

Модерни техники за паралелизъм и асинхронност в C++

Thread Pools, Coroutines и Publisher/Subscriber Pattern

Алекс Цветанов, КСИ, ФКСТ, 121222225

Паралелно програмиране

21 октомври 2025 г.

1 Увод: Паралелизъм vs Асинхронност

2 Thread Pools

- Концепция
- Защо е необходим?
- Предизвикателства
- Lock-Free Queues
- Примерна имплементация

3 Coroutines

- Концепция
- Как работят и защо са по-евтини?
- Предизвикателства
- Примерна имплементация

4 Publisher/Subscriber Pattern

- Концепция
- Приложения
- Предизвикателства
- Примерна имплементация
- Паралелизирано разпращане

5 Комбиниране на техниките

6 Заключение

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

Паралелизъм



Едновременно
изпълнение

Асинхронност

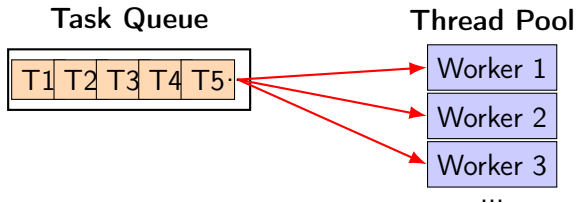


Чередуване

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!
Thread Pool подход:

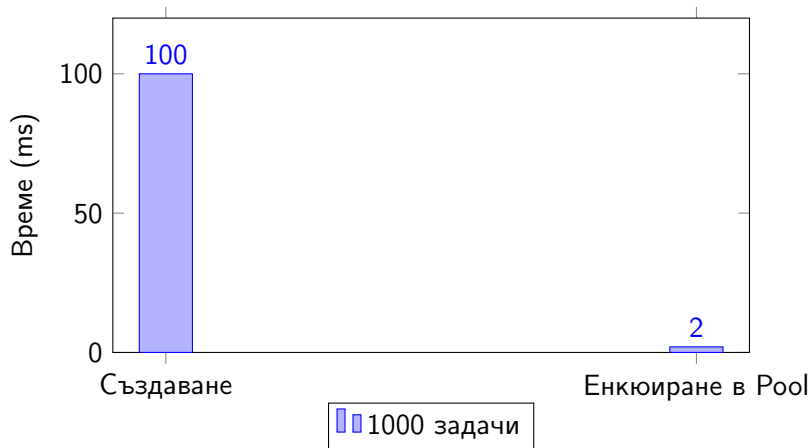
Наивен подход:

```
1 // Създаване на нова нишка
2 // за ВСЯКА задача
3 for (auto& task : tasks) {
4     std::thread t([&task]() {
5         process(task);
6     });
7     t.detach(); // Опасно!
8 }
9
10 // Проблеми:
11 // 1. System call за създаване
12 // 2. Заделяне на stack памет
13 // 3. Context switching overhead
14 // 4. Унищожаване на ресурси
15 // 5. Няма контрол върху броя
```

```
1 // Еднократно създаване
2 ThreadPool pool(
3     std::thread::
4     hardware_concurrency()
5 );
6
7 // Многократно използване
8 for (auto& task : tasks) {
9     pool.enqueue([&task]() {
10         process(task);
11     });
12 }
13 pool.wait_all();
14
15 // Предимства:
16 // 1. Нишките се създават веднъж
17 // 2. Преизползване на ресурси
18 // 3. Контролиран паралелизъм
19 // 4. По-добра cache locality
```

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

Overhead на създаване на нишки



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

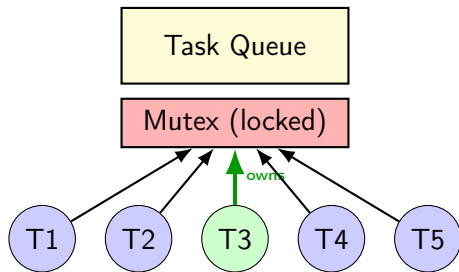
Предизвикателство: Синхронизация на опашката

Проблем: Споделена опашка изисква lock за защита от race conditions

```
1 class ThreadPool {
2     std::queue<std::function<void()>> tasks;
3     std::mutex queue_mutex; // <-- КРИТИЧНА СЕКЦИЯ
4     std::condition_variable condition;
5
6     void worker_thread() {
7         while (true) {
8             std::function<void()> task;
9             {
10                 std::unique_lock<std::mutex> lock(queue_mutex); // LOCK!
11                 condition.wait(lock, [this] { return stop || !tasks.empty(); });
12                 if (stop && tasks.empty()) return;
13                 task = std::move(tasks.front());
14                 tasks.pop();
15             } // Unlock гук
16             task(); // Изпълнение извън lock-a
17         }
18     }
19 };
```

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

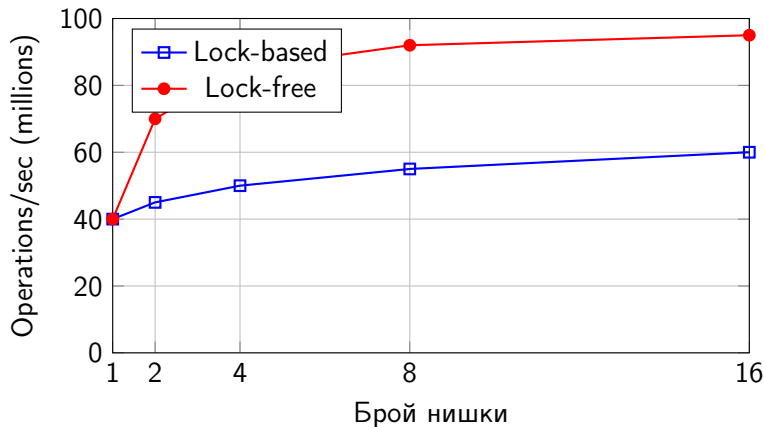
Ключови техники:

- 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

```
1 template<typename T>
2 class LockFreeQueue {
3     struct Node {
4         std::shared_ptr<T> data;
5         std::atomic<Node*> next;
6         Node() : next(nullptr) {}
7     };
8
9     std::atomic<Node*> head;
10    std::atomic<Node*> tail;
11
12 public:
13     void enqueue(T value) {
14         auto new_node = new Node();
15         new_node->data = std::make_shared<T>(std::move(value));
16
17         Node* old_tail = tail.load(std::memory_order_relaxed);
18         while (!tail.compare_exchange_weak(old_tail, new_node,
19             std::memory_order_release, std::memory_order_relaxed)) {
20             // Retry ако друга нишка промени tail
21         }
22         old_tail->next.store(new_node, std::memory_order_release);
23     }
24
25     std::shared_ptr<T> dequeue() {
26         Node* old_head = head.load(std::memory_order_acquire);
27         while (old_head && !head.compare_exchange_weak(old_head,
28             old_head->next.load(), std::memory_order_release)) {
29             // Retry
30         }
```

Lock-Free vs Lock-Based: Performance



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

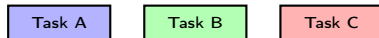
Thread Pool c Lock-Based Queue

```
1 class ThreadPool {
2     std::vector<std::thread> workers;
3     std::queue<std::function<void()>> tasks;
4     std::mutex queue_mutex;
5     std::condition_variable condition;
6     bool stop = false;
7
8 public:
9     ThreadPool(size_t threads) {
10         for (size_t i = 0; i < threads; ++i) {
11             workers.emplace_back([this] {
12                 while (true) {
13                     std::function<void()> task;
14                     {
15                         std::unique_lock<std::mutex> lock(queue_mutex);
16                         condition.wait(lock, [this] {
17                             return stop || !tasks.empty();
18                         });
19                         if (stop && tasks.empty()) return;
20                         task = std::move(tasks.front());
21                         tasks.pop();
22                     }
23                     task(); // Изпълнение извън lock-a
24                 }
25             });
26         }
27     }
28     // enqueue(), destructor...
29 };
```

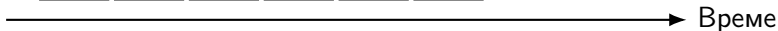
Coroutines: Какво са?

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

Нормални функции:



Coroutines:



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 $\sim 1-10\mu s$
- Паралелно изпълнение
- Използват множество ядра

Coroutines (user-level):

- Управлявани от програмата
- Кооперативен scheduling
- Леки (KB state)
- Context switch $\sim 10-100ns$
- 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

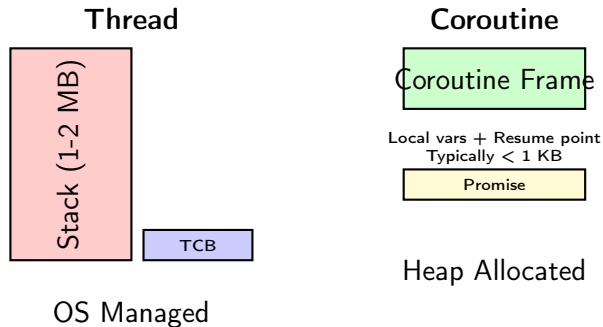
Coroutine:

- Състоянието се съхранява в heap (coroutine frame)
- `co_await` запазва локални променливи и позиция
- Контролът се връща на caller-a
- 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)

Memory Layout: Thread vs Coroutine



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

Предизвикателство: Bus Errors и Stack Overflow

Проблем 1: Неправилно използване на указатели

```
1 Task<int> dangerous_coroutine() {  
2     int local_var = 42;  
3     co_await some_async_operation();  
4     // Ако coroutine се премести в друга нишка,  
5     // stack-allocated променливи могат да станат невалидни!  
6     return local_var; // Потенциален bus error!  
7 }
```

Проблем 2: Lifetime на локални променливи

```
1 Task<void> reference_problem() {  
2     std::string temp = "dangerous";  
3     auto& ref = temp;  
4  
5     co_await switch_thread(); // Преминаване към друга нишка  
6  
7     std::cout << ref; // UNDEFINED BEHAVIOR! temp вече не съществува  
8 }
```

Решение:

- Всички данни в coroutine frame (не на stack)
- Избягвайте references към локални променливи
- Използвайте `std::move` или копиране

Проблем: Coroutines нямат традиционен stack trace

```
1 Task<void> level3() {  
2     co_await std::suspend_always{};  
3     throw std::runtime_error("Error!"); // Къде е stack trace-а?  
4 }  
5  
6 Task<void> level2() {  
7     co_await level3();  
8 }  
9  
10 Task<void> level1() {  
11     co_await level2();  
12 }  
13  
14 // При exception, debugger показва само текущата coroutine,  
15 // не целия "async call stack"
```

Решения:

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

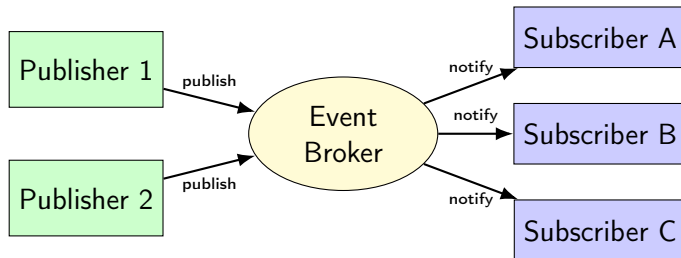
Проста Coroutine имплементация (C++20)

```
1 \begin{lstlisting}[basicstyle=\ttfamily\tiny]
2 #include <coroutine>
3 #include <iostream>
4
5 template<typename T>
6 struct Task {
7     struct promise_type {
8         T value;
9
10         Task get_return_object() {
11             return Task{std::coroutine_handle<promise_type>::from_promise(*this)};
12         }
13         std::suspend_never initial_suspend() { return {}; }
14         std::suspend_always final_suspend() noexcept { return {}; }
15         void unhandled_exception() { std::terminate(); }
16
17         void return_value(T v) { value = v; }
18     };
19
20     std::coroutine_handle<promise_type> handle;
21
22     T get() { return handle.promise().value; }
23     ~Task() { if (handle) handle.destroy(); }
24 };
25
26 Task<int> async_computation() {
27     std::cout << "Starting...\n";
28     co_await std::suspend_always{}; // Suspend тук
29     std::cout << "Resuming...\n";
30     co_return 42;
31 }
```

Реален пример: Async I/O с Coroutines

```
1 Task<std::string> async_read_file(const std::string& filename) {
2     // Simulate async file reading
3     co_await async_open(filename);
4     std::string content;
5
6     while (true) {
7         auto chunk = co_await async_read_chunk();
8         if (chunk.empty()) break;
9         content += chunk;
10    }
11
12    co_await async_close();
13    co_return content;
14 }
15
16 Task<void> process_files() {
17     // Множество паралелни четения (но в една нишка!)
18     auto file1 = async_read_file("data1.txt");
19     auto file2 = async_read_file("data2.txt");
20     auto file3 = async_read_file("data3.txt");
21
22     // Чакаме всички
23     auto content1 = co_await file1;
24     auto content2 = co_await file2;
25     auto content3 = co_await file3;
26
27     // Обработка...
28     std::cout << "Total size: " << content1.size() + content2.size() + content3.size();
29 }
```

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

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 триггерва множество реакции

Предизвикателство 1: Гаранции за доставка

Проблем: Какво ако subscriber не получи съобщението?

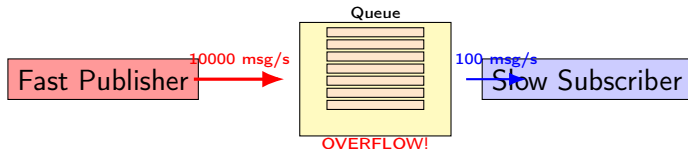
Опции за доставка:

- 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-a)
- Паралелизация на subscribers (нашата тема!)

Предизвикателство 3: Ordering

Проблем: Запазване на реда на съобщенията при паралелна обработка

```
1 // Subscriber  
2 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** – препоредване след обработка
- 4 **Per-key ordering** – само свързани съобщения се подреждат

Проста Pub/Sub имплементация

```
1 template<typename Event>
2 class EventBroker {
3     using Subscriber = std::function<void(const Event&)>;
4     using SubscriberId = size_t;
5
6     std::unordered_map<SubscriberId, Subscriber> subscribers;
7     std::mutex mutex;
8     SubscriberId next_id = 0;
9
10 public:
11     SubscriberId subscribe(Subscriber&& callback) {
12         std::lock_guard<std::mutex> lock(mutex);
13         auto id = next_id++;
14         subscribers[id] = std::move(callback);
15         return id;
16     }
17
18     void unsubscribe(SubscriberId id) {
19         std::lock_guard<std::mutex> lock(mutex);
20         subscribers.erase(id);
21     }
22
23     void publish(const Event& event) {
24         std::lock_guard<std::mutex> lock(mutex);
25         for (auto& [id, subscriber] : subscribers) {
26             subscriber(event); // Синхронно извикване
27         }
28     }
29 };
```

Асинхронна имплементация с Thread Pool

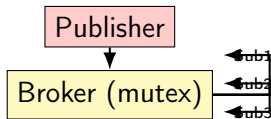
```
1 template<typename Event>
2 class AsyncEventBroker {
3     using Subscriber = std::function<void(const Event&);>;
4     std::vector<Subscriber> subscribers;
5     ThreadPool pool;
6     std::mutex mutex;
7
8 public:
9     AsyncEventBroker(size_t num_threads = std::thread::hardware_concurrency())
10         : pool(num_threads) {}
11
12     void subscribe(Subscriber&& callback) {
13         std::lock_guard<std::mutex> lock(mutex);
14         subscribers.push_back(std::move(callback));
15     }
16
17     void publish(const Event& event) {
18         std::lock_guard<std::mutex> lock(mutex);
19         // Всеки subscriber се обработва в отделна задача
20         for (auto& subscriber : subscribers) {
21             pool.enqueue([subscriber, event]() {
22                 subscriber(event); // Паралелно изпълнение!
23             });
24         }
25     } // publish() не чака обработката
26 };
```

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

Паралелизация на subscriber dispatch

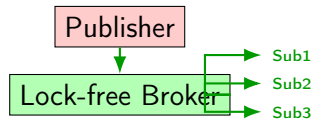
Проблем: Наивната имплементация има serialization bottleneck

Наивен подход



Serialization!

Оптимизиран подход



Паралелно!

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

Заглавие: "Efficient Parallelized Message Dispatching in Publisher/Subscriber Systems"

Основни приноси:

1 Lock-free dispatch algorithm

- CAS-based subscriber registration
- Wait-free publish() имплементация
- RCU за безопасно четене на subscriber list

2 Thread Pool integration

- Work-stealing за балансировка
- Subscriber affinity за cache locality

3 Performance benchmarks

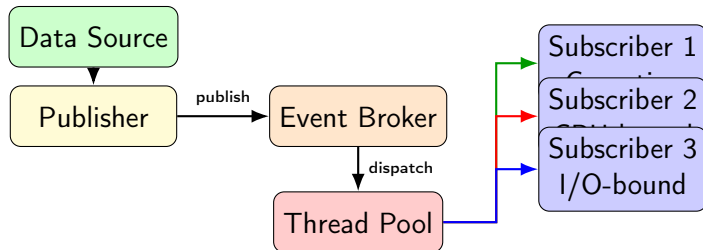
- 10x throughput подобрене при 16+ subscribers
- 50% latency редукция спрямо mutex-based подход
- Scaling до 128 CPU cores

Комбинацията Thread Pool + Lock-free Pub/Sub = High-Performance системи

Code snippet от работата: RCU-based Subscriber List

```
1 template<typename Event>
2 class RCUEventBroker {
3     struct SubscriberNode {
4         std::function<void(const Event&)> callback;
5         std::shared_ptr<SubscriberNode> next;
6     };
7
8     std::atomic<std::shared_ptr<SubscriberNode>> head;
9     ThreadPool pool;
10
11 public:
12     void subscribe(std::function<void(const Event&)> callback) {
13         auto new_node = std::make_shared<SubscriberNode>();
14         new_node->callback = std::move(callback);
15
16         auto old_head = head.load(std::memory_order_acquire);
17         do {
18             new_node->next = old_head;
19         } while (!head.compare_exchange_weak(old_head, new_node,
20             std::memory_order_release, std::memory_order_acquire));
21     }
22
23     void publish(const Event& event) {
24         // Lock-free read
25         auto node = head.load(std::memory_order_acquire);
26         while (node) {
27             pool.enqueue([cb = node->callback, event]() { cb(event); });
28             node = node->next;
29         }
30     }
```

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 conn
Event routing	Pub/Sub	Loose coupling
High-throughput	Lock-free + Pool	Минимизира contention
Mixed workload	Hybrid	Комбинираща силни страни

Практически съвет:

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

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

Модул 2: Комуникация между задачи

- Pub/Sub като message passing механизъм
- Lock-free structures за синхронизация
- Thread pools за task orchestration

Scaling до distributed systems:

- ZeroMQ, RabbitMQ – distributed pub/sub
- Apache Kafka – event streaming
- gRPC, Protobuf – efficient serialization

Допълнителни теми за изследване:

- Memory ordering (acquire/release semantics)
- Hazard pointers за garbage collection
- Transactional memory

Препоръчителна литература

Книги:

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

Papers:

- "Lock-Free Data Structures" – Michael & Scott
- "RCU (Read-Copy-Update)" – McKenney
- "Efficient Parallelized Message Dispatching" – (ваша работа!)

Online ресурси:

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

Въпроси?

Теми за дискусия:

- ABA problem и решенията му
- Memory ordering в practice
- Profiling tools (perf, VTune, Tracy)
- Real-world case studies
- C++23/26 нововъведения (executors, async RAI)

Благодаря за вниманието!

Контакт: alex@example.com | GitHub: github.com/alex