HSE
Faculty of Computer Science

Конспект углубленного курса по C++

AUTHOR
Your Name

COURSE
Advanced C++
Fall 2023

Оглавление

Часть I

Лекция 11 – lock free

Глава 1

Анатомия синхронизации: Mutex vs Atomic и цена ожидания

Многопоточное программирование в C++ строится на примитивах синхронизации, предоставляемых стандартной библиотекой. В основе всех *lock-free* и *wait-free* алгоритмов лежит фундаментальный строительный блок — `std::atomic`.

Атомарные переменные: базовый API

Атомарная переменная (`std::atomic<T>`) гарантирует отсутствие гонок данных (*data race*) при одновременном доступе из нескольких потоков. Это достигается за счет использования специальных инструкций процессора (например, `LOCK XADD` на x86) или барьеров памяти.

Ключевая особенность атомиков в том, что операции над ними (загрузка, сохранение, инкремент) неделимы. Невозможна ситуация, когда один поток прочитал "половину" записанного значения или когда два потока одновременно инкрементировали переменную, но результат увеличился только на 1.

Базовые операции

```cpp
#include <atomic>

std::atomic<int> counter{0};

void worker() {
    // 1. Атомарная запись (Store)
    counter.store(10);

    // 2. Атомарное чтение (Load)
    int val = counter.load();

    // 3. Атомарная модификация (Read-Modify-Write)
    // Возвращает ПРЕДЫДУЩЕЕ значение
    int old_val = counter.fetch_add(1);

    // Эквивалентно fetch_add(1), но возвращает
```

```
17      // результат в зависимости от перегрузки (post/pre)
18      counter++;
19  }
```

API `std::atomic` различается для разных типов данных:

- **Целочисленные типы:** Поддерживают арифметику `fetch_add`, `fetch_sub`, `fetch_and`, `fetch_or`, `fetch_xor`.
- **Указатели:** Поддерживают адресную арифметику (например, `fetch_add` сдвигает указатель на размер типа).
- **Пользовательские структуры:** Поддерживают только `load`, `store` и `compare_exchange` (CAS).

Сравнительный анализ: Atomic vs Mutex

Существует распространенное заблуждение, что атомарные операции всегда быстрее, чем использование `std::mutex`. Это утверждение верно только для определенных сценариев и часто ложно при высокой конкуренции (*high contention*).

Бенчмарк: Инкремент счетчика

Рассмотрим классический бенчмарк: N потоков инкрементируют одну общую переменную.

Вариант 1: Mutex

```
std::mutex mtx;
int shared_data = 0;

void mutex_inc() {
    std::lock_guard<std::mutex> lock(mtx);
    shared_data++;
}
```

Вариант 2: Atomic

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

void atomic_inc() {
    shared_data.fetch_add(1, std::memory_order_relaxed);
}
```

При малом количестве потоков (1-4) атомарная версия работает быстрее, так как отсутствует накладной расход на захват блокировки. Однако, по мере роста числа потоков (например, до 16 и выше на 8-ядерном процессоре), производительность атомиков деградирует.

Физика процесса: Атомарный инкремент требует эксклюзивного владения кэш-линией (*cache coherency protocol*, например, MESI). Когда множество ядер пытаются записать в одну и

ту же переменную, кэш-линия начинает "метаться" между ядрами (cache ping-pong). Шина процессора насыщается трафиком синхронизации кэшей.

В случае `std::mutex`, потоки, не сумевшие захватить блокировку, уходят в режим ожидания (sleep) на уровне ядра ОС. Они перестают конкурировать за шину памяти, позволяя владельцу мьютекса быстро выполнить работу.

Резюме раздела

`std::mutex` может оказаться эффективнее `std::atomic` при высокой конкуренции, так как он сериализует доступ и убирает паразитную нагрузку на шину памяти от ожидающих потоков.

Анатомия ожидания: Spinlock, Futex и Throttling

Понимание того, как реализован `std::mutex` "под капотом", критически важно для выбора инструмента синхронизации.

Spinlock (Спинлок)

Простейшая реализация блокировки на атомиках — это Spinlock. Поток крутится в цикле, постоянно проверяя флаг, пока не получит доступ.

> **Spinlock**
>
> Механизм синхронизации, при котором ожидающий поток не приостанавливает свое выполнение, а находится в цикле активного ожидания (*busy wait*), потребляя такты процессора.

```
class Spinlock {
    std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
    void lock() {
        // test_and_set возвращает true, если флаг УЖЕ был установлен.
        // Если вернул false — мы успешно установили флаг (захватили лок).
        // memory_order_acquire — для корректной синхронизации.
        while (flag.test_and_set(std::memory_order_acquire)) {
            // Busy wait: поток сжигает CPU впустую
            // На x86 здесь часто ставят инструкцию _mm_pause()
        }
    }

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

Проблема Spinlock в Userspace: Использование спинлоков в прикладном коде (userspace) — опасный анти-паттерн.

1. **CPU Burning:** Поток занимает ядро на 100%, не выполняя полезной работы. Это тратит энергию и греет процессор.
2. **Priority Inversion:** Если высокоприоритетный поток ждет спинлок, захваченный низкоприоритетным потоком, а планировщик не дает времени владельцу, система может зависнуть.

Проблема троттлинга (Throttling)

Особая опасность спинлоков возникает в облачных средах (Docker, Kubernetes) с лимитами по CPU.

Представьте ситуацию:

1. Поток А захватывает спинлок.
2. Планировщик ОС (или гипервизор) прерывает выполнение потока А, так как его квант времени (timeslice) истек.
3. Поток Б пытается захватить тот же спинлок. Он начинает крутиться в `while`, сжигая свой квант времени.
4. Поток Б не может продвинуться, так как Поток А "спит" и не может освободить лок.

Если квота CPU исчерпана, поток А может быть "заморожен" на десятки миллисекунд (throttling latency). Все это время остальные потоки, ожидающие спинлок, просто сжигают процессорное время впустую.

Реализация Mutex через Futex

Современный `std::mutex` (например, в Linux через glibc/pthreads) использует гибридную схему, основанную на системном вызове **futex** (Fast Userspace Mutex).

Алгоритм работы `std::mutex::lock()`:

1. **Fast Path (Userspace):** Поток пытается атомарно (CAS) изменить состояние мьютекса с "свободен" на "занят". Если удается — блокировка захвачена без системного вызова. Это очень быстро (десятки наносекунд).
2. **Slow Path (Kernel Space):** Если мьютекс занят, поток делает системный вызов `futex_wait`. Ядро ОС переводит поток в состояние `BLOCKED` и убирает его из очереди планирования. Поток перестает потреблять CPU.

Когда владелец вызывает `unlock()`, он сначала освобождает флаг, а затем (если есть ожидающие) делает `futex_wake`, чтобы ядро разбудило один из спящих потоков.

> **Важно!**
>
> `std::mutex` эффективен, потому что ожидание переносится в ядро (*kernel space*). Spinlock же оставляет ожидание в пространстве пользователя, сжигая ресурсы.

Аппаратные ограничения и std::atomic<T>

Не любой тип `T` может быть обработан атомарно на аппаратном уровне. Процессоры обычно гарантируют атомарность только для типов, помещающихся в машинное слово (64 бита) или двойное слово (128 бит, инструкция `CMPXCHG16B` на x64).

is_always_lock_free

Стандарт C++ предоставляет механизм проверки, является ли атомик "настоящим" (lock-free) или эмулируется библиотекой.

```cpp
struct LargeData {
    char buffer[400]; // Слишком большой для регистра CPU
};

std::atomic<LargeData> data;

// Вернет false, так как процессор не умеет атомарно писать 400 байт
bool is_fast = std::atomic<LargeData>::is_always_lock_free;
```

Если `is_always_lock_free` ложно, компилятор реализует `std::atomic` через скрытый глобальный мьютекс (хеш-таблица мьютексов). В этом случае использование `std::atomic` теряет смысл с точки зрения производительности и свойства lock-free.

> **На заметку**
>
> Перед использованием `std::atomic` со структурами всегда проверяйте `is_always_lock_free` через `static_assert`.

Эволюция ожидания в C++20

До C++20 у разработчиков не было стандартного способа реализовать эффективное ожидание на атомарной переменной без спинлоков или использования сторонних мьютексов/condition variables.

В C++20 добавлены методы `wait`, `notify_one` и `notify_all` непосредственно в `std::atomic`. Это фактически предоставляет интерфейс футексов на уровне языка.

```cpp
#include <atomic>
#include <thread>

std::atomic<int> signal_flag{0};

void consumer() {
    // Эффективное ожидание:
    // Блокирует поток, пока signal_flag == 0.
    // Не потребляет CPU в цикле!
    int value = 0;
    while ((value = signal_flag.load()) == 0) {
        signal_flag.wait(0);
    }
    // ... работаем ...
}

void producer() {
    signal_flag.store(1);
    signal_flag.notify_one(); // Будим ожидающий поток
```

```
20 }
```

Метод `wait(old_val)` работает следующим образом:

1. Атомарно сравнивает текущее значение атомика с `old_val`.
2. Если значения не равны — немедленно возвращает управление (значение уже изменилось).
3. Если равны — поток усыпляется (как на мьютексе/футексе) до вызова `notify`.

Это позволяет строить эффективные механизмы синхронизации, сочетающие скорость атомиков (fast path) и энергоэффективность блокировок (slow path).

Глава 2

Теория гарантий прогресса: Lock-free и Wait-free

В разработке многопоточных систем термины *lock-free* и *wait-free* часто используются ошибочно как синонимы "высокой производительности". Однако в строгом академическом смысле это не метрики скорости, а гарантии прогресса системы (*progress guarantees*).

Эти гарантии описывают поведение алгоритма в условиях максимальной конкуренции (*contention*) или при нештатном поведении планировщика (например, приостановка потоков).

Lock-freedom: Глобальный прогресс

> **Lock-free (LF)**
>
> Алгоритм называется **lock-free**, если гарантируется, что при непрерывном выполнении системы хотя бы один поток сделает полезный прогресс за конечное число шагов.

Свойство *lock-freedom* является глобальным (system-wide). Оно не гарантирует, что конкретный поток T_1 когда-либо завершит свою операцию. Оно лишь гарантирует, что система в целом не зависнет. Если T_1 не может завершить операцию, это происходит только потому, что какой-то другой поток T_2 *успешно* завершил свою операцию, изменив состояние разделяемой структуры.

Критерий приостановки (Suspension Criterion)

Интуитивный способ проверки алгоритма на lock-freedom — мысленный эксперимент с "заморозкой" потока.

> **Важно!**
>
> Если мы произвольно остановим (suspend) любой поток (например, в отладчике или из-за троттлинга ОС) в любой точке выполнения, другие потоки **должны** иметь возможность продолжать выполнение и завершать свои операции.

Если остановка одного потока приводит к тому, что остальные потоки начинают бесконечно ожидать (как в случае с захваченным мьютексом), алгоритм **не является** lock-free.

Wait-freedom: Локальный прогресс

> **Wait-free (WF)**
>
> Алгоритм называется **wait-free**, если гарантируется, что **каждый** поток сделает прогресс (завершит операцию) за конечное, ограниченное число шагов, независимо от действий других потоков.

Это более строгое требование. Wait-free алгоритм исключает не только взаимные блокировки (deadlocks), но и голодание (starvation). Никакая конкуренция не может заставить поток выполняться бесконечно долго.

Иерархия гарантий выглядит так:

$$\text{Wait-free} \subset \text{Lock-free} \subset \text{Blocking (с блокировками)}$$

Любой wait-free алгоритм является lock-free, но обратное неверно.

Классификация алгоритмов синхронизации

Рассмотрим классические примитивы через призму гарантий прогресса.

1. Mutex и Spinlock (Blocking)

Использование `std::mutex` или самописного спинлока автоматически выводит алгоритм из категории lock-free.

```
std::mutex mtx;

void blocked_op() {
    mtx.lock();
    // CRITICAL SECTION
    // Если поток будет остановлен здесь (OS scheduler, crash),
    // то НИКТО больше не сможет войти в секцию.
    // Прогресс системы равен нулю.
    mtx.unlock();
}
```

Спинлок, реализованный на атомиках (как в Главе 1), технически использует lock-free инструкции процессора (CAS, test-and-set), но *алгоритмически* является блокирующим. Ожидание в цикле `while` зависит от действий другого потока (владельца), что нарушает определение lock-freedom.

2. CAS-Loop (Lock-free)

Классический паттерн для реализации атомарных транзакций (например, добавление в стек) — цикл Compare-And-Swap.

```
void lock_free_op(std::atomic<int>& shared_data) {
    int old_val = shared_data.load();
    int new_val;
```

```
4
5       do {
6           new_val = complex_calculation(old_val);
7           // Пытаемся атомарно обновить:
8           // Если shared_data == old_val, то пишем new_val и выходим (true).
9           // Если нет, в old_val записывается актуальное значение, повторяем
            ↪  (false).
10      } while (!shared_data.compare_exchange_weak(old_val, new_val));
11  }
```

Анализ прогресса:

- Представим, что 10 потоков одновременно зашли в этот цикл.
- Они конкурируют за `compare_exchange`.
- В конкретный момент времени только **один** поток выиграет гонку (у него CAS вернет `true`). Он совершит прогресс и выйдет из функции.
- Остальные 9 потоков получат `false`, обновят свои локальные копии `old_val` и пойдут на следующую итерацию.
- **Результат:** Система совершила прогресс (одна операция выполнена), несмотря на то, что 9 потоков "потратили время впустую".

Этот алгоритм — **Lock-free**, но не **Wait-free**, так как теоретически возможен сценарий, где "поток-неудачник" бесконечно проигрывает гонку CAS более быстрым потокам (livelock/starvation). Однако система в целом продолжает работать.

3. Атомарные Load/Store (Wait-free)

Простейшие операции над `std::atomic` (если они аппаратно поддерживаются) являются wait-free.

```
1  std::atomic<int> flag{0};
2
3  void set_flag() {
4      // Выполняется за фиксированное число инструкций процессора.
5      // Не содержит циклов.
6      // Не зависит от других потоков.
7      flag.store(1);
8  }
```

Wait-free алгоритмы сложнее в реализации для комплексных структур данных, так как требуют механизмов помощи (*helping*), когда один поток, видя, что другой застрял, помогает ему завершить операцию.

Зачем нужны эти гарантии?

Частое заблуждение: "Lock-free код быстрее". Как показано в бенчмарках (Глава 1), это не всегда так из-за высокой стоимости CAS-инструкций и cache-contention. Главная ценность lock-free не в пропускной способности (*throughput*), а в надежности и предсказуемости задержек (*latency*).

1. Устойчивость к инверсии приоритетов (Priority Inversion)

В системах с блокировками возможна ситуация, когда низкоприоритетный поток захватывает мьютекс и прерывается планировщиком. Высокоприоритетный поток, ожидающий этот мьютекс, вынужден ждать, пока низкоприоритетный получит процессорное время. В lock-free алгоритмах высокоприоритетный поток может просто выполнить свою операцию (через CAS), "перебив" или завершив работу за медленный поток.

2. Устойчивость к сбоям (Fault Tolerance)

Если поток, владеющий мьютексом, аварийно завершается (segfault, kill), мьютекс остается "зависшим" (в несогласованном состоянии), что приводит к дедлоку всей системы (если не использовать специальные *robust mutexes*). Lock-free структуры данных всегда остаются в согласованном состоянии: сбой потока посередине CAS-цикла не портит данные, так как изменение либо произошло целиком, либо не произошло вовсе.

Резюме раздела

- **Lock-free:** Гарантирует отсутствие взаимных блокировок (deadlock). Система всегда движется вперед, но отдельные потоки могут голодать.
- **Wait-free:** Гарантирует отсутствие голодания (starvation). Время выполнения операции ограничено сверху.
- Lock-free алгоритмы необходимы там, где недопустима блокировка всей системы из-за остановки одного потока (системы реального времени, обработчики сигналов, ядра ОС).

Глава 3

Паттерн CAS-Loop и сложные атомарные операции

Стандартная библиотека C++ предоставляет ограниченный набор атомарных операций модификации: сложение (`fetch_add`), вычитание (`fetch_sub`) и побитовые операции (`and`, `or`, `xor`). Однако на практике часто требуются более сложные транзакции, например, атомарное умножение, вычисление максимума или обновление битовых полей по маске.

Для реализации любой произвольной операции Read-Modify-Write (RMW) используется универсальный примитив: **Compare-And-Swap (CAS)**.

Анатомия Compare-And-Swap

Инструкция CAS — это самая "тяжелая" и мощная операция в арсенале атомиков. На ней строятся все lock-free структуры данных.

В C++ она представлена семейством методов `compare_exchange`.

Сигнатура и семантика

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

Ключевая особенность, которую часто упускают новички: первый аргумент `expected` передается **по ссылке**. Он играет двойную роль: входного параметра (ожидание) и выходного (реальность).

Логика работы метода (псевдокод):

```
// Атомарно внутри процессора:
if (this->value == expected) {
    this->value = desired; // Успех: записали новое значение
    return true;
} else {
    expected = this->value; // Неудача: обновили expected реальным значением
    return false;
}
```

Это избавляет программиста от необходимости делать повторный `load()` при неудаче. Метод сам сообщает: "Ты ожидал А, но там лежит Б. Я записал Б в твою переменную `expected`, попробуй еще раз".

Weak vs Strong: Цена гарантий

Стандарт C++ разделяет CAS на две версии.

> **Spurious Failure (Ложный отказ)**
>
> Ситуация, когда `compare_exchange` возвращает `false`, и значение не обновляется, даже если текущее значение атомика **равно** `expected`.

compare_exchange_weak

Разрешает ложные отказы. На некоторых архитектурах (ARM, PowerPC) атомарные операции реализуются через пару инструкций LL/SC (Load Linked / Store Conditional). Если между ними произошло прерывание или другой поток тронул кэш-линию, запись может не пройти, даже если значение не менялось.

- **Преимущество:** Генерирует более эффективный машинный код на RISC-архитектурах.
- **Использование:** Всегда используется в циклах.

compare_exchange_strong

Гарантирует отсутствие ложных отказов. Возвращает `false` только если значение реально не совпадает.

- **Цена:** На RISC-архитектурах компилятор генерирует вложенный цикл для сокрытия ложных отказов.
- **Использование:** Когда операция выполняется однократно (вне цикла) и повтор вычисления `desired` слишком дорог.

> **Важно!**
>
> В 99% случаев при написании lock-free алгоритмов вы будете использовать паттерн цикла. В цикле **всегда** используйте `weak` версию. Использование `strong` в цикле приводит к двойной вложенности циклов на уровне ассемблера.

Реализация Atomic Fetch-Max

Рассмотрим задачу: реализовать атомарную функцию, обновляющую переменную значением `max(current, value)`. Стандартного `fetch_max` нет.

Мы используем паттерн **CAS-Loop**.

```
#include <atomic>
#include <algorithm>

```

```
void atomic_fetch_max(std::atomic<int>& atom, int val) {
    // 1. Предварительная загрузка текущего состояния (Snapshot)
    // Можно использовать memory_order_relaxed, так как порядок пока не важен
    int prev = atom.load(std::memory_order_relaxed);

    // 2. Цикл попыток (CAS Loop)
    while (prev < val) {
        // 3. Попытка атомарной подмены
        // Если atom == prev, то записать val. Вернуть true.
        // Если atom ≠ prev, то prev = atom. Вернуть false.
        if (atom.compare_exchange_weak(prev, val,
                                       std::memory_order_release,
                                       std::memory_order_relaxed)) {
            break; // Успех!
        }

        // 4. Неявная ветвь else (при неудаче):
        // compare_exchange_weak УЖЕ обновил переменную 'prev'
        // актуальным значением из атомика.
        // Мы просто идем на следующую итерацию while.
        // Там условие (prev < val) проверится заново с новым prev.
    }
}
```

Построчный разбор механики

1. `int prev = atom.load(...)`: Мы делаем локальную копию значения. Это наше предположение о состоянии мира.
2. `while (prev < val)`: Оптимизация. Если текущее значение в атомике уже больше или равно `val`, нам не нужно ничего писать. Мы выходим без дорогих операций записи.
3. `compare_exchange_weak(prev, val, ...)`: Это точка синхронизации.
   - **Сценарий Успеха:** Между `load` (или предыдущей попыткой) и этой инструкцией никто не трогал `atom`. Значение успешно обновлено.
   - **Сценарий Провала:** Другой поток успел записать свое значение. В `atom` теперь лежит условное число 100, а мы думали там 50.
4. **Автоматическое обновление:** При провале переменная `prev` по ссылке получает значение 100. На следующей итерации цикла мы проверим `100 < val` и попробуем сделать CAS(100, val).

Доказательство свойства Lock-free

Почему этот алгоритм является Lock-free, но не Wait-free?

1. Не Wait-free: Теоретически возможна ситуация, когда поток-неудачник бесконечно крутится в цикле `while`. Это происходит, если между моментом проверки условия и вызовом CAS всегда вклинивается другой поток, изменяющий переменную. Время выполнения операции для конкретного потока не ограничено сверху.

2. Является Lock-free: Цикл продолжается только тогда, когда CAS возвращает `false`. CAS

возвращает `false` (игнорируя spurious failure) только тогда, когда значение в атомике **изменилось**. Значение меняется только в том случае, если какой-то другой поток успешно выполнил свой CAS (или `store`).

Резюме раздела

Следовательно, бесконечное выполнение цикла в одном потоке означает бесконечное количество успешных обновлений данных в других потоках. Глобальный прогресс системы гарантирован.

На заметку

CAS-loop — это оптимистичная блокировка. Мы предполагаем, что конфликта не будет, делаем работу локально, а в конце пытаемся "закоммитить" результат. Если конфликт был — откатываемся и повторяем. Это выгоднее мьютексов при низкой и средней конкуренции.

Глава 4

Модели памяти: Sequentially Consistent vs Relaxed

При работе с атомарными переменными программист обязан явно или неявно выбирать **модель памяти** (Memory Order). Этот выбор определяет, какие гарантии компилятор и процессор предоставляют относительно порядка выполнения инструкций и видимости изменений между потоками.

Ошибочный выбор модели памяти приводит к гейзенбагам, которые невозможно воспроизвести в отладчике, но которые гарантированно возникают под высокой нагрузкой на специфическом оборудовании (например, на архитектуре ARM).

Sequentially Consistent (SC)

Модель по умолчанию в C++ — `std::memory_order_seq_cst` (Sequentially Consistent).

> **Sequential Consistency**
>
> Модель исполнения, при которой результат работы многопоточной программы эквивалентен некоторому последовательному чередованию (*interleaving*) инструкций всех потоков на одноядерном процессоре.

Это самая сильная ("строгая") модель. Она интуитивно понятна человеку, так как сохраняет причинно-следственные связи.

Гарантии SC:

1. **Total Order:** Существует единый глобальный порядок всех модификаций всех SC-атомиков. Все потоки видят изменения в одном и том же порядке.
2. **Отсутствие переупорядочивания:** Инструкции исходного кода не могут "перепрыгивать" через SC-операции ни вверх, ни вниз.

Цена абстракции

За простоту приходится платить производительностью. Чтобы обеспечить SC, компилятор обязан:

- Отключить многие оптимизации (перестановка инструкций).

- Вставить аппаратные барьеры памяти (*memory fences/barriers*). На x86 это часто инструкции `MFENCE` или `LOCK prefixed instructions`, которые сбрасывают конвейер процессора и принудительно синхронизируют буферы записи.

Аппаратная реальность: Store Buffers

Чтобы понять необходимость более слабых моделей, нужно рассмотреть архитектуру процессора. Запись в оперативную память (RAM) — операция экстремально медленная (сотни тактов CPU).

Чтобы не останавливать конвейер на каждой инструкции записи, процессоры используют **Store Buffer** (буфер записи).

Механика:

1. Ядро выполняет инструкцию записи `x = 1`.
2. Значение помещается в локальный Store Buffer ядра, а не в кэш L1/RAM. Процессор продолжает выполнение следующих инструкций **мгновенно**.
3. Спустя время буфер сбрасывается (*flush*) в когерентный кэш, и только тогда изменение становится видимым другим ядрам.

> **Важно!**
>
> Наличие Store Buffer приводит к тому, что порядок *выполнения* инструкций (Program Order) не совпадает с порядком *видимости* изменений (Visibility Order) для других ядер.

Relaxed Ordering (Ослабленная модель)

Модель `std::memory_order_relaxed` снимает все ограничения на синхронизацию и порядок видимости, оставляя только одну гарантию: **атомарность** самой операции над переменной.

- **Нет Happends-Before:** Запись `x.store(1, relaxed)` в одном потоке не гарантирует, что другой поток увидит это значение "своевременно" относительно других переменных.
- **Свобода компилятора:** Компилятор может менять местами Relaxed-инструкции с обычными операциями, если это не ломает однопоточную логику.

Это самая дешевая модель. На большинстве архитектур она компилируется в обычные инструкции `MOV` без барьеров.

Парадокс Store Buffer: "Невозможный" результат

Рассмотрим классический пример, демонстрирующий разницу между SC и Relaxed (пример Деккера).

У нас есть две атомарные переменные `x` и `y`, инициализированные нулями.

```cpp
std::atomic<int> x{0}, y{0};

// Поток 1
void thread_1() {
    x.store(1, std::memory_order_relaxed); // (A)
    int r1 = y.load(std::memory_order_relaxed); // (B)
}

// Поток 2
void thread_2() {
    y.store(1, std::memory_order_relaxed); // (C)
    int r2 = x.load(std::memory_order_relaxed); // (D)
}
```

Анализ в Sequential Consistency

Если бы мы использовали `seq_cst`, возможны разные чередования, например:

- $A \to B \to C \to D \implies r1 = 0, r2 = 1$
- $C \to D \to A \to B \implies r1 = 1, r2 = 0$
- $A \to C \to B \to D \implies r1 = 1, r2 = 1$

Исход $r1 = 0, r2 = 0$ **невозможен**. Чтобы $r1$ был 0, (B) должно выполниться до (C). Чтобы $r2$ был 0, (D) должно выполниться до (A). Это создает цикл противоречий.

Анализ в Relaxed (Store Buffer Reality)

При использовании `relaxed` исход $r1 = 0, r2 = 0$ становится **возможным**.

Физическое объяснение:

1. Поток 1 выполняет (A): `x=1`. Значение попадает в Store Buffer ядра 1. В основной памяти x все еще 0.
2. Поток 1 сразу выполняет (B): читает y. В памяти y равно 0. Результат: `r1=0`.
3. Поток 2 выполняет (C): `y=1`. Значение попадает в Store Buffer ядра 2.
4. Поток 2 сразу выполняет (D): читает x. Ядро 2 не видит буфер ядра 1. В памяти x все еще 0. Результат: `r2=0`.
5. Спустя время буферы сбрасываются в память, но слишком поздно — потоки уже прочитали нули.

С точки зрения внешнего наблюдателя, произошло переупорядочивание: чтение (Load) выполнилось раньше, чем запись (Store) — эффект **StoreLoad Reordering**.

Резюме раздела

- Используйте `seq_cst` по умолчанию. Это безопасно и предсказуемо.
- Используйте `relaxed` только для счетчиков статистики или когда порядок видимости переменной абсолютно не влияет на логику других потоков.
- Никогда не используйте `relaxed` для реализации механизмов публикации данных (флаг готовности, указатель на объект), так как данные могут стать видимы *позже*, чем флаг готовности.

Глава 5

Практикум: Разработка Lock-free Stack и его фатальные ошибки

Разработка lock-free контейнеров кардинально отличается от разработки их блокирующих аналогов. Если в `std::stack` под мьютексом мы можем использовать `std::vector`, то в lock-free мире динамический массив практически неприменим. Переаллокация памяти (resizing) требует эксклюзивного доступа ко всему массиву, что противоречит идее lock-free.

Поэтому стандартом де-факто для lock-free стека является односвязный список (Treiber Stack).

Базовая архитектура

Структура данных предельно проста: атомарный указатель на вершину списка (`head`).

```
#include <atomic>

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

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

template <typename T>
class LockFreeStack {
    std::atomic<Node<T>*> head;

public:
    LockFreeStack() : head(nullptr) {}
    // ... методы ...
};
```

Реализация операции Push

Операция вставки элемента является безопасной и относительно простой. Мы создаем новый узел (который пока невидим для других потоков) и атомарно подменяем `head`.

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

    // 1. Загружаем текущую голову
    new_node->next = head.load();

    // 2. CAS-цикл
    // Мы пытаемся сделать new_node новой головой.
    // Если head изменился за это время, CAS обновит new_node->next
    // (через параметр expected) и мы попробуем снова.
    while (!head.compare_exchange_weak(new_node->next, new_node)) {
        // Тело цикла пустое.
        // new_node->next автоматически обновлен актуальным значением head.
    }
}
```

Этот код корректен и является lock-free. Новый узел является локальным для потока до момента успешного CAS, поэтому гонок данных на нем нет.

Наивная реализация Pop

Кажется, что удаление элемента симметрично вставке. Нам нужно прочитать `head`, запомнить следующий элемент и атомарно переставить `head` на следующий.

```
// ВНИМАНИЕ: ЭТОТ КОД СОДЕРЖИТ КРИТИЧЕСКИЕ ОШИБКИ
bool pop(T& result) {
    Node<T>* old_head = head.load();

    while (old_head &&
           !head.compare_exchange_weak(old_head, old_head->next)) {
        // Если CAS не удался, old_head обновлен.
        // Крутимся, пока не удалим или пока стек не станет пустым.
    }

    if (old_head == nullptr) {
        return false; // Стек пуст
    }

    result = old_head->data; // Извлечение данных
    delete old_head;         // Освобождение памяти
    return true;
}
```

Данная реализация иллюстрирует "проблему наивного lock-free". Она компилируется, проходит простые тесты, но неизбежно приведет к падению (segfault) и коррупции данных под нагрузкой.

Фатальная ошибка №1: Use-After-Free

В многопоточной среде нельзя просто так разыменовать указатель, полученный из общей структуры данных. Между моментом чтения указателя и доступом к его полям объект мог быть удален другим потоком.

В строке `compare_exchange_weak(old_head, old_head→next)` происходит обращение к `old_head→next`. Рассмотрим сценарий краха:

1. **Поток A:** Читает `head`. Пусть это адрес `0x1000`.
2. **Поток A:** Прерывается планировщиком ОС сразу после чтения, но ДО вызова CAS.
3. **Поток B:** Вызывает `pop()`. Успешно удаляет узел `0x1000`.
4. **Поток B:** Вызывает `delete 0x1000`. Память возвращена ОС или аллокатору.
5. **Поток A:** Просыпается. Пытается вычислить второй аргумент для CAS: `old_head->next`.
6. **Результат:** Обращение к освобожденной памяти (`0x1000`). Segmentation Fault.

> **Важно!**
>
> В lock-free алгоритмах с динамической памятью **нельзя** использовать `delete` сразу после извлечения узла, если есть вероятность, что другие потоки всё ещё держат на него ссылки.

Фатальная ошибка №2: Проблема ABA

Даже если мы решим проблему чтения мусора (например, не будем удалять память, а положим её в список переиспользования), нас поджидает проблема ABA (ABA Problem). Она связана с тем, что CAS проверяет равенство *значений* (адресов), а не *идентичность* объектов во времени.

Пусть стек выглядит так: **Top -> A -> B -> C**.

1. **Поток 1:** Начинает `pop`.
   - Читает `old_head = A`.
   - Читает `next = A→next` (это **B**).
   - Засыпает перед CAS. Он "думает", что хочет заменить A на B.
2. **Поток 2:** Вытесняет Поток 1 и работает агрессивно.
   - `pop()`: Удаляет A. Стек: **B -> C**. Узел A освобожден.
   - `pop()`: Удаляет B. Стек: **C**. Узел B освобожден.
   - `push(D)`: Добавляет D. Стек: **D -> C**.
   - `push(E)`: Аллокатор памяти, видя, что адрес от узла A свободен, **переиспользует** его для нового узла E.
   - Теперь адрес E равен адресу A.
   - Стек: **A(новый) -> D -> C**.
3. **Поток 1:** Просыпается.

- Выполняет `CAS(expected=A, desired=B)`.
- Процессор сравнивает текущий `head` и `expected`. Оба равны адресу A. **CAS успешен!**
- Поток 1 устанавливает `head` в **B**.

Катастрофа: Текущее состояние стека: **Top -> B**. Но узел B был давно удален! Узлы D и C потеряны (утечка памяти), структура списка сломана, `head` указывает на освобожденную память.

Стратегии решения

Написание корректного lock-free стека (MPMC — Multi-Producer Multi-Consumer) требует сложных схем управления памятью (Memory Reclamation).

1. Tagged Pointers (Версионирование)

В 64-битных процессорах используются не все 64 бита адреса (обычно только 48). Мы можем упаковать счетчик обновлений (tag) в неиспользуемые биты указателя. При каждом изменении `head` мы инкрементируем счетчик. Теперь CAS сравнивает пару `{ptr, counter}`. В примере с ABA: адрес будет тот же (A), но счетчик изменится. CAS провалится.

2. Hazard Pointers

Потоки публикуют список указателей ("Hazard Pointers"), которые они сейчас читают, в глобальном массиве. Поток, желающий удалить узел, сначала сканирует массивы Hazard Pointers всех потоков. Если на узел кто-то смотрит — удаление откладывается.

3. Ослабление требований (MPSC)

Если алгоритм допускает наличие только одного потребителя (Single Consumer), то проблемы ABA и Use-After-Free при `pop` исчезают, так как конфликт возникает только между методами `pop` и `pop`. `push` остается безопасным для многих потоков.

Резюме раздела

Lock-free программирование — это не просто расстановка атомиков. Это борьба с моделью памяти, аллокатором и самой природой конкурентности. Без применения Hazard Pointers или RCU написание корректного связного списка на C++ невозможно.