами

Заключение

Подведение итогов о реализации управления ресурсами



std::mutex

Простой мьютекс

Используется для защиты общего ресурса от одновременного доступа несколькими потоками

Работает по принципу захвата (lock) и освобождения (unlock)

Где применим:

- Когда общий ресурс (например, переменная или объект) должен изменяться только одним потоком одновременно
- Классический случай – защита разделяемых данных (баланс счетов, разделяемая память)

```
std::mutex mtx;  // Обычный мьютекс для защиты данных
double balance = 0.0;

void deposit(double amount) {
    std::lock_guard<std::mutex> lock(mtx);  // Захватываем мьютекс
    balance += amount;  // Обновляем баланс
    std::cout << "Deposited: " << amount << ", Current balance: " << balance
<< std::endl;
}

int main() {
    std::thread t1(deposit, 1000);
    std::thread t2(deposit, 500);

    t1.join();
    t2.join();

    std::cout << "Final balance: " << balance << std::endl;
    return 0;
}
```

std::recursive_mutex

Рекурсивный мьютекс – позволяет одному и тому же потоку захватывать его несколько раз без блокировки самого себя

Где применим:

- Полезно, когда функции могут вызывать сами себя рекурсивно, или если несколько функций в цепочке вызовов должны захватывать один и тот же мьютекс
- Примеры:
  работа с древовидными структурами данных, рекурсивные алгоритмы, вложенные вызовы методов в объектах, которые требуют синхронизации

```
std::recursive_mutex rmtx;  // Рекурсивный мьютекс
int shared_result = 1;

void factorial(int n) {
    if (n <= 1) return;

    std::lock_guard<std::recursive_mutex> lock(rmtx);  // Захватываем
рекурсивный мьютекс
    shared_result *= n;  // Обновляем общий результат
    factorial(n - 1);    // Рекурсивный вызов
}

int main() {
    std::thread t1(factorial, 5);

    t1.join();
    std::cout << "Final result: " << shared_result << std::endl;
    return 0;
}
```

std::timed_mutex

Мьютекс с таймаутом

Позволяет потокам пытаться захватить мьютекс в течение определённого времени

Где применим:

- Когда необходимо предотвратить тупики и разрешить потокам «отказываться» от попыток захватить ресурс после определённого времени.
- Примеры:
  работа с устройствами ввода-вывода, где время отклика важно, системы с высокой конкуренцией за ресурсы

```
std::timed_mutex tmtx;  // Мьютекс с таймаутом

void try_to_access(int id) {
    if (tmtx.try_lock_for(std::chrono::milliseconds(100))) {
        std::cout << "Thread " << id << " accessed the resource." << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(250));
        tmtx.unlock();  // Освобождаем мьютекс
    } else {
        std::cout << "Thread " << id << " could not access the resource in
time." << std::endl;
    }
}

int main() {
    std::thread t1(try_to_access, 1);    std::thread t2(try_to_access, 2);

    t1.join();    t2.join();
    return 0;
}
```

std::shared_mutex

Мьютекс с таймаутом

Несколько потоков одновременно получать доступ к ресурсу для чтения, но запись возможна только одним потоком

Где применим:

- Когда часто выполняются операции чтения, но запись происходит редко
- Примеры:
  кэширование данных, базы данных с высокой частотой чтения и низкой частотой записи

```
std::shared_mutex shmtx;  // Разделяемый мьютекс
int shared_data = 0;

void reader(int id) {
    std::shared_lock<std::shared_mutex> lock(shmtx);  // Чтение данных
    std::cout << "Reader " << id << " reads: " << shared_data << std::endl;
}
void writer(int value) {
    std::unique_lock<std::shared_mutex> lock(shmtx);  // Запись данных
    shared_data = value;
    std::cout << "Writer updated data to: " << shared_data << std::endl;
}

int main() {
    std::thread t1(reader, 1);
    // .. Threads t2, t3 создание
    t1.join();
    // t2, t3 join
    return 0;
}
```

Ключевое слово volatile

Используется для указания компилятору, что значение переменной может изменяться в любой момент времени внешними факторами, такими как другой поток или аппаратные устройства

- Без volatile: Компилятор может оптимизировать код, предполагая, что значение переменной не изменяется неожиданным образом, если это не указано в коде
- С volatile: Компилятор не будет кешировать значение переменной и будет каждый раз обращаться к памяти напрямую

```
volatile bool stop = false;

void worker() {
    while (!stop) {
        // Работаем, пока не будет сигнал на остановку
    }
}

int main() {
    std::thread t(worker);
    std::this_thread::sleep_for(std::chrono::seconds(1));
    stop = true;  // Устанавливаем флаг завершения
    t.join();
}
```

Ключевое слово mutable

Разрешает изменять члены класса, помеченные как mutable, даже если метод помечен как const

В контексте многопоточности полезно для изменения состояния объектов, которые могут быть разделены между потоками, при этом соблюдая семантику «константности» интерфейса

```
class ThreadSafeCounter {
private:
    mutable std::mutex mtx;  // Мьютекс можно изменять, даже если объект константный
    int counter = 0;

public:
    void increment() const {
        std::lock_guard<std::mutex> lock(mtx);  // Захватываем мьютекс
        ++counter;  // Изменяем счётчик
    }

    int get() const {
        std::lock_guard<std::mutex> lock(mtx);  // Захватываем мьютекс для защиты
        return counter;
    }
};
```

Связь между volatile, мьютексами и многопоточностью

Volatile **не защищает данные от гонок**, а только предотвращает оптимизацию компилятором

То есть, даже с использованием volatile, **нужно использовать мьютексы или атомарные операции** для обеспечения корректного доступа к данным в многопоточности

Мьютексы обеспечивают защиту от гонок данных, но их использование может приводить к накладным расходам (блокировка / разблокировка) и риску тупиков, если не обращаться с ними аккуратно

Mutable **используется для изменения данных**, помеченных как const, что может быть полезно при синхронизации доступа к данным через мьютексы в многопоточном коде

Работа приложения

Основные понятия процесса, ядра, потока

Однопоточное -> многопоточное ПО

И синхронное –> асинхронное

Основные проблемы многопоточности

Гонки данных, условия гонки и deadlock’и

Способы синхронизации данных

Как решение проблем многопоточности

Управление памятью и ресурсами в C++

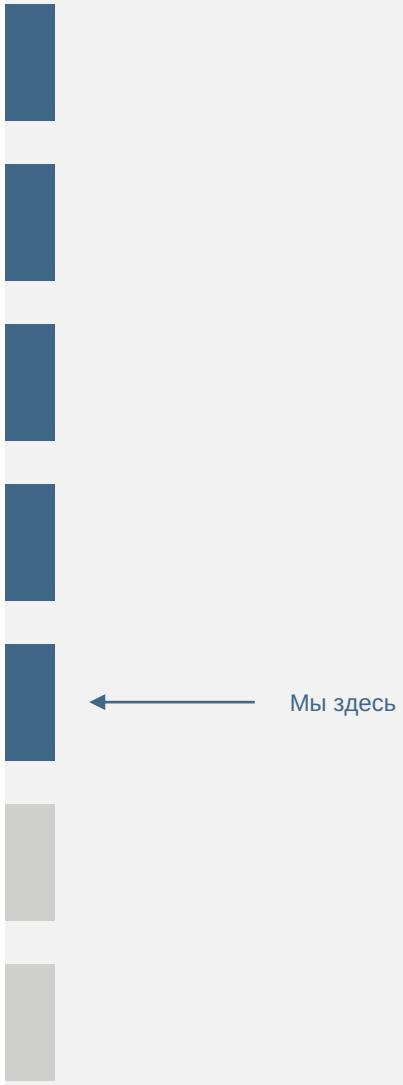
Основные советы и примеры

Сравнение принципов языков C++, Rust и Go

В контексте управления ресурсами

Заключение

Подведение итогов о реализации управления ресурсами



Стандартная библиотека потоков C++11

Представляет поток выполнения в программе

Создание потока:

- **Конструктор:**
  Принимает функцию и аргументы для выполнения в новом потоке
- **Метод join():**
  Ожидает завершения потока
- **Метод detach():**
  Отсоединяет поток, позволяя ему выполняться независимо.

```
void printMessage(const std::string& message) {
  std::cout << message << std::endl;
}

int main() {
  std::thread t(printMessage, "Hello from thread!");
  t.join(); // Ждем завершения потока
  return 0;
}
```

Средства межпоточной коммуникации

Атомарные операции – предоставляют методы для атомарного чтения, записи и модификации переменных

Памятные порядки:

- **std::memory_order_relaxed**: Нет синхронизации между потоками
- **std::memory_order_acquire** и **std::memory_order_release**: Синхронизация операций чтения и записи
- **std::memory_order_seq_cst**: Гарантирует последовательный порядок операций

Средства межпоточной коммуникации

Future и обещания

- **std::future**:
  Объект, представляющий результат асинхронной операции, который станет доступен в будущем
- **std::promise**
  Используется для задания значения или исключения, связанного с std::future
- **std::async**:
  Запускает функцию асинхронно и возвращает std::future

```
#include <future>

int computeSquare(int x) {
  return x * x;
}

int main() {
  std::future<int> result = std::async(std::launch::async,
computeSquare, 5);
  std::cout << "Result: " << result.get() << std::endl; //
Вывод: 25
  return 0;
}
```

Синхронизация доступа к общим ресурсам

```
class BankAccount {

private:
  int balance;
  std::mutex mtx;

public:
  BankAccount(int initial) : balance(initial) {}
  void deposit(int amount) {
      std::lock_guard<std::mutex> lock(mtx);
      balance += amount;
  }
  void withdraw(int amount) {
      std::lock_guard<std::mutex> lock(mtx);
      if (balance >= amount) {
          balance -= amount;
      } else {
          throw std::runtime_error("Insufficient
funds");
      }
  }
  int getBalance() {
      std::lock_guard<std::mutex> lock(mtx);
      return balance;
  }
};
```

Управление памятью

Проблемы

Указатели могут стать недействительными, если объект удален в другом потоке

Риск утечек памяти или двойного освобождения

Использование умных указателей

std::shared_ptr:

Автоматическое управление временем жизни объектов, разделяемых между потоками

std::unique_ptr:

Гарантирует владение объектом одним указателем

```
void processData(std::shared_ptr<Data> dataPtr) {
  // Обработка данных
}

int main() {
  auto data = std::make_shared<Data>();
  std::thread t1(processData, data);
  std::thread t2(processData, data);
  t1.join();
  t2.join();
  return 0;
}
```

! Передача параметров по значению или константной ссылке – избегать передачи сырых указателей

Исключения в многопоточных программах

Проблемы

Не обработанное в потоке исключение может привести к завершению программы

Стандарт C++ не определяет поведение для неотловленных исключений в потоках

Рекомендации

try-catch блоки

Обрабатывать исключения внутри потоков

std::promise и std::future

Передача исключений в основной поток

```cpp
void task(std::promise<void>&& promise) {
  try {
      // Некоторая операция, которая может бросить исключение
      throw std::runtime_error("Error in thread");
  } catch (...) {
      promise.set_exception(std::current_exception());
  }
}

int main() {
  std::promise<void> promise;
  std::future<void> result = promise.get_future();
  std::thread t(task, std::move(promise));

  try {
      result.get();
  } catch (const std::exception& e) {
      std::cout << "Caught exception: " << e.what() << std::endl;
  }
```

Производительность многопоточных программ

Факторы, влияющие на производительность

Overhead переключения контекста
Большое количество потоков может привести к частым переключениям, увеличивая накладные расходы

Синхронизация
Использование мьютексов и других механизмов может замедлить программу из-за блокировок

Конкуренция за ресурсы
Несколько потоков могут конкурировать за один и тот же ресурс (например, память или диск)

Оптимизация

Минимизация блокировок
Разрабатывать алгоритмы с минимальной необходимостью синхронизации

Использование атомарных операций
Когда возможно, использовать атомарные переменные вместо мьютексов

Пул потоков
Ограничение количества потоков и повторное использование существующих

Работа приложения

Основные понятия процесса, ядра, потока

Однопоточное -> многопоточное ПО

И синхронное –> асинхронное

Основные проблемы многопоточности

Гонки данных, условия гонки и deadlock’и

Способы синхронизации данных

Как решение проблем многопоточности

Управление памятью и ресурсами в C++

Основные советы и примеры

Сравнение принципов языков C++, Rust и Go

В контексте управления ресурсами

Мы здесь

Заключение

Подведение итогов о реализации управления ресурсами

Rust: Система владения памятью

Borrow Checker
Механизм компилятора, который проверяет правила владения и заимствования

Владение

- Каждый ресурс имеет владельца
- В один момент времени может быть только один владелец
- Ресурс освобождается, когда владелец выходит из области видимости

Заимствование

- Ссылки на данные могут быть immutable или mutable, но не одновременно
- **Immutable-ссылки (&T):** Неограниченное количество
- **Mutable-ссылка (&mut T):** Только одна

Преимущества:

Предотвращение утечек памяти и гонок данных на этапе компиляции

Нет необходимости в сборщике мусора

Rust: Средства параллельного программирования

Безопасный параллелизм

Rust гарантирует, что параллельные программы не имеют гонок данных

Каналы

- Используются для передачи данных между потоками
- Обеспечивают безопасную коммуникацию без необходимости синхронизации

Асинхронность

- **async/await:**
  Позволяет писать асинхронный код, который выглядит как синхронный
- **Executor:**
  Требуется для запуска асинхронных задач (например, tokio или async-std)

```rust
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
  let handle = tokio::spawn(async {
      sleep(Duration::from_secs(1)).await;
      println!("Hello from async task");
  });

  handle.await.unwrap();
}
```

Rust: Сравнение с C++

Параметр	Rust	C++
Конкуренция	Безопасный параллелизм Библиотека Tokio предоставляет мощные инструменты для создания высокопроизводительных сетевых приложений	Поддержка многопоточности, но требует тщательного управления ресурсами и синхронизацией для предотвращения гонок данных Асинхронное программирование неудобнее чем в Rust
Безопасность памяти	Безопасность памяти Предотвращение таких ошибок, как использование освобожденной памяти или двойная свободная память	Отсутствие автоматической проверки безопасности памяти может привести к различным ошибкам, связанным с управлением памятью
Производительность	Высокая производительность, сопоставимая с C++, благодаря системе владения и отсутствию сборщика мусора	Максимальная производительность среди языков общего назначения, но требует глубокого понимания низкоуровневых деталей и тщательной оптимизации
Использование памяти	Низкое использование памяти благодаря системе владения и отсутствию сборщика мусора	Может быть выше из-за необходимости ручного управления памятью и потенциальных утечек
Кривая обучения	Более крутая из-за системы владения, но обеспечивает большую безопасность – требует времени для освоения концепций заимствования и жизненного цикла	Менее крутая, но требует глубоких знаний низкоуровневых деталей и принципов программирования
Уровень абстракции	Средний Предоставляет абстракции для безопасного управления памятью и параллелизмом, но сохраняет низкоуровневый контроль	Низкий уровень абстракции, дает большой контроль над аппаратными ресурсами
Контроль над ресурсами	Высокий контроль над ресурсами благодаря системе владения и возможности низкоуровневого программирования	Максимальный контроль над ресурсами, но требует глубоких знаний и осторожности
Безопасность	Высокая безопасность, особенно в отношении безопасности памяти	Требует от разработчика высокой дисциплины для предотвращения ошибок, связанных с памятью и параллелизмом

Go: Go-routines и каналы

Go-routines

- Легковесные потоки, управляемые рантаймом Go
- Создаются с помощью ключевого слова go

```
func sayHello() {
    fmt.Println("Hello from Go-routine")
}

func main() {
    go sayHello()
    time.Sleep(time.Second) // Ожидание
завершения Go-routine
}
```

Каналы

- Используются для передачи данных между Go-routines
- Обеспечивают синхронизацию и безопасность при передаче данных

```
func main() {
    messages := make(chan string)

    go func() {
        messages <- "Hello from Go-routine"
    }()

    msg := <-messages
    fmt.Println(msg)
}
```

Go: Сравнение с C++

Параметр	Go	C++
Конкуренция	Goroutines и каналы предоставляют легкий и безопасный способ для параллельного программирования Встроенная поддержка асинхронного ввода-вывода	Многопоточность, но требует ручного управления ресурсами и синхронизацией Асинхронное программирование возможно, но менее удобно
Безопасность памяти	Сборщик мусора и отсутствие указателей на сырые байты предотвращают большинство ошибок, связанных с памятью	Отсутствие автоматического управления памятью требует осторожности и может привести к утечкам памяти и другим проблемам
Производительность	Высокая производительность, особенно в I/O-bound задачах Компилятор Go оптимизирует код для современных процессоров	Максимальная производительность среди языков общего назначения, но требует тщательной оптимизации
Использование памяти	Обычно эффективное использование памяти благодаря сборщику мусора Однако, в некоторых случаях может быть выше, чем в C++	Может быть более эффективным при ручном управлении памятью, но требует большего внимания
Кривая обучения	Относительно пологая Синтаксис прост и лаконичен	Более крутая кривая обучения, особенно для начинающих программистов
Уровень абстракции	Средний Предоставляет абстракции для параллельного программирования и управления памятью	Низкий уровень абстракции, дает большой контроль над аппаратными ресурсами
Контроль над ресурсами	Меньший контроль над ресурсами по сравнению с C++, но достаточный для большинства задач	Максимальный контроль над ресурсами
Безопасность	Высокая безопасность благодаря встроенным механизмам безопасности памяти и отсутствию указателей на сырые байты	Требует от разработчика высокой дисциплины для предотвращения ошибок

Rust vs Go

Параметр	Rust	Go
Конкуренция	Безопасный параллелизм Библиотека Tokio предоставляет мощные инструменты для создания высокопроизводительных сетевых приложений	Goroutines и каналы предоставляют легкий и безопасный способ для параллельного программирования Встроенная поддержка асинхронного ввода-вывода
Безопасность памяти	Безопасность памяти Предотвращение таких ошибок, как использование освобожденной памяти или двойная свободная память	Сборщик мусора и отсутствие указателей на сырые байты предотвращают большинство ошибок, связанных с памятью
Производительность	Высокая производительность, сопоставимая с C++, благодаря системе владения и отсутствию сборщика мусора	Высокая производительность, особенно в I/O-bound задачах Компилятор Go оптимизирует код для современных процессоров
Использование памяти	Низкое использование памяти благодаря системе владения и отсутствию сборщика мусора	Обычно эффективное использование памяти благодаря сборщику мусора Однако, в некоторых случаях может быть выше, чем в C++
Кривая обучения	Более крутая из-за системы владения, но обеспечивает большую безопасность – требует времени для освоения концепций заимствования и жизненного цикла	Относительно пологая Синтаксис прост и лаконичен
Уровень абстракции	Средний Предоставляет абстракции для безопасного управления памятью и параллелизмом, но сохраняет низкоуровневый контроль	Средний Предоставляет абстракции для параллельного программирования и управления памятью
Контроль над ресурсами	Высокий контроль над ресурсами благодаря системе владения и возможности низкоуровневого программирования	Меньший контроль над ресурсами по сравнению с C++, но достаточный для большинства задач
Безопасность	Высокая безопасность, особенно в отношении безопасности памяти	Высокая безопасность благодаря встроенным механизмам безопасности памяти и отсутствию указателей на сырые байты

Работа приложения

Основные понятия процесса, ядра, потока

Однопоточное -> многопоточное ПО

И синхронное –> асинхронное

Основные проблемы многопоточности

Гонки данных, условия гонки и deadlock’и

Способы синхронизации данных

Как решение проблем многопоточности

Управление памятью и ресурсами в C++

Основные советы и примеры

Сравнение принципов языков C++, Rust и Go

В контексте управления ресурсами

Заключение

Подведение итогов о реализации управления ресурсами

Мы здесь

Общая сравнительная таблица

Подход	Описание	Когда применять	Преимущества	Недостатки
Синхронное однопоточное	Последовательное выполнение инструкций в одном потоке	Простые скрипты, утилиты	Простота реализации	Низкая производительность при длительных операциях
Синхронное многопоточное	Параллельное выполнение независимых задач в разных потоках	Задачи, требующие параллельной обработки	Повышенная производительность на многоядерных процессорах	Сложность управления потоками, синхронизации
Асинхронное однопоточное	Последовательное выполнение с возможностью переключения между задачами	I/O-bound задачи	Высокая пропускная способность	Сложность реализации, управление ресурсами
Асинхронное многопоточное	Сочетание асинхронности и параллелизма	Большинство современных серверных приложений	Высокая производительность и масштабируемость	Сложность реализации, управление ресурсами
Параллелизм	Разбиение задачи на независимые подзадачи, выполняемые одновременно	Вычислительно интенсивные задачи	Высокая производительность	Сложность разбиения задачи
Корутины	Легковесные потоки, управляемые программистом	Асинхронное программирование, создание высококонкурентных систем	Эффективное использование ресурсов, гибкость	Требуют специальной поддержки языка или библиотеки

Спасибо за внимание!
Ваши вопросы?