HSE
Faculty of Computer Science

Конспект углубленного курса по C++

AUTHOR
Your Name

COURSE
Advanced C++
Fall 2023

Оглавление

Часть I

Лекция 10 – Advanced thread

Глава 1

Anatomy of a Thread: Cost, Kernel & Scheduling

Многопоточность в C++ — это не просто абстракция `std::thread`. За каждым объектом стандартной библиотеки скрывается сложный механизм операционной системы, имеющий свою цену в тактах процессора и байтах оперативной памяти. Понимание физического устройства потока, алгоритмов планировщика ядра (OS Scheduler) и накладных расходов на переключение контекста является фундаментом для построения высоконагруженных систем.

В этой главе мы деконструируем понятие "поток", разберем стоимость его создания и эксплуатации, а также проанализируем архитектурные ошибки, возникающие при наивном использовании примитивов стандарта.

Физическая структура потока

> **Контекст исполнения (Execution Context)**
>
> Поток (thread) физически представляет собой совокупность состояния регистров процессора и выделенного региона памяти под стек. В многопоточной среде все потоки одного процесса разделяют общее адресное пространство (кучу, сегмент кода, глобальные переменные), но имеют изолированный стек и набор регистров.

С точки зрения ядра Linux, поток — это "легковесный процесс" (LWP — Light Weight Process). Для планировщика нет принципиальной разницы между процессом и потоком, кроме того факта, что потоки делят таблицу страниц памяти (Page Table).

Минимальный набор данных, определяющий поток, включает:

1. **Регистры общего назначения (GPR):** RAX, RBX, RCX и т.д. в архитектуре x86_64.
2. **Указатель инструкций (Instruction Pointer, RIP):** Адрес текущей исполняемой команды.
3. **Указатель стека (Stack Pointer, RSP):** Адрес вершины стека данного потока.
4. **Стек (Stack):** Область памяти для локальных переменных, адресов возврата и аргументов функций.

При переключении контекста ядро ОС обязано сохранить текущее состояние регистров уходящего потока в специальную структуру в Kernel Space и загрузить значения регистров приходящего потока.

Стоимость создания потока

Создание потока — операция дорогая. В отличие от вызова функции, который занимает наносекунды (просто `call` и `push` в стек), создание потока (`std::thread`) требует системного вызова (`clone` в Linux), выделения структур ядра и настройки виртуальной памяти.

Эмпирическая оценка времени создания потока на современном железе составляет **от 5 до 50 микросекунд**. Это на несколько порядков медленнее, чем создание объекта в куче.

Бенчмарк: Thread vs Function

Сравним стоимость выполнения пустой работы через вызов функции и через создание нового потока.

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <chrono>

void dummy_work() {
    // Имитация минимальной работы
    volatile int x = 0;
    x++;
}

int main() {
    const int N = 10000;

    auto start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i) {
        dummy_work();
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "Function calls: "
              << std::chrono::duration_cast<std::chrono::microseconds>(end -
              ↪  start).count()
              << " us\n";

    start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i) {
        std::thread t(dummy_work);
        t.join();
    }
    end = std::chrono::high_resolution_clock::now();
    std::cout << "Thread creation: "
              << std::chrono::duration_cast<std::chrono::microseconds>(end -
              ↪  start).count()
              << " us\n";
}
```

Результаты на типичном сервере показывают, что создание потоков занимает в тысячи раз больше времени. Если ваша задача выполняется быстрее, чем время создания потока (единицы микросекунд), использование `std::thread` напрямую приведет к деградации производительности.

Memory Overhead: Стек и Виртуальная память

По умолчанию в Linux размер стека для главного потока и создаваемых потоков (`pthread_create`) составляет **8 МБ**. Это значение можно проверить командой `ulimit -s`.

> **Важно!**
>
> Поток не потребляет 8 МБ физической памяти (RAM) мгновенно. Стек выделяется в *виртуальном адресном пространстве*. Физические страницы (frames) выделяются ядром лениво (Demand Paging) по мере обращения к адресам стека.

Однако, даже виртуальная память — ресурс исчерпаемый. Рассмотрим задачу C10K (обслуживание 10 000 соединений). Если мы используем модель "один поток на соединение" (Thread-per-connection):

$$10\,000 \text{ threads} \times 8 \text{ MB} \approx 80 \text{ GB Virtual Memory}$$

Для 32-битных систем это гарантированный крах (адресное пространство всего 4 ГБ). Для 64-битных систем это создает огромную нагрузку на TLB (Translation Lookaside Buffer) процессора и структуры ядра, управляющие памятью (VMA — Virtual Memory Areas).

Если программа под нагрузкой реально использует стек (глубокая рекурсия или большие буферы на стеке), то потребление физической памяти также станет колоссальным, что приведет к OOM (Out Of Memory) Killer.

Планировщик Linux (CFS)

В современных ОС используется **вытесняющая многозадачность** (preemptive multitasking). Разработчик не контролирует, когда поток начнет исполняться и когда будет прерван. Этим занимается планировщик ядра.

В Linux используется **CFS (Completely Fair Scheduler)**.

Механика работы CFS

CFS моделирует "идеальный многозадачный процессор" на реальном железе.

- Каждому потоку присваивается `vruntime` (virtual runtime) — время, которое поток уже провел на процессоре, взвешенное на его приоритет (nice value).
- Потоки организованы в **красно-черное дерево** (Red-Black Tree), отсортированное по `vruntime`.
- Планировщик всегда выбирает поток с наименьшим `vruntime` (самый левый узел дерева).

Квант времени и Переключение контекста

Минимальный интервал перепланирования (latency target) в Linux может составлять, например, 6-24 мс, но реальные кванты времени динамические.

Переключение контекста (Context Switch) — это не только сохранение регистров. Это:

1. Смена режима процессора (User mode → Kernel mode).
2. Загрязнение кэша инструкций и данных (Cache Pollution). Когда новый поток начинает исполнение, нужные ему данные, скорее всего, отсутствуют в L1/L2 кэшах.
3. Накладные расходы на обновление структур планировщика.

На заметку

Типичный сервер может иметь 256 логических ядер (например, AMD EPYC). Балансировка нагрузки и миграция потоков между NUMA-нодами в таких условиях — сложнейшая задача. Миграция потока на другое ядро аннулирует его прогретый кэш.

Антипаттерн: Thread per Request

Исторически веб-серверы писались по модели: "Пришел запрос → Создали поток → Обработали → Уничтожили поток".

Почему это плохо в современном HighLoad: 1. **Unbounded Concurrency:** При всплеске трафика (DDoS или хабраэффект) создается бесконечное число потоков. 2. **Thrashing:** Система тратит больше времени на переключение контекста между тысячами потоков, чем на полезную работу. 3. **Latency Tail:** Время ответа начинает непредсказуемо расти из-за очередей в планировщике.

> **Thread Pool (Пул потоков)**
>
> Архитектурный паттерн, при котором создается фиксированное число долгоживущих потоков (обычно равное числу аппаратных ядер), разбирающих задачи из общей очереди.

Это унифицирует место создания потоков и ограничивает их максимальное число, предотвращая перегрузку системы.

Проблемы стандартных абстракций C++

Стандарт C++ предлагает высокоуровневые инструменты, которые часто вводят в заблуждение своей простотой.

std::execution::par (C++17)

Алгоритмы с политиками исполнения, например `std::for_each(std::execution::par, ...)`, обещают автоматическое распараллеливание.

```
std::vector<int> data = ...;
std::for_each(std::execution::par, data.begin(), data.end(), [](int& x) {
    heavy_computation(x);
});
```

Проблема: Стандарт не специфицирует, как именно реализуется параллелизм.

- Реализация может создать новый поток на каждый вызов алгоритма.

- Если вы вызовете такой алгоритм внутри уже запущенного параллельного кода, произойдет комбинаторный взрыв числа потоков (Oversubscription).
- Отсутствует контроль над тем, в каком пуле исполняются задачи.

В продакшен-коде использование `std::execution::par` без глубокого понимания реализации стандартной библиотеки конкретного компилятора (libstdc++, libc++) считается небезопасным.

std::async и std::future

`std::async` может запускать задачу синхронно (в том же потоке) или асинхронно, в зависимости от флагов и реализации. Возвращаемый `std::future` в деструкторе блокирует поток до завершения задачи, что часто приводит к неожиданным последовательным исполнениям там, где ожидалась параллельность.

Резюме раздела

- Создание потока — дорогая операция (>5 мкс). Избегайте частого создания/уничтожения потоков.
- Каждый поток резервирует 8 МБ виртуального адресного пространства. 100k потоков недопустимы.
- Переключение контекста стоит дорого из-за cache misses.
- Используйте Thread Pool для контроля конкурентности.
- Осторожно с `std::async` и `std::execution::par` — они скрывают детали управления потоками, что опасно для HighLoad.

Глава 2

Building a Production-Grade ThreadPool

Осознав стоимость создания потоков и ограничения операционной системы в предыдущей главе, мы приходим к необходимости архитектурного паттерна **ThreadPool**. Идея проста: мы создаем фиксированный набор потоков (worker threads) на старте приложения и переиспользуем их для выполнения множества задач.

Однако реализация надежного, готового к продакшену пула потоков — задача нетривиальная. Она требует глубокого понимания примитивов синхронизации, управления жизненным циклом потоков и обработки исключительных ситуаций. В этой главе мы шаг за шагом спроектируем и реализуем ThreadPool на современном C++.

Фундамент: Блокирующая очередь (Blocking Queue)

Сердцем любого пула потоков является очередь задач. Воркеры (потребители) должны извлекать задачи из очереди, а клиенты (производители) — добавлять их.

Ключевое требование к такой очереди — она должна быть **блокирующей** для потребителей. Если задач нет, поток-воркер не должен крутиться в цикле (`while(empty) `), потребляя 100% CPU (busy wait). Он должен "уснуть" на уровне ядра ОС, освободив ресурсы процессора, и проснуться только тогда, когда появится новая задача или поступит сигнал остановки.

Для реализации этого механизма нам потребуются три компонента:

1. `std::deque<T>` или `std::queue<T>` — контейнер для хранения данных.
2. `std::mutex` — для обеспечения взаимного исключения (mutual exclusion) при доступе к контейнеру.
3. `std::condition_variable` — для организации ожидания и уведомления потоков.

Реализация UnboundedBlockingQueue

Рассмотрим реализацию неограниченной (unbounded) очереди. "Неограниченная" означает, что метод `Push` никогда не блокируется из-за переполнения (пока есть оперативная память).

```
#include <mutex>
#include <condition_variable>
```

```cpp
#include <deque>
#include <optional>

template <typename T>
class UnboundedBlockingQueue {
public:
    void Push(T item) {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            buffer_.push_back(std::move(item));
        }
        // Уведомляем один спящий поток (если есть)
        not_empty_.notify_one();
    }

    std::optional<T> Pop() {
        std::unique_lock<std::mutex> lock(mutex_);

        // Ожидаем, пока очередь не станет непустой ИЛИ не будет закрыта
        not_empty_.wait(lock, [this]() {
            return !buffer_.empty() || is_closed_;
        });

        if (buffer_.empty() && is_closed_) {
            return std::nullopt; // Очередь закрыта и пуста
        }

        T item = std::move(buffer_.front());
        buffer_.pop_front();
        return item;
    }

    void Close() {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            is_closed_ = true;
        }
        // Будим ВСЕ потоки, чтобы они увидели флаг is_closed_
        not_empty_.notify_all();
    }

private:
    std::deque<T> buffer_;
    std::mutex mutex_;
    std::condition_variable not_empty_;
    bool is_closed_ = false;
};
```

Разбор механики синхронизации

1. Lock Guard vs Unique Lock

В методе `Push` используется `std::lock_guard`, так как нам нужно просто захватить мьютекс на время скоупа. В методе `Pop` используется `std::unique_lock`. Это критически важ-

но, так как `std::condition_variable::wait` требует именно `unique_lock`. Внутри `wait` атомарно происходит следующее: 1. Мьютекс разблокируется. 2. Поток переводится в режим ожидания (sleep). При пробуждении мьютекс снова захватывается.

2. Spurious Wakeups (Ложные пробуждения)

Обратите внимание на конструкцию вызова `wait`:

```
not_empty_.wait(lock, [this]() {
    return !buffer_.empty() || is_closed_;
});
```

Это эквивалентно циклу:

```
while (!(!buffer_.empty() || is_closed_)) {
    not_empty_.wait(lock);
}
```

> **Важно!**
>
> Никогда не используйте `if` с `wait`. POSIX Threads (и стандарт C++) допускают **Spurious Wakeups** — ситуация, когда поток просыпается без вызова `notify`. Если не проверить условие (предикат) повторно в цикле, поток может попытаться извлечь элемент из пустой очереди, что приведет к UB или исключению.

3. Graceful Shutdown (Закрытие очереди)

Метод `Close` устанавливает флаг `is_closed_` и вызывает `notify_all()`. Это необходимо для корректного завершения работы. Представим ситуацию: 5 воркеров спят в методе `Pop`, ожидая задач. Если мы просто перестанем посылать задачи, воркеры никогда не проснутся и программа зависнет при выходе. Вызов `notify_all()` будит их всех, они проверяют предикат `is_closed_`, видят `true` и корректно выходят, возвращая `std::nullopt`.

Архитектура ThreadPool

Теперь, имея надежную очередь, мы можем реализовать сам пул.

Интерфейс задач

Для простоты будем считать, что задача — это `std::function<void()>`. Это стирает тип функтора (лямбда, указатель на функцию, объект с `operator()`), позволяя хранить разнородные задачи в одной очереди.

> **На заметку**
>
> В реальных высокопроизводительных системах `std::function` может быть заменена на кастомную реализацию delegate без аллокаций памяти (SBO — Small Buffer Optimization), так как `std::function` может аллоцировать память в куче.

Полная реализация ThreadPool

```cpp
#include <vector>
#include <thread>
#include <functional>
#include <iostream>

// Используем UnboundedBlockingQueue, описанную выше
using Task = std::function<void()>;

class ThreadPool {
public:
    explicit ThreadPool(size_t num_threads) {
        Start(num_threads);
    }

    ~ThreadPool() {
        Stop();
    }

    // Запрещаем копирование и перемещение для простоты
    ThreadPool(const ThreadPool&) = delete;
    ThreadPool& operator=(const ThreadPool&) = delete;

    void Submit(Task task) {
        queue_.Push(std::move(task));
    }

private:
    void Start(size_t num_threads) {
        for (size_t i = 0; i < num_threads; ++i) {
            workers_.emplace_back([this]() {
                WorkerRoutine();
            });
        }
    }

    void Stop() {
        // 1. Сообщаем очереди, что больше задач не будет
        // и будим всех спящих воркеров.
        queue_.Close();

        // 2. Ждем завершения каждого потока.
        for (auto& worker : workers_) {
            if (worker.joinable()) {
                worker.join();
            }
        }
    }

    void WorkerRoutine() {
        while (true) {
            // Блокируемся в ожидании задачи
            auto task_opt = queue_.Pop();
```

```
53
54              // Если вернулся nullopt, значит очередь закрыта и пуста -> выход
55              if (!task_opt.has_value()) {
56                  break;
57              }
58
59              auto& task = task_opt.value();
60              try {
61                  task(); // Выполнение задачи
62              } catch (const std::exception& e) {
63                  // Критически важно ловить исключения!
64                  // Иначе исключение покинет поток и вызовет std::terminate
65                  std::cerr << "Worker exception: " << e.what() << std::endl;
66              } catch (...) {
67                  std::cerr << "Worker unknown exception" << std::endl;
68              }
69          }
70      }
71
72  private:
73      std::vector<std::thread> workers_;
74      UnboundedBlockingQueue<Task> queue_;
75  };
```

Анализ Corner Cases и Ошибок

1. Исключения в воркерах

В методе `WorkerRoutine` блок `try-catch` обязателен. Стандарт C++ гласит: если исключение покидает функцию верхнего уровня потока (в данном случае лямбду, переданную в конструктор `std::thread`), вызывается `std::terminate()`, который аварийно завершает **весь процесс**, а не только один поток.

Без `try-catch`, одна сбойная задача ("poison pill") убила бы всё приложение.

2. Порядок остановки (Destruction Order)

В деструкторе (или методе `Stop`) порядок действий критичен:

1. Сначала **Queue Close**. Это устанавливает флаг и будит потоки.
2. Только потом **Thread Join**.

Если поменять местами или забыть `Queue Close`, вызов `worker.join()` заблокируется навечно (deadlock), так как главный поток будет ждать завершения воркера, а воркер будет спать внутри `queue.Pop()`, ожидая задач, которые никогда не придут.

3. Невозможность принудительного убийства (Thread Cancellation)

Новички часто спрашивают: "Как убить задачу, которая зависла?". В стандартном C++ (да и в большинстве ОС) безопасного способа принудительно остановить поток извне **не существует**.

- `pthread_cancel` или `TerminateThread` (WinAPI) существуют, но их использование опасно.
- Если убить поток, пока он держит `std::mutex`, этот мьютекс останется заблокированным навсегда (abandoned mutex). Любой другой поток, попытавшийся его захватить, зависнет.
- Также не будут вызваны деструкторы стековых объектов (RAII не сработает), что приведет к утечкам ресурсов.

Единственный способ прервать задачу — кооперативная отмена, когда сама задача периодически проверяет флаг `std::atomic<bool> should_stop`.

4. Data Races при проверке состояния

Рассмотрим распространенную ошибку реализации очереди без мьютексов при проверке на пустоту:

```
// ОШИБКА!
bool IsEmpty() const {
    return buffer_.empty(); // Чтение без мьютекса!
}
```

`std::deque::empty()` не является атомарной операцией. Чтение состояния контейнера параллельно с его модификацией (в методе `Push` из другого потока) — это классическая **Data Race**, приводящая к Undefined Behavior. Любой доступ к разделяемым данным должен быть защищен тем же мьютексом, что и операции записи.

Резюме раздела

Мы построили корректный ThreadPool, который:

- Использует фиксированное число потоков.
- Эффективно ожидает задачи без busy-wait (благодаря `condition_variable`).
- Корректно обрабатывает завершение работы (Graceful Shutdown).
- Устойчив к исключениям внутри пользовательских задач.

Этот код является минимальным базисом. В реальных системах (например, в игровых движках или веб-серверах) его усложняют, добавляя приоритеты задач, work-stealing (кражу задач между очередями разных потоков) и локальные очереди для уменьшения конкуренции за единый мьютекс.

Глава 3

User-Space Concurrency: Implementing Fibers

В предыдущих главах мы работали с потоками операционной системы (`std::thread`). Мы выяснили, что это тяжеловесные объекты: переключение контекста требует вмешательства ядра (syscall), загрязняет кэш и занимает микросекунды.

Для систем, требующих обработки сотен тысяч одновременных соединений (C100K+), модель "один поток на соединение" становится узким местом. Решением является **User-Space Concurrency** — реализация многозадачности силами самого приложения, без участия ядра ОС в планировании конкретных задач.

В этой главе мы реализуем библиотеку **Файберов** (Fibers, также известны как Green Threads или Coroutines) с нуля. Мы опустимся на самый низкий уровень — уровень ассемблера и регистров процессора.

Кооперативная многозадачность

В отличие от вытесняющей (preemptive) многозадачности, где ОС насильно прерывает потоки по таймеру, файберы используют **кооперативную** (cooperative) модель.

> **Файбер (Fiber)**
>
> Легковесный поток исполнения, управляемый планировщиком в пространстве пользователя (User Space). Переключение между файберами происходит только тогда, когда сам файбер явно отдает управление (вызывает `Yield`).

Модель M:N Threading: Мы запускаем M файберов поверх N физических потоков ОС (обычно N равно числу ядер). Если файбер блокируется (ждет I/O или мьютекс), физический поток не блокируется, а переключается на выполнение другого файбера.

Архитектура Файбера

Чтобы превратить функцию в "поток", который можно остановить и продолжить, нам нужно сохранить его состояние. Что определяет состояние исполнения?

1. **Стек (Stack):** Локальные переменные и цепочка вызовов. У файбера должен быть свой собственный стек, выделенный в куче.

2. **Регистры процессора (Context):** Значения переменных, находящихся в данный момент на "верхушке" вычислений.
3. **Указатель инструкций (RIP):** Место в коде, где мы остановились.

Инфраструктура: Стек

Стек — это просто непрерывный блок памяти. Стек растет "вниз" (от старших адресов к младшим) на архитектуре x86_64.

```
class Stack {
public:
    explicit Stack(size_t size)
        : stack_{std::make_unique<char[]>(size)}
        , top_{stack_.get() + size} // Указатель на КОНЕЦ массива (старший адрес)
    {
        // Выравнивание стека по границе 16 байт (требование ABI)
        top_ = reinterpret_cast<char*>(
            reinterpret_cast<uintptr_t>(top_) & ~0xF
        );
    }

    void* Top() const { return top_; }
private:
    std::unique_ptr<char[]> stack_;
    char* top_;
};
```

Deep Dive: Assembly & Calling Conventions

Самая сложная часть реализации — переключение контекста. C++ не имеет стандартных средств для прямой записи в регистры `RSP` (Stack Pointer) и `RIP` (Instruction Pointer). Нам придется использовать ассемблер.

Согласно **System V AMD64 ABI** (соглашение о вызовах в Linux/Unix), регистры делятся на две группы:

- **Caller-saved (Volatile):** `rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`-`r11`. Функция может менять их как угодно. Если вызывающему коду они нужны, он сам их сохраняет *перед* вызовом.
- **Callee-saved (Non-volatile):** `rbx`, `rbp`, `r12`-`r15`. Вызываемая функция **обязана** сохранить их значения и восстановить перед возвратом.

> **Важно!**
>
> Для реализации переключения контекста нам достаточно сохранять только **Callee-saved** регистры. Почему? Потому что переключение контекста выглядит как вызов функции `SaveContext`. Компилятор C++, генерирующий код вызова этой функции, *уже* сохранил все нужные ему Volatile-регистры на стеке (или они ему не нужны). Наша задача — сохранить только те регистры, которые компилятор ожидает увидеть неизменными после возврата из функции.

Структура контекста:

```cpp
struct Context {
    void* rip; // Instruction Pointer (адрес возврата)
    void* rsp; // Stack Pointer

    // Callee-saved регистры
    void* rbp; // Base Pointer
    void* rbx;
    void* r12;
    void* r13;
    void* r14;
    void* r15;
};
```

Магия переключения: SaveContext и JumpContext

Мы реализуем две функции на ассемблере.

1. SaveContext

Сохраняет текущее состояние в структуру `Context` и возвращает `0`. Сигнатура: `extern "C" int SaveContext(Context* ctx);`

> **На заметку**
>
> В System V ABI первый аргумент функции (в данном случае указатель `ctx`) передается в регистре `RDI`. Возвращаемое значение — в `RAX`.

```
.global SaveContext
SaveContext:
    // RDI содержитуказательна   Context

    // 1. Сохраняемадресвозврата   (RIP).
    // Текущий RIP лежитнаверхушкестекаположила    ( инструкция call).
    mov rax, [rsp]
    mov [rdi + 0], rax  // ctx->rip = [rsp]

    // 2. Сохраняемуказательстека   (RSP).
    // Намнужен  RSP довызова  call, поэтому +8 байтпропускаем ( адресвозврата )
    lea rax, [rsp + 8]
    mov [rdi + 8], rax  // ctx->rsp = rsp + 8

    // 3. Сохраняемостальныерегистры
    mov [rdi + 16], rbp
    mov [rdi + 24], rbx
    mov [rdi + 32], r12
    mov [rdi + 40], r13
    mov [rdi + 48], r14
    mov [rdi + 56], r15

    // 4. Возвращаем 0 (ESaveContextResult::Saved)
    xor rax, rax
    ret
```

2. JumpContext

Восстанавливает состояние из структуры и передает управление. Эта функция **не возвращает управление** туда, откуда её вызвали. Она "возвращается" туда, где был сохранен контекст. Сигнатура: `extern "C" void JumpContext(Context* ctx);`

```
.global JumpContext
JumpContext:
    // RDI содержитуказательна   Context

    // 1. Восстанавливаем Callee-saved регистры
    mov rbp, [rdi + 16]
    mov rbx, [rdi + 24]
    mov r12, [rdi + 32]
    mov r13, [rdi + 40]
    mov r14, [rdi + 48]
    mov r15, [rdi + 56]

    // 2. ВосстанавливаемСтек ! Самый( важныймомент )
    mov rsp, [rdi + 8]

    // 3. Подготавливаем "возврат".
    // Мыхотим , чтобыпотокпродолжилвыполнениетак    , будто
    // SaveContext толькочтовернулуправление   , носдругимрезультатом   .
    // Загружаемсохраненный  RIP встек , чтобыинструкция  ret егозабрала .

    mov rax, [rdi + 0] // Загружаемцелевой  RIP
    push rax           // Кладемегонастек

    // 4. Устанавливаемвозвращаемоезначение   1 (ESaveContextResult::Resumed)
    mov rax, 1

    // 5. Прыжок!
    ret // Снимет RIP состекаипрыгнеттуда
```

> **Важно!**
>
> Атрибут `__attribute__((returns_twice))` для `SaveContext` критически важен. Он сообщает компилятору, что функция может вернуть управление дважды (как `setjmp`), запрещая агрессивные оптимизации (например, хранение переменных только в регистрах без сброса на стек перед вызовом).

Трамплин и Запуск файбера

Мы научились переключаться. Но как запустить новый файбер в первый раз? У него нет сохраненного контекста.

Мы должны *сфабриковать* контекст вручную.

1. Выделяем стек.
2. Устанавливаем `context.rsp` на вершину этого стека.
3. Устанавливаем `context.rip` на адрес функции-трамплина.

Зачем нужен трамплин? Мы не можем просто указать `rip` на пользовательскую функцию. Когда пользовательская функция завершится (`ret`), процессору нужно знать, куда вернуться. В пустом стеке нет адреса возврата → Segfault. Трамплин — это обертка, которая вызывает пользовательский код, а после его завершения вызывает `Scheduler::Terminate()`, корректно завершая жизнь файбера.

```
// static
void Fiber::Trampoline() {
    // Получаем текущий файбер (установленный шедулером перед прыжком)
    Fiber* fiber = GetCurrentFiber();
    fiber->RunUserRoutine(); // Вызов пользовательской лямбды

    // Если мы здесь, значит задача выполнена.
    // Файбер нельзя просто удалить, нужно уйти в шедулер.
    GetCurrentScheduler()->Terminate();
}

Fiber::Fiber(std::function<void()> routine)
    : stack_(kStackSize), routine_(std::move(routine)) {

    context_.rsp = stack_.Top();
    context_.rip = reinterpret_cast<void*>(Trampoline);
}
```

Планировщик (Scheduler)

Планировщик управляет очередью готовых файберов (`Runnable`) и распределяет процессорное время. Поскольку мы пишем однопоточный планировщик (для простоты), он работает в главном потоке программы.

Intrusive List

Чтобы избежать аллокаций памяти при каждом добавлении файбера в очередь планировщика (что убило бы производительность), мы используем **Интрузивный список**. Файбер сам является узлом списка:

```
class Fiber : public IntrusiveListItem<Fiber> { ... };
```

Это позволяет перемещать файбер между очередями (Ready Queue ↔ Wait Queue мьютекса) простым переписыванием указателей `prev/next` внутри объекта файбера, без `new/delete`.

Цикл планирования

```
void Scheduler::Run() {
    while (!queue_.Empty()) {
        Fiber* fiber = queue_.PopFront();
        current_fiber_ = fiber;

```

```
6           // Магия переключения:
7           // 1. Сохраняем контекст планировщика в main_context_
8           // 2. Загружаем контекст файбера
9           Switch(&main_context_, fiber->GetContext());
10
11          // Сюда мы попадаем, когда файбер вернул управление (Yield/Suspend)
12
13          if (fiber->GetState() == EFiberState::Finished) {
14              delete fiber;
15          } else if (fiber->GetState() == EFiberState::Runnable) {
16              queue_.PushBack(fiber);
17          }
18          // Если Suspended, то ничего не делаем (он хранится в wait queue
            ↪  примитива)
19      }
20  }
```

Примитивы синхронизации: Yield и Suspend

В мире файберов нельзя использовать `std::mutex`, так как он заблокирует весь физический поток, остановив все файберы планировщика. Нам нужны свои примитивы.

Yield (Добровольная уступка)

Файбер говорит: "Я еще не закончил, но готов уступить место другим".

1. Состояние файбера остается `Runnable`.
2. Переключаемся в контекст шедулера.
3. Шедулер кладет файбер в конец очереди.

Suspend (Блокировка)

Используется в мьютексах и condition variables.

1. Файбер меняет состояние на `Suspended`.
2. Переключаемся в контекст шедулера.
3. **Важно:** Шедулер НЕ кладет файбер обратно в очередь `RunQueue`. Файбер "исчезает" для планировщика.
4. Ответственность за возвращение файбера к жизни берет на себя тот, кто его заблокировал (например, `Mutex::Unlock` вызовет `Schedule(fiber)`).

Резюме раздела

Мы реализовали файберы — механизм User-Space многозадачности.

- Мы контролируем стек и регистры вручную.
- Переключение контекста стоит десятки наносекунд (просто `mov` регистров), в отличие от микросекунд у потоков ОС.
- Мы используем только Callee-saved регистры, полагаясь на System V ABI.
- Кооперативная многозадачность требует явного `Yield` или ожидания на примитивах синхронизации, чтобы система была отзывчивой.

Эта технология лежит в основе Goroutines в Go, Coroutines в C++20 и Fiber API в Windows.

Глава 4

Linux Low-Level Sync: The Futex

До сих пор мы рассматривали два полюса синхронизации:

1. **Спинлоки (Spinlocks):** Работают полностью в User Space, используя атомарные инструкции. Они феноменально быстры при отсутствии конкуренции, но при ожидании "сжигают" процессорное время впустую, нагревая воздух.
2. **Тяжелые мьютексы (Kernel Mutexes):** Старые реализации (например, в ранних версиях Linux) всегда обращались к ядру ОС для блокировки. Это экономит CPU (поток спит), но системный вызов (`syscall`) стоит сотни наносекунд, даже если конкуренции нет.

Истина, как всегда, посередине. Большую часть времени мьютекс свободен, и захватывать его нужно быстро (как спинлок). Но если он занят, поток должен честно уснуть (как системный мьютекс).

Именно эту гибридную модель реализует **Futex** (Fast Userspace Mutex) — фундаментальный примитив синхронизации в Linux, на котором построены `std::mutex`, `pthread_mutex` и механизмы синхронизации в JVM и Go runtime.

Философия Futex

Основная идея Futex звучит так: "**Не тревожь ядро, если нет конкуренции**".

> **Futex (Fast Userspace Mutex)**
>
> Механизм, позволяющий потокам синхронизироваться через общую область памяти в пространстве пользователя, прибегая к системным вызовам только в случае необходимости ожидания (блокировки) или пробуждения других потоков.

Технически, futex состоит из двух компонентов:

1. **Aligned Integer в User Space:** Обычная 32-битная переменная (обычно `std::atomic<int32_t>`), хранящая состояние блокировки.
2. **Очередь ожидания в Kernel Space:** Структура внутри ядра, где хранятся спящие потоки, привязанные к физическому адресу этой переменной.

Системный вызов futex()

В Linux нет отдельных сисколлов `futex_wait` или `futex_wake`. Есть один мультиплексированный вызов `sys_futex`. Стандартная библиотека C++ не предоставляет прямого доступа к нему, поэтому для реализации собственных примитивов приходится использовать функцию `syscall`.

```
#include <linux/futex.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <atomic>

// Обертка над системным вызовом
long sys_futex(void* addr1, int op, int val1,
               const struct timespec* timeout, void* addr2, int val3) {
    return syscall(SYS_futex, addr1, op, val1, timeout, addr2, val3);
}
```

Рассмотрим две главные операции.

FUTEX_WAIT

Сигнатура: `futex(addr, FUTEX_WAIT, expected_val, ...)`

Семантика: "Проверь атомарно, что по адресу `addr` все еще лежит значение `expected_val`. Если это так — усыпи текущий поток. Если значение изменилось — верни управление немедленно (код ошибки `EAGAIN`)."

Эта атомарная проверка критически важна. Рассмотрим сценарий гонки без нее (Lost Wakeup Problem):

1. Поток А видит, что мьютекс занят (`val == 1`).
2. Поток А решает уснуть.
3. *В этот момент происходит переключение контекста.*
4. Поток Б освобождает мьютекс (`val = 0`) и вызывает `WAKE`. Но никто еще не спит, уведомление уходит в пустоту.
5. *Управление возвращается к А.*
6. Поток А вызывает "безусловный sleep" и засыпает навсегда, хотя мьютекс уже свободен.

Futex решает это, выполняя проверку значения и засыпание как единую транзакцию внутри ядра (под спинлоком планировщика).

FUTEX_WAKE

Сигнатура: `futex(addr, FUTEX_WAKE, count, ...)`

Семантика: "Разбуди `count` потоков, ожидающих по этому адресу". Обычно используют:

- `count = 1`: аналог `notify_one` (для мьютексов).
- `count = INT_MAX`: аналог `notify_all` (для cond_var или `barrier`).

Реализация Mutex на базе Futex

Напишем простейший мьютекс. Мы будем использовать три состояния переменной `state`:

- **0:** Разблокировано (Unlocked).
- **1:** Заблокировано, нет ожидающих (Locked, no waiters).
- **2:** Заблокировано, есть ожидающие (Locked, with waiters).

Состояние "2" необходимо, чтобы оптимизировать `Unlock`. Если при разблокировке мы видим "1", значит, будить никого не надо, и можно не делать дорогой сисколл `FUTEX_WAKE`.

```
class FutexMutex {
public:
    void lock() {
        int c;
        // 1. Fast Path: Попытка атомарно сменить 0 -> 1.
        // Если успешно, мы захватили мьютекс без системных вызовов.
        if ((c = cmpxchg(0, 1)) != 0) {

            // 2. Slow Path: Мьютекс занят.
            // Если состояние было 1, меняем его на 2 (сигнализируя о наличии ждущего).
            if (c != 2) {
                c = xchg(2);
            }

            // Крутимся в цикле, пока не захватим
            while (c != 0) {
                // Пытаемся уснуть.
                // Ядро усыпит нас ТОЛЬКО если state == 2.
                // Если state изменился (кто-то сделал unlock), мы не уснем.
                sys_futex(&state, FUTEX_WAIT, 2, nullptr, nullptr, 0);

                // Проснулись (или не спали). Пробуем захватить 2 -> 2
                // (фактически просто check, но нам нужно атомарно убедиться)
                // В реальной реализации тут стоит повторить попытку 0 -> 2
                c = xchg(2);
            }
        }
    }

    void unlock() {
        // 1. Fast Path: Атомарно меняем любое значение на 0.
        // fetch_sub(1) было бы оптимизацией, чтобы отличить 1 от 2,
        // но здесь для простоты используем exchange.
        if (state.exchange(0) == 2) {
            // 2. Slow Path: Будим одного ждущего, только если старое значение было 2.
            sys_futex(&state, FUTEX_WAKE, 1, nullptr, nullptr, 0);
        }
    }

private:
    // Вспомогательные обертки над std::atomic
```

```
    int cmpxchg(int expected, int desired) {
        int expected_copy = expected;
        state.compare_exchange_strong(expected_copy, desired);
        return expected_copy;
    }

    int xchg(int desired) {
        return state.exchange(desired);
    }

    std::atomic<int> state{0}; // Требует выравнивания (обычно 4 байта)
};
```

Внутри ядра: Kernel Wait Queues

Что происходит, когда вы зовете `FUTEX_WAIT`?

1. **Hash Table Lookup:** Ядро не может просто взять виртуальный адрес `&state`, так как у разных процессов разные адресные пространства (а futex может быть shared между процессами). Ядро транслирует виртуальный адрес в физический (или в уникальный ключ inode+offset для memory-mapped файлов). 2. По этому ключу вычисляется хэш, и выбирается соответствующая цепочка (bucket) в глобальной хэш-таблице `futex_queues`. 3. **Spinlock:** Блокируется бакет хэш-таблицы. 4. **Atomic Check:** Ядро читает значение из user-space памяти. Если оно не совпадает с `expected`, блокировка снимается, и функция возвращает ошибку. 5. **Sleep:** Если значения совпадают, создается объект ожидания, добавляется в очередь, и поток переводится в состояние `TASK_INTERRUPTIBLE`. Спинлок отпускается, вызывается планировщик (`schedule()`).

Этот механизм гарантирует, что futex является самым эффективным способом блокировки в Linux, сочетая скорость атомиков (в 99% случаев) и корректность системного ожидания при высокой нагрузке.

На заметку

Понимание futex необходимо для следующей главы, так как многие Lock-Free алгоритмы используют его как "fallback mechanism" (механизм отката). Нельзя бесконечно крутиться в CAS-цикле, если ожидание затягивается; рано или поздно поток нужно усыпить.

Глава 5

Lock-Free Programming: Atomic Foundations

В предыдущих главах мы использовали блокировки (мьютексы, футексы), чтобы защитить общие данные. Это подход **пессимистичной** синхронизации: "Я предполагаю, что кто-то помешает мне, поэтому я закрою дверь на замок, прежде чем что-то делать".

Lock-Free программирование — это подход **оптимистичной** синхронизации. Потоки не блокируют друг друга. Вместо этого они пытаются выполнить операцию, и если обнаруживают конфликт (кто-то другой изменил данные быстрее), они повторяют попытку.

Определения и Гарантии

Часто термин "Lock-Free" используют неправильно, называя так любой код без `std::mutex`. Формальное определение основано на гарантиях прогресса системы.

> **Lock-Free (Свобода от блокировок)**
>
> Алгоритм называется Lock-Free, если гарантируется, что хотя бы один поток в системе будет продвигаться вперед (делать полезную работу) за конечное число шагов, даже если другие потоки замедлились или остановились.

> **Wait-Free (Свобода от ожидания)**
>
> Более сильная гарантия. Каждый поток гарантированно завершает свою операцию за конечное число шагов, независимо от действий других потоков.

> **Важно!**
>
> Lock-Free не означает "быстрее". Lock-Free алгоритмы часто потребляют больше CPU из-за циклов повторных попыток (CAS-loops) и могут быть медленнее качественных мьютексов при высокой конкуренции (contention). Их главная цель — устойчивость к остановке потоков (например, если поток убит или прерван планировщиком, он не держит блокировку, мешая другим).

Атомики в C++

Стандарт C++11 ввел модель памяти и тип `std::atomic<T>`.

Операции над атомиками неделимы. Невозможно увидеть атомарную переменную в "частично измененном" состоянии. Основные операции:

- `load()`: Атомарное чтение.
- `store()`: Атомарная запись.
- `exchange(val)`: Записывает новое значение и возвращает старое (RMW — Read-Modify-Write).
- `fetch_add(val)`: Прибавляет значение и возвращает старое.

Compare-And-Swap (CAS)

Фундаментом всех Lock-Free структур данных является операция **Compare-And-Swap**. В C++ она представлена двумя методами: `compare_exchange_weak` и `compare_exchange_strong`.

Сигнатура:

```
bool compare_exchange_strong(T& expected, T desired);
```

Логика (псевдокод, выполняемый атомарно):

```
if (*this == expected) {
    *this = desired;
    return true;
} else {
    expected = *this; // Обновляем expected актуальным значением!
    return false;
}
```

Паттерн CAS-Loop

Рассмотрим простейшую задачу: атомарное изменение переменной по произвольной формуле $X_{new} = f(X_{old})$. Почему нельзя просто написать `atomic = f(atomic.load())`? Потому что между чтением (`load`) и записью (`store`) может вклиниться другой поток и изменить значение. Наш `store` перезатрет его результат. Это классическая гонка (Lost Update).

Правильный подход — использование цикла CAS:

```
std::atomic<int> value{0};

void atomic_update(int operand) {
    int old_val = value.load(); // 1. Снэпшот состояния
    int new_val;
    do {
        // 2. Вычисление нового значения на основе локальной копии
        new_val = old_val + operand; // Здесь может быть любая функция f(old_val)

        // 3. Попытка публикации
        // Если value все еще равно old_val, то записать new_val.
```

```
        // Если нет (кто-то успел изменить value), то обновить old_val и вернуть
        ↪ false.
    } while (!value.compare_exchange_weak(old_val, new_val));
}
```

Strong vs Weak CAS

- `compare_exchange_strong`: Гарантирует успех, если значение равно ожидаемому. Обычно реализуется инструкцией процессора (например, `LOCK CMPXCHG` на x86).
- `compare_exchange_weak`: Разрешает **Spurious Failures** (ложные отказы). Может вернуть `false`, даже если значение равно ожидаемому.

Зачем нужен `weak`? На некоторых архитектурах (ARM, PowerPC) CAS реализуется через пару инструкций `LL/SC` (Load-Linked / Store-Conditional). Если между ними произошло прерывание или обращение к кэш-линии, `SC` может не сработать. `weak` версия позволяет избежать лишних циклов внутри самой инструкции CAS, перекладывая ответственность на внешний цикл пользователя. В циклах (`while`) всегда предпочтительнее использовать `weak`, так как цикл все равно перезапустится.

Пример: Lock-Free Stack (Treiber Stack)

Реализуем классический Lock-Free стек (LIFO). Это одна из простейших структур, так как все изменения происходят только на верхушке (`head`).

Узлы стека:

```
template <typename T>
struct Node {
    T data;
    Node* next;

    Node(const T& d) : data(d), next(nullptr) {}
};
```

Операция Push

```
std::atomic<Node<T>*> head{nullptr};

void Push(const T& data) {
    Node<T>* new_node = new Node<T>(data);

    // Фаза 1: Читаем текущую голову
    new_node->next = head.load(std::memory_order_relaxed);

    // Фаза 2: Пытаемся подменить голову на нашу новую ноду
    // Если head изменился (кто-то успел сделать push/pop),
    // CAS обновит new_node->next актуальным значением head
    // и мы попробуем снова.
```

```
    while (!head.compare_exchange_weak(new_node->next, new_node,
                                       std::memory_order_release,
                                       std::memory_order_relaxed)) {
        // Тело цикла пустое, вся работа в условии
    }
}
```

Это идеально работает. Новый узел невидим для других потоков, пока CAS не увенчается успехом. Как только CAS прошел, узел мгновенно становится новой головой.

Операция Pop и утечки памяти

```
std::optional<T> Pop() {
    Node<T>* old_head = head.load(std::memory_order_acquire);

    while (old_head && !head.compare_exchange_weak(old_head, old_head->next,
                                                   std::memory_order_acquire,
                                                   std::memory_order_relaxed)) {
        // Если CAS не прошел, old_head обновился. Проверяем, не стал ли он
        ↪ nullptr.
    }

    if (!old_head) return std::nullopt; // Стек пуст

    T res = old_head->data;

    // ПРОБЛЕМА: Когда удалять old_head?
    // delete old_head; // <--- ОПАСНО!

    return res;
}
```

Проблема безопасного удаления памяти

В коде выше строка `delete old_head` закомментирована не случайно. Это главная проблема Lock-Free структур в C++.

Представим сценарий: 1. Поток А читает `head` (адрес 0x100) в локальную переменную `old_head`. 2. Поток А засыпает перед CAS. 3. Поток Б делает `Pop`, успешно меняет `head`, забирает ноду 0x100 и вызывает `delete` (возвращает память ОС). 4. Поток А просыпается. Он пытается обратиться к `old_head->next` внутри CAS или просто сравнить значение. 5. **Use-After-Free:** Память по адресу 0x100 уже освобождена и, возможно, выделена под другой объект. Чтение `old_head->next` приводит к Segfault или чтению мусора.

В языках с Garbage Collector (Java, Go, C#) этой проблемы нет — GC не удалит объект, пока на него есть ссылки (включая локальную ссылку в потоке А). В C++ нам нужно реализовать собственный механизм управления памятью для Lock-Free.

Эта проблема и её решения (Hazard Pointers, RCU) настолько объемны, что им посвящена отдельная глава.

Спинлок (Spinlock) на атомиках

Для полноты картины реализуем примитивнейший мьютекс (спинлок) на `std::atomic_flag`. Это единственный тип, который гарантированно является Lock-Free на всех архитектурах (даже самых простых).

```
class Spinlock {
    // ATOMIC_FLAG_INIT устанавливает флаг в состояние false (clear)
    std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
    void lock() {
        // test_and_set устанавливает флаг в true и возвращает предыдущее
        ↪ значение.
        // Мы крутимся, пока предыдущее значение было true (значит, было занято).
        while (flag.test_and_set(std::memory_order_acquire)) {
            // Оптимизация для процессора: "я в цикле ожидания"
            // Снижает энергопотребление и позволяет SMT-потокам работать
            ↪ быстрее.
#if defined(__x86_64__) || defined(_M_X64)
            _mm_pause();
#elif defined(__aarch64__)
            asm volatile("yield");
#endif
        }
    }

    void unlock() {
        flag.clear(std::memory_order_release);
    }
};
```

Резюме раздела

- **Атомики** обеспечивают неделимость операций и видимость изменений между потоками (Memory Ordering).
- **CAS (Compare-And-Swap)** — универсальный примитив для построения Lock-Free алгоритмов.
- **CAS-Loop** позволяет применять произвольные функции к разделяемым данным оптимистично.
- Основная сложность Lock-Free не в алгоритмах изменения данных, а в **управлении памятью** (когда безопасно удалить узел?).

Глава 6

Advanced Lock-Free: ABA & Memory Reclamation

В предыдущей главе мы реализовали Lock-Free стек и столкнулись с фундаментальной проблемой отсутствия Garbage Collector в C++: мы не знаем, когда безопасно вызывать `delete` для узла, исключенного из структуры данных.

Однако проблема управления памятью в Lock-Free алгоритмах глубже, чем просто утечки или Segfault. Повторное использование памяти (memory recycling) может привести к логическому разрушению структуры данных, даже если все указатели валидны. Этот феномен известен как проблема ABA.

В этой главе мы разберем механику ABA на уровне состояний памяти, изучим аппаратные методы решения (Tagged Pointers) и реализуем программный алгоритм безопасной рекламации памяти — Hazard Pointers. Также мы рассмотрим более сложную структуру — Lock-Free очередь Майкла-Скотта.

Проблема ABA

Название ABA происходит от последовательности изменения состояний: значение было A, стало B, затем снова стало A. Для наивного наблюдателя, использующего CAS (Compare-And-Swap), кажется, что значение не менялось, хотя мир вокруг изменился кардинально.

Анатомия катастрофы (Сценарий на Стеке)

Рассмотрим наш Lock-Free стек (LIFO). Состояние стека: `Top` $\rightarrow$ `A` $\rightarrow$ `B` $\rightarrow$ `C`.

Поток 1 (Жертва):

- Планирует выполнить `Pop`.
- Читает текущую голову: `old_head = A` (адрес 0x1000).
- Читает следующий элемент: `next = A`$\rightarrow$`next` (узел B, адрес 0x2000).
- *Поток прерывается планировщиком ОС.*

Поток 2 (Разрушитель):

- Выполняет `Pop`: Успешно удаляет A. Стек: `Top` $\rightarrow$ `B` $\rightarrow$ `C`.
- Удаляет узел A (`delete A`). Память по адресу 0x1000 возвращается аллокатору.

- Выполняет `Pop`: Успешно удаляет B. Стек: `Top` $\rightarrow$ `C`.
- Удаляет узел B (`delete B`). Адрес 0x2000 свободен.
- Выполняет `Push`: Создает новый узел. Аллокатор, оптимизируя работу, возвращает только что освобожденный адрес 0x1000.
- Записывает в новый узел (по адресу 0x1000) какие-то данные. Его `next` указывает на C.
- Стек: `Top` $\rightarrow$ `A' (0x1000)` $\rightarrow$ `C`.

Поток 1 (Продолжение):

- Просыпается. Готовится выполнить CAS для смены головы.
- Аргументы CAS:
  - Адрес переменной: `&Top`.
  - Ожидаемое значение: 0x1000 (узел A).
  - Новое значение: 0x2000 (узел B, сохраненный в переменной `next`).
- **CAS Проверка:** Текущий `Top` равен 0x1000? Да, равен (это новый узел A').
- **CAS Успех:** `Top` меняется на 0x2000 (узел B).

Результат: Голова стека указывает на адрес 0x2000. Но узел по этому адресу был удален Потоком 2! Стек сломан. Следующий `Pop` приведет к чтению освобожденной памяти (Use-After-Free) или, что хуже, к чтению данных другого объекта, который аллокатор разместил по адресу 0x2000.

Решение 1: Tagged Pointers (Указатели с версиями)

Проблема ABA возникает из-за того, что указатель (адрес) не уникален во времени. Адрес 0x1000 сегодня — это узел стека, завтра — текстура видеокарты, послезавтра — снова узел стека.

Чтобы сделать указатель уникальным, нужно добавить к нему **счетчик версий** (Tag). Каждое обновление указателя (CAS) должно инкрементировать этот счетчик. Тогда последовательность $A \rightarrow B \rightarrow A$ превратится в $A_1 \rightarrow B_2 \rightarrow A_3$. CAS, ожидающий A_1, провалится, увидев A_3.

Реализация на x86_64

В 64-битных процессорах виртуальные адреса используют только младшие 48 бит. Старшие 16 бит обычно должны быть нулями (или копией 47-го бита). Мы можем использовать эти 16 бит для хранения счетчика.

```
// Упрощенная концепция (требует битхаков)
struct TaggedPointer {
    uint64_t raw; // 48 бит адрес + 16 бит тег

    static const uint64_t TAG_MASK = 0xFFFF000000000000;
    static const uint64_t PTR_MASK = 0x0000FFFFFFFFFFFF;

```

```
    Node* get_ptr() const {
        return reinterpret_cast<Node*>(raw & PTR_MASK);
    }

    uint16_t get_tag() const {
        return (raw & TAG_MASK) >> 48;
    }

    // При создании нового ptr инкрементируем tag
    static TaggedPointer make(Node* ptr, uint16_t old_tag) {
        return { (uint64_t(ptr) & PTR_MASK) | (uint64_t(old_tag + 1) << 48) };
    }
};
```

В C++ для этого удобно использовать двойную ширину CAS (`cmpxchg16b` на x86_64), который оперирует парой 64-битных чисел (итого 128 бит). Стандартный `std::atomic<shared_ptr>` часто реализован именно так, но это дорого.

Решение 2: Hazard Pointers (Опасные указатели)

Tagged Pointers решают ABA, но не решают проблему преждевременного `delete`. Самым надежным чисто программным решением в C++ (без использования GC) является метод **Hazard Pointers** (HP), предложенный Майклом (Maged Michael).

Идея: Каждый поток-читатель имеет свой личный список "опасных указателей" (обычно 1-2 слота). Перед тем как работать с указателем, поток записывает его в свой HP-слот, объявляя всем остальным: "Я читаю этот объект, не удаляйте его!".

Поток-писатель, желающий удалить объект, сначала сканирует списки HP всех потоков.

- Если адрес найден в чьем-то HP — удалять нельзя. Адрес добавляется в список "на пенсию" (Retire List).
- Если адрес нигде не найден — можно безопасно вызвать `delete`.

Алгоритм работы с HP в Pop

Это сложнее, чем кажется, из-за гонок данных. Мы не можем просто записать указатель в HP, потому что к моменту записи он уже мог быть удален.

```
// Глобальный массив Hazard Pointers (по одному слоту на поток)
std::atomic<Node*> HP[MAX_THREADS];

std::optional<T> Pop(int thread_id) {
    Node* old_head;
    Node* next_node;

    while (true) {
        // 1. Читаем текущую голову
        old_head = head.load(std::memory_order_acquire);
        if (!old_head) return std::nullopt;

```

```
        // 2. Объявляем указатель "опасным" (защищаем его)
        HP[thread_id].store(old_head, std::memory_order_seq_cst);

        // 3. КРИТИЧЕСКАЯ ПРОВЕРКА!
        // За время между шагом 1 и 2 old_head мог быть удален.
        // Если head изменился, значит, наш HP защищает уже "мертвый" объект.
        // Нужно начать сначала.
        if (head.load(std::memory_order_acquire) != old_head) {
            HP[thread_id].store(nullptr, std::memory_order_relaxed); // Снимаем
            ↪  защиту
            continue;
        }

        // 4. Теперь указатель безопасно защищен. Можно читать поля.
        next_node = old_head->next; // Безопасно!

        // 5. Пытаемся удалить из стека
        if (head.compare_exchange_weak(old_head, next_node)) {
            break; // Успех! Мы извлекли элемент.
        }

        // Неудача: снимаем защиту и пробуем снова
        HP[thread_id].store(nullptr, std::memory_order_relaxed);
    }

    HP[thread_id].store(nullptr, std::memory_order_release); // Мы закончили

    // 6. Отправляем old_head на удаление
    RetireNode(old_head);

    return old_head->data;
}
```

RetireNode и Scan

Функция `RetireNode` не делает `delete` сразу. Она добавляет указатель в локальный буфер потока (`thread_local vector`). Когда буфер заполняется (например, достигает размера $2 \cdot N$, где N — число потоков), запускается процедура `Scan`.

`Scan`: 1. Собирает все ненулевые указатели из глобального массива `HP` в `std::set` (или сортированный вектор). 2. Проходит по локальному буферу "пенсионеров". 3. Если указатель из буфера ЕСТЬ в множестве `HP` — оставляем его на потом. 4. Если указателя НЕТ в множестве `HP` — делаем реальный `delete`.

Это гарантирует, что мы никогда не удалим объект, который кто-то читает.

Решение 3: RCU (Read-Copy-Update)

Для структур, где чтений много, а записей мало, HP может быть слишком дорогим из-за необходимости писать в `atomic` (это сбивает кэш-линии). Альтернатива — RCU.

Идея: разделить время на "эпохи" (generations). 1. У нас есть глобальный счетчик эпох. 2. Писатель, удаляющий данные, "публикует" новую версию структуры данных, а старую

откладывает в список удаления текущей эпохи. 3. Писатель ждет, пока *все* читатели, начавшие работу в старой эпохе, завершат свои операции (Quiescent State). 4. После этого старую эпоху можно безопасно чистить.

В user-space RCU сложен в реализации (нужно знать, когда читатель "вышел" из критической секции), но библиотека `liburcu` предоставляет готовые решения.

Практика: Lock-Free Очередь (Queue)

Очередь (FIFO) сложнее стека, так как имеет две точки изменения: `Head` (откуда забираем) и `Tail` (куда добавляем). Классический алгоритм — **Michael-Scott Queue**.

Проблемы двух указателей

Если мы просто храним `atomic<Node*> Head` и `Tail`, мы не можем обновить их оба атомарно одной инструкцией. Вставка в конец требует двух шагов:

1. `Tail->next = new_node` (линковка).
2. `Tail = new_node` (продвижение хвоста).

Если поток заснет между 1 и 2, `Tail` останется указывать на предпоследний элемент. Следующий поток, пришедший делать `Push`, увидит, что `Tail->next != nullptr`.

Идея помощи (Helping): Lock-Free алгоритм очереди устроен так, что если поток видит "отставший" `Tail` (у которого `next` не null), он не ждет, а **помогает** завершить операцию, продвигая `Tail` вперед с помощью CAS, и только потом делает свою вставку.

Dummy Node (Фиктивный узел)

Чтобы избежать гонок на пустой очереди (когда `Head == Tail == nullptr`), очередь Майкла-Скотта всегда содержит хотя бы один "фиктивный" узел. Изначально: `Head = Tail = new Node(dummy)`.

- `Push`: Добавляет после `Tail`, двигает `Tail`.
- `Pop`: Читает `Head->next`. Если есть, возвращает данные, и делает `Head = Head->next`. Старый `Head` (бывший dummy) удаляется (через HP), а узел, который содержал данные, становится новым dummy.

Резюме раздела

- Проблема ABA разрушает логику Lock-Free алгоритмов при переиспользовании памяти.
- **Tagged Pointers** решают ABA аппаратно, добавляя счетчик версий в неиспользуемые биты указателя.
- **Hazard Pointers** — де-факто стандарт для управления памятью в C++ Lock-Free. Они гарантируют безопасность доступа, но требуют накладных расходов на `store-load` барьеры при каждом чтении.
- Реализация очередей требует кооперации потоков: если один "застрял" посередине операции, другие должны ему помочь, чтобы сохранить свойство Lock-Free.