Модерни техники за паралелизъм и асинхронност в C++

Thread Pools, Coroutines и Publisher/Subscriber Pattern

Алекс Цветанов, КСИ, ФКСТ

Паралелно програмиране

22 октомври 2025 г.

План на лекцията

Мотивация

Защо е важно?

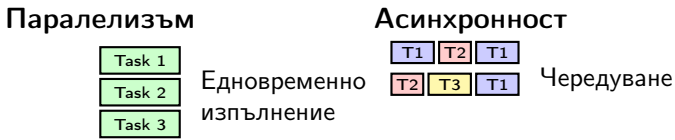
- Съвременните CPU имат множество ядра
- I/O операциите блокират нишки
- Нуждата от high-throughput системи
- Скалируемост и ефективност

Нашият фокус днес:

1. Thread Pools (паралелизъм)
2. Coroutines (асинхронност)
3. Publisher/Subscriber (комуникация)
4. Lock-free структури
5. Практически имплементации

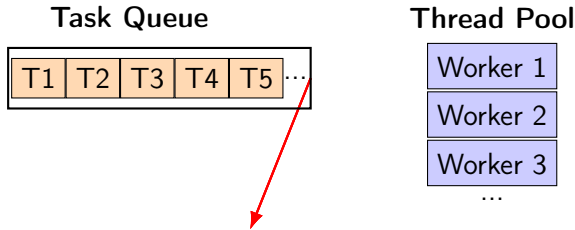
Как да използваме ресурсите оптимално?

Паралелизъм vs Асинхронност



Thread Pools: истински паралелизъм (multiple cores)
Coroutines: кооперативна многозадачност (single core OK)

Thread Pool: Концепция



Идея: Фиксиран брой worker нишки, които обработват задачи от споделена опашка

Предимства:

- Контрол върху броя нишки
- Преизползване на ресурси
- По-добра производителност

Защо Thread Pool? Цената на създаване на нишки

Проблем: Създаването и унищожаването на std::thread обекти е скъпа операция!

Наивен подход:

```
// Създаване на нова нишка
// за ВСЯКА задача
for (auto& task : tasks) {
    std::thread t([&task]() {
        process(task);
    });
    t.detach(); // Опасно!
}

// Проблеми:
// 1. System call за създаване
// 2. Заделяне на stack памет
// 3. Context switching overhead
// 4. Унищожаване на ресурси
// 5. Няма контрол върху броя
```

Thread Pool подход:

```
// Еднократно създаване
ThreadPool pool(
    std::thread::
    hardware_concurrency()
);

// Многократно използване
for (auto& task : tasks) {
    pool.enqueue([&task]() {
        process(task);
    });
}
pool.wait_all();

// Предимства:
// 1. Нишките се създават веднъж
// 2. Преизползване на ресурси
// 3. Контролиран паралелизъм
// 4. По-добра cache locality
```

Benchmark: Създаване на 1000 нишки $\approx$ 50-100ms, Thread Pool $\approx$ 1-2ms

Overhead на създаване на нишки

Заключение: Thread Pool е 50x по-бърз за множество малки задачи!

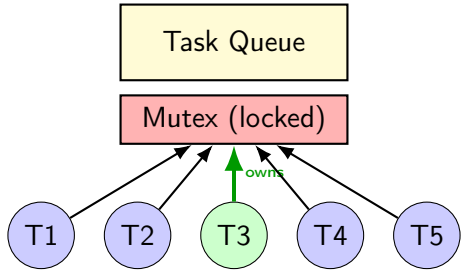
Предизвикателство: Синхронизация на опашката

Проблем: Споделена опашка изисква lock за защита от race conditions

```
class ThreadPool {
    std::queue<std::function<void()>> tasks;
    std::mutex queue_mutex;  // <-- КРИТИЧНА СЕКЦИЯ
    std::condition_variable condition;

    void worker_thread() {
        while (true) {
            std::function<void()> task;
            {
                std::unique_lock<std::mutex> lock(queue_mutex);  // LOCK!
                condition.wait(lock, [this] { return stop || !tasks.empty(); });
                if (stop && tasks.empty()) return;
                task = std::move(tasks.front());
                tasks.pop();
            }  // Unlock тук
            task();  // Изпълнение извън lock-а
        }
    }
};
```

Bottleneck: Всички нишки се конкурират за един mutex!

Проблеми със standard mutex



Contention: 4 нишки чакат, само 1 работи!

Последици:

- Context switching
- Cache invalidation
- Намалена throughput при много нишки

Lock-Free структури: Концепция

Идея: Използване на атомарни операции вместо mutex

Lock-based:

- × Blocking
- × Contention
- × Priority inversion
- × Deadlock риск
- ✓ По-лесна имплементация

Lock-free:

- ✓ Non-blocking
- ✓ No contention
- ✓ Wait-free progress
- ✓ По-висока throughput
- × Сложна имплементация

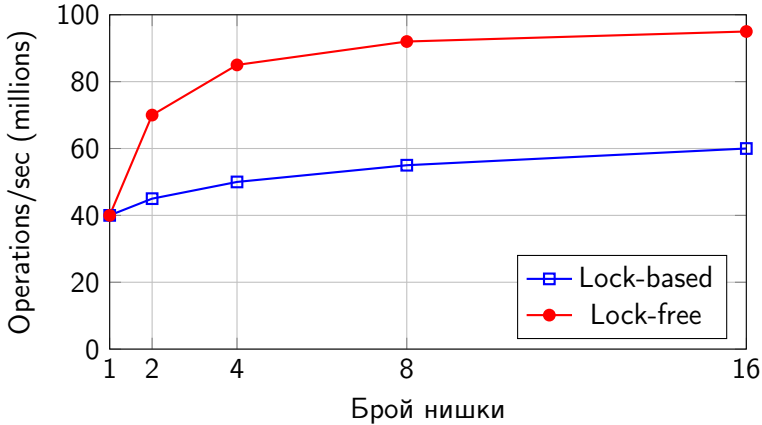
Ключови техники:

- Compare-And-Swap (CAS): `std::atomic::compare_exchange_weak`
- Memory ordering: `memory_order_acquire`, `memory_order_release`
- ABA problem решения (tagged pointers, hazard pointers)

Lock-Free Queue: Концепция

```
template<typename T>
class LockFreeQueue {
    struct Node {
        std::shared_ptr<T> data;
        std::atomic<Node*> next;
        Node() : next(nullptr) {}
    };
    std::atomic<Node*> head;
    std::atomic<Node*> tail;
public:
    void enqueue(T value) {
        auto new_node = new Node();
        new_node->data = std::make_shared<T>(std::move(value));
        Node* old_tail = tail.load(std::memory_order_relaxed);
        while (!tail.compare_exchange_weak(old_tail, new_node,
               std::memory_order_release, std::memory_order_relaxed)) {
            // Retry ако друга нишка промени tail
        }
        old_tail->next.store(new_node, std::memory_order_release);
    }
    std::shared_ptr<T> dequeue() {
        Node* old_head = head.load(std::memory_order_acquire);
        while (old_head && !head.compare_exchange_weak(old_head,
               old_head->next.load(), std::memory_order_release)) {
            // Retry
        }
        return old_head ? old_head->data : nullptr;
    }
};
```

Lock-Free vs Lock-Based: Performance



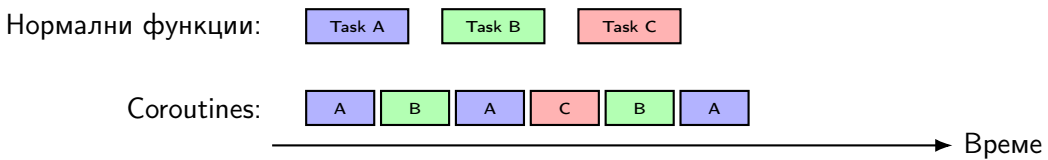
Забележка: Lock-free структури scaling-ват много по-добре!

Thread Pool с Lock-Based Queue

```cpp
class ThreadPool {
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> tasks;
    std::mutex queue_mutex;
    std::condition_variable condition;
    bool stop = false;

public:
    ThreadPool(size_t threads) {
        for (size_t i = 0; i < threads; ++i) {
            workers.emplace_back([this] {
                while (true) {
                    std::function<void()> task;
                    {
                        std::unique_lock<std::mutex> lock(queue_mutex);
                        condition.wait(lock, [this] {
                            return stop || !tasks.empty();
                        });
                        if (stop && tasks.empty()) return;
                        task = std::move(tasks.front());
                        tasks.pop();
                    }
                    task(); // Изпълнение извън lock-а
                }
            });
        }
    }
    // enqueue(), destructor...
};
```

Coroutines: Какво са?

Определение: Функции, които могат да suspend-ват и resume-ват изпълнението си



Ключови думи в C++20:

- `co_await` – suspend и чакане на резултат
- `co_yield` – връщане на междинна стойност
- `co_return` – завършване на coroutine

Threads vs Coroutines

Threads (OS-level):

- Управлявани от ОС
- Преемптивен scheduling
- Тежки (MB stack)
- Context switch $\approx$1-10μs
- Паралелно изпълнение
- Използват множество ядра

Coroutines (user-level):

- Управлявани от програмата
- Кооперативен scheduling
- Леки (KB state)
- Context switch $\approx$10-100ns
- Конкурентно, не паралелно
- Едно ядро (освен ако...)

Идеално: Coroutines за I/O, Threads за CPU-intensive работа

Coroutines: Механизъм на работа

Традиционна функция:

- Stack frame се създава при извикване
- Изпълнение до return
- Stack frame се унищожава
- Невъзможно повторно влизане

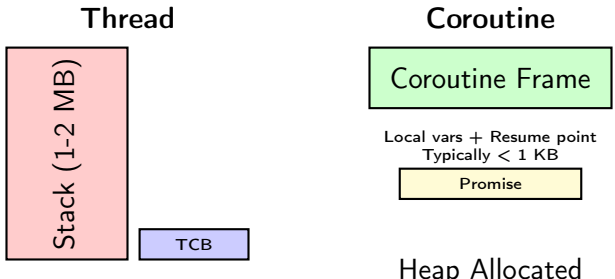
Coroutine:

- Състоянието се съхранява в heap (coroutine frame)
- `co_await` запазва локални променливи и позиция
- Контролът се връща на caller-а
- По-късно се resume-ва от същата позиция

Защо са по-евтини от threads?

1. Без system calls за context switching
2. Малко състояние (само локални променливи)
3. Без thread stack (1-2 MB спестени)
4. Compiler оптимизации (inline възможни)

Memory Layout: Thread vs Coroutine



Резултат: Можем да имаме милиони coroutines, но само стотици threads!

Предизвикателство: Dangling References и Pointers

ВАЖНО: Локални променливи в coroutine са БЕЗОПАСНИ (в coroutine frame)!

```
Task<int> safe_coroutine() {
    int local_var = 42;  // OK! В coroutine frame
    co_await some_async_operation();
    return local_var;  // БЕЗОПАСНО!
}
```

Проблем 1: Указатели/референции към ВЪН от coroutine

```
Task<void> dangerous_coroutine(int* ptr) {  // Външен указател!
    co_await some_async_operation();
    *ptr = 42;  // ОПАСНО! ptr може да е невалиден
}

void caller() {
    int x = 10;
    auto task = dangerous_coroutine(&x);  // Адрес от caller stack
    // Ако caller приключи преди coroutine да resume-не...
    // x вече не съществува! -> Dangling pointer
}
```

Предизвикателство: Dangling References и Pointers

Проблем 2: Referencing локални променливи от caller; Lambda capture by reference

```
1  void caller() {
2      std::vector<int> data = {1, 2, 3, 4, 5};
3
4      auto task = [&data]() -> Task<void> {  // Capture by reference!
5          co_await async_operation();
6          // data е на stack-а на caller()
7          // Ако caller() приключи преди resume -> CRASH!
8          for (int x : data) { std::cout << x; }  // BUS ERROR!
9      }();
10
11     // caller() завършва, data се унищожава
12     // но coroutine все още го реферира!
13 }
```

Обобщение:

- ✓ Локални променливи В coroutine → coroutine frame (SAFE)
- × Указатели/референции КЪМ външни данни → dangling (DANGEROUS)
- ✓ Използвайте capture by value: `[data]` вместо `[&data]`

Решение: Как да поставим данни в coroutine frame?

Правило: Всички променливи дефинирани в coroutine тяло автоматично се съхраняват в heap-allocated coroutine frame

Грешно (stack reference):

```
Task<void> bad_example() {
    std::string temp = "data";
    std::string& ref = temp;

    co_await switch_context();

    // ref може да е невалиден!
    std::cout << ref;
}

// Външна променлива
void caller() {
    int x = 42;
    auto task = [&x]() -> Task<void> {
        co_await something();
        // x е dangling reference!
        use(x);
    }();
}
```

Правилно (coroutine frame):

```
Task<void> good_example() {
    // Копие в coroutine frame
    std::string data = "data";

    co_await switch_context();

    // data е винаги валиден
    std::cout << data;
}

// Правилно capture
void caller() {
    int x = 42;
    auto task = [x]() -> Task<void> {
        // x е копирано (by value)
        co_await something();
        use(x); // SAFE!
    }();
}
```

Ключът: Компилаторът автоматично премества локални променливи в coroutine frame. Проблемът идва от references/pointers към външни данни!

Техники за безопасно управление на данни

1. Копиране вместо referencing

```
Task<void> process(std::string data) {
    // by value = copy in frame
    co_await async_operation();
    use(data);  // Safe
}
```

2. std::move за големи обекти

```
Task<void> process(std::vector<int> vec) {
    auto local_vec = std::move(vec);
    // Move в coroutine frame
    co_await async_operation();
    use(local_vec);
}
```

3. shared_ptr за споделени данни

```
Task<void> process(std::shared_ptr<BigData>
     data) {
    // shared_ptr копира контролния блок,
    // не данните
    co_await async_operation();
    use(*data);  // Safe - data е жив докато
                 // има reference
}
```

4. Promise type с custom allocation

```
struct promise_type {
    void* operator new(size_t size) {
      // Custom allocator за coroutine frame
      return my_pool_allocator.allocate(size);
    }
    // Всички локални променливи + promise
    // отиват тук
};
```

Предизвикателство: Debugging и Stack Traces

Проблем: Coroutines нямат традиционен stack trace

```
Task<void> level3() {
    co_await std::suspend_always{};
    throw std::runtime_error("Error!");  // Къде е stack trace-а?
}

Task<void> level2() {
    co_await level3();
}

Task<void> level1() {
    co_await level2();
}

// При exception, debugger показва само текущата coroutine,
// не целия "async call stack"
```

Решения:

- Използвайте coroutine-aware debuggers (LLDB, Visual Studio)
- Логване на входни/изходни точки
- Custom promise types с tracing

Проста Coroutine имплементация (C++20)

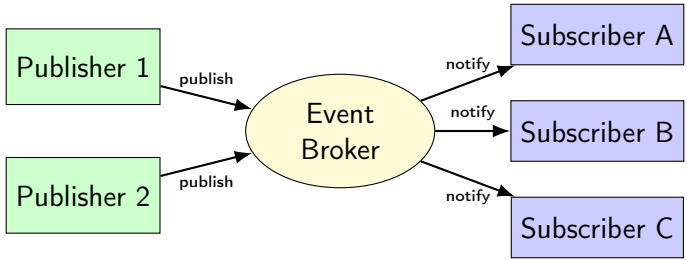
```cpp
#include <coroutine>
#include <iostream>

template<typename T>
struct Task {
    struct promise_type {
        T value;
        Task get_return_object() {
            return Task{std::coroutine_handle<promise_type>::from_promise(*this)};
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() { std::terminate(); }
        void return_value(T v) { value = v; }
    };
    std::coroutine_handle<promise_type> handle;
    T get() { return handle.promise().value; }
    ~Task() { if (handle) handle.destroy(); }
};
Task<int> async_computation() {
    std::cout << "Starting...\n";
    co_await std::suspend_always{};  // Suspend тук
    std::cout << "Resuming...\n";
    co_return 42;
}
int main() {
    auto task = async_computation();  // Стартира, спира на co_await
    task.handle.resume();  // Продължава изпълнението
    std::cout << "Result: " << task.get() << "\n";
}
```

Реален пример: Async I/O с Coroutines

```cpp
Task<std::string> async_read_file(const std::string& filename) {
    // Simulate async file reading
    co_await async_open(filename);
    std::string content;

    while (true) {
        auto chunk = co_await async_read_chunk();
        if (chunk.empty()) break;
        content += chunk;
    }

    co_await async_close();
    co_return content;
}

Task<void> process_files() {
    // Множество паралелни четения (но в една нишка!)
    auto file1 = async_read_file("data1.txt");
    auto file2 = async_read_file("data2.txt");
    auto file3 = async_read_file("data3.txt");

    // Чакаме всички
    auto content1 = co_await file1;
    auto content2 = co_await file2;
    auto content3 = co_await file3;

    // Обработка...
    std::cout << "Total size: " << content1.size() + content2.size() + content3.size();
}
```

Предимство: Хиляди едновременни I/O операции без хиляди threads!

Publisher/Subscriber: Design Pattern



Определение: Architectural pattern за асинхронна комуникация
Компоненти:

- **Publisher** – генерира събития/съобщения
- **Subscriber** – регистрира интерес и обработва
- **Event Broker** – посредник (loose coupling)

Защо Pub/Sub Pattern?

Проблеми на директната комуникация:

- Tight coupling между компоненти
- Трудно масштабиране
- Промяна в един компонент засяга много други

Предимства на Pub/Sub:

- ✓ **Loose coupling** – publishers не знаят за subscribers
- ✓ **Scalability** – добавяне на subscribers без промяна
- ✓ **Flexibility** – динамична (не)регистрация
- ✓ **Асинхронност** – non-blocking communication

vs Observer Pattern:

- Observer = synchronous, директна връзка
- Pub/Sub = asynchronous, чрез посредник (broker)

Къде се използва Pub/Sub?

Event-Driven Архитектури

- GUI Applications (button clicks, window events)
- Game engines (collision events, state changes)

Microservices Communication

- Distributed systems (RabbitMQ, Apache Kafka)
- Service orchestration

Real-Time Data Processing

- Stock market data feeds
- IoT sensor networks
- Log aggregation systems

Reactive Programming

- RxCpp (Reactive Extensions)
- Async data streams

Примери от практиката

Financial Trading System:

1. Market data feed (publisher)
2. Risk engine (subscriber)
3. Trading strategies (subscribers)
4. Compliance monitor (subscriber)
5. Logging system (subscriber)

Всички компоненти получават същите данни, но ги обработват различно

IoT Smart Home:

1. Temperature sensor (publisher)
2. Heating system (subscriber)
3. Mobile app (subscriber)
4. Data logger (subscriber)
5. Alert system (subscriber)

Едно събитие тригерва множество реакции

Предизвикателство 1: Гаранции за доставка

Проблем: Какво ако subscriber не получи съобщението?

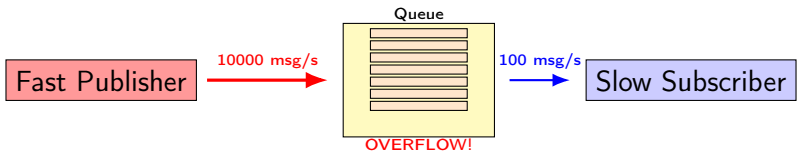
Опции за доставка:

1. **At-most-once** – "fire and forget"
   - Най-бързо, но може да се загуби
   - Подходящо за некритични данни (metrics)
2. **At-least-once** – retry при неуспех
   - Може да има дубликати
   - Нуждае се от idempotent обработка
3. **Exactly-once** – гарантирана еднократна доставка
   - Най-сложно, изисква транзакции
   - Критични системи (payments, medical)

Trade-off: Надеждност vs Performance

Предизвикателство 2: Backpressure

Проблем: Publisher генерира събития по-бързо, отколкото subscribers могат да обработят



Решения:

- Buffering с bounded queue
- Dropping (отпадане на стари съобщения)
- Throttling (забавяне на publisher-а)
- Паралелизация на subscribers (нашата тема!)

Предизвикателство 3: Ordering

Проблем: Запазване на реда на съобщенията при паралелна обработка

```
// Subscriber
с thread pool може да получи: 2, 1, 3 // заради паралелна обработка!
```

Решения:

1. **Single-threaded processing** – губим паралелизъм
2. **Partition-based** – групи съобщения с гарантиран ред
3. **Sequence numbers** – преподреждане след обработка
4. **Per-key ordering** – само свързани съобщения се подреждат

Проста Pub/Sub имплементация

```cpp
template<typename Event>
class EventBroker {
    using Subscriber = std::function<void(const Event&)>;
    using SubscriberId = size_t;

    std::unordered_map<SubscriberId, Subscriber> subscribers;
    std::mutex mutex;
    SubscriberId next_id = 0;

public:
    SubscriberId subscribe(Subscriber&& callback) {
        std::lock_guard<std::mutex> lock(mutex);
        auto id = next_id++;
        subscribers[id] = std::move(callback);
        return id;
    }

    void unsubscribe(SubscriberId id) {
        std::lock_guard<std::mutex> lock(mutex);
        subscribers.erase(id);
    }

    void publish(const Event& event) {
        std::lock_guard<std::mutex> lock(mutex);
        for (auto& [id, subscriber] : subscribers) {
            subscriber(event);  // Синхронно извикване
        }
    }
};
```

Проблем: publish() блокира до обработка на всички subscribers!

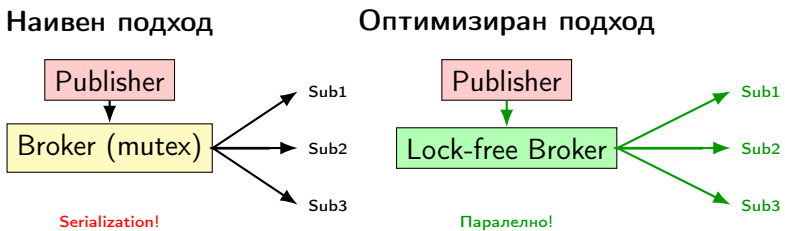
Асинхронна имплементация с Thread Pool

```
template<typename Event>
class AsyncEventBroker {
    using Subscriber = std::function<void(const Event&)>;
    std::vector<Subscriber> subscribers;
    ThreadPool pool;
    std::mutex mutex;

public:
    AsyncEventBroker(size_t num_threads = std::thread::hardware_concurrency())
        : pool(num_threads) {}

    void subscribe(Subscriber&& callback) {
        std::lock_guard<std::mutex> lock(mutex);
        subscribers.push_back(std::move(callback));
    }

    void publish(const Event& event) {
        std::lock_guard<std::mutex> lock(mutex);
        // Всеки subscriber се обработва в отделна задача
        for (auto& subscriber : subscribers) {
            pool.enqueue([subscriber, event]() {
                subscriber(event);  // Паралелно изпълнение!
            });
        }
    }  // publish() не чака обработката
};
```

Предимство: Non-blocking publish(), паралелна обработка!

Паралелизация на subscriber dispatch

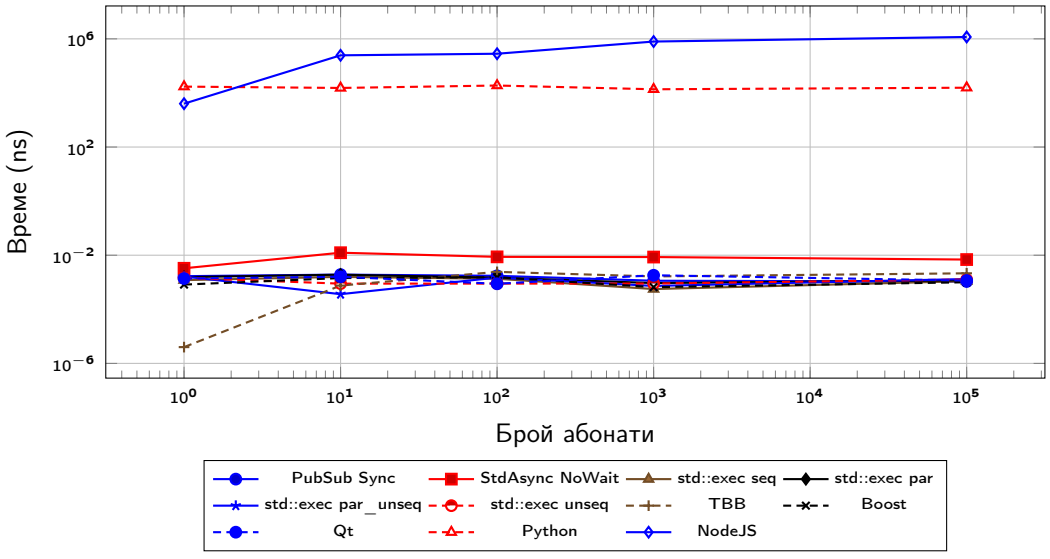
Проблем: Наивната имплементация има serialization bottleneck



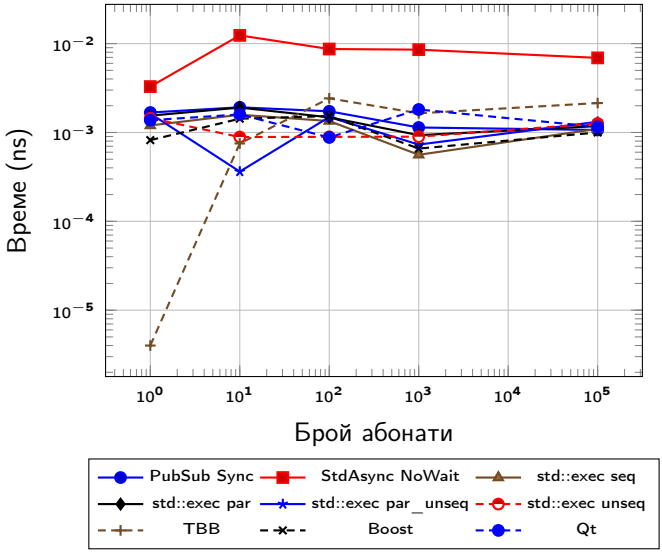
Техники за оптимизация:

- Lock-free subscriber list (RCU - Read-Copy-Update)
- Per-subscriber queue вместо centralized
- Wait-free publish операции
- Batch processing на множество събития

Сравнителна графика: Всички имплементации



Сравнителна графика: C++ имплементации



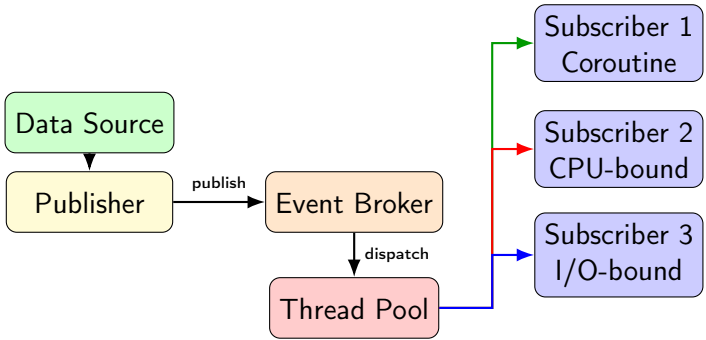
oneTBB:

```
template<auto Event, typename... Args>
bool Publisher::emit_tbb_async(Args
      &&... args) {
 auto* handler = get_handler<Event>();
 tbb::parallel_for_each(
  handler->callbacks.begin(),
  handler->callbacks.end(),
  [&](const auto& cb) {
   cb(std::forward<Args>(args)...);
  });
 return true;
}
```

std::execution:

```
template<auto Event, typename... Args>
bool Publisher::emit_async(std::
      execution::parallel_policy policy,
       Args&&... args) {
 auto* handler = get_handler<Event>();
 std::for_each(policy,
  handler->callbacks.begin(),
  handler->callbacks.end(),
  [&](const auto& cb) {
    cb(std::forward<Args>(args)...);
  });
 return true;
}
```

Интеграция: Thread Pool + Coroutines + Pub/Sub



Hybrid подход:

- Thread Pool за паралелна обработка (CPU-intensive)
- Coroutines за I/O операции (networking, disk)
- Pub/Sub за декуплиране и организация

Кога какво да използваме?

Задача	Техника	Защо?
CPU-intensive	Thread Pool	Използва всички ядра
I/O операции	Coroutines	Ниски overhead, милиони connections
Event routing	Pub/Sub	Loose coupling
High-throughput	Lock-free + Pool	Минимизира contention
Mixed workload	Hybrid	Комбинира силни страни

Практически съвет:

1. Измерете bottleneck-овете (профилиране!)
2. Не оптимизирайте преждевременно
3. Комбинирайте техники според нуждите

Обобщение

Какво научихме днес:

1. **Thread Pools**
   - Преизползване на нишки за ефективност
   - Lock-based vs lock-free имплементации
   - Цената на създаване на threads (50-100x!)
2. **Coroutines**
   - Леки, кооперативни задачи
   - Идеални за I/O-bound workloads
   - 100x по-евтини от threads (KB vs MB)
3. **Publisher/Subscriber**
   - Decoupling чрез event broker
   - Паралелизирано разпращане на съобщения
   - Препратка към научна работа
4. **Комбиниране**
   - Hybrid архитектури за реални системи

Връзка с курса

Модул 2: Асинхронно програмиране с .NET

- **Лекция 2-2:** Подходи за асинхронен код → Thread Pools (C++)
- **Лекция 2-3:** Async-await шаблон → Coroutines (C++20)
- **Лекция 2-4:** Паралелен достъп до данни → Lock-free structures
- **Лекция 2-1:** Оптимизация и performance → Benchmarks (днес)

C++ vs .NET паралели:

- .NET Task Parallel Library ↔ C++ Thread Pool
- .NET async/await ↔ C++ coroutines (co_await)
- .NET Concurrent Collections ↔ C++ lock-free queues
- .NET Events ↔ C++ Publisher/Subscriber pattern

Тази лекция разширява:

- Low-level имплементации на високо-ниво концепции
- Performance considerations (защо Thread Pool вместо threads?)
- Практически шаблони за комуникация (Pub/Sub)

Препоръчителна литература

Книги:

- "C++ Concurrency in Action" – Anthony Williams
- "The Art of Multiprocessor Programming" – Herlihy & Shavit
- "Programming with POSIX Threads" – David Butenhof

Papers & Talks:

- "Lock-Free Data Structures" – Michael & Scott
- "RCU (Read-Copy-Update)" – McKenney
- "Optimizing Asynchronous Event Dispatch in Modern C++ Publish/Subscribe Systems" – Alex Tsvetanov

Lock-Free Queue имплементации и talks:

- "User API and C++ Implementation of MPMC Lock Free Queue" – Erez Strauss, CppCon 2024
- "Building a Lock-free MPMC Queue for Tcmalloc" – Matt Kulukundis, CppCon 2021
- github.com/DNedic/lockfree – Djordje Nedic, Pablo Duboue, Lennart Schierling

Online ресурси:

- cppreference.com – coroutines, atomics, memory ordering
- preshing.com – lock-free programming blog
- 1024cores.net – Dmitry Vyukov's concurrency blog

Въпроси?

Благодаря за вниманието!

Контакт: atsvetanov@tu-sofia.bg | GitHub: github.com/Alex-Tsvetanov