

HSE

Faculty of Computer Science

# Конспект углубленно- го курса по C++

---

AUTHOR

Your Name

COURSE

Advanced C++  
Fall 2023

# Оглавление

<b>I</b>	<b>Лекция 08 – Baby thread</b>	<b>3</b>
<b>1</b>	<b>Аппаратные основы многопоточности и роль ОС</b>	<b>4</b>
1.1	Эволюция архитектуры процессоров	4
1.1.1	От однопоточной парадигмы к SMP	4
1.2	Проблема Memory Wall	5
1.2.1	Физические ограничения и латентность	5
1.2.2	Иерархия кэшей	5
1.3	Hyper-threading (SMT)	6
1.3.1	Концепция виртуальных ядер	6
1.4	Роль Операционной Системы	6
1.4.1	Планирование и Квант Времени	6
1.4.2	Переключение контекста (Context Switch)	6
1.4.3	Миграция потоков и Processor Affinity	7
1.5	NUMA (Non-Uniform Memory Access)	7
<b>2</b>	<b>Управление потоками в C++: API и стоимость абстракций</b>	<b>8</b>
2.1	Базовый API: <code>std::thread</code>	8
2.1.1	Запуск и передача аргументов	8
2.2	Проблема времени жизни: <code>join()</code> и <code>detach()</code>	9
2.2.1	Ловушка деструктора <code>std::thread</code>	9
2.3	RAII для потоков: <code>std::jthread</code>	10
2.4	Стоимость создания потока	10
2.4.1	Бенчмарк: Создание потоков в цикле	10
2.5	Проблема Oversubscription	11
2.6	Ловушка <code>std::async</code>	11
2.7	Решение: Thread Pool	12
<b>3</b>	<b>Гонки данных и модель памяти</b>	<b>13</b>
3.1	Race Condition vs Data Race	13
3.2	Пример 1: Вывод в консоль (Race Condition)	13
3.2.1	Решение: <code>std::osyncstream</code>	14
3.3	Пример 2: Инкремент (Data Race)	14
3.3.1	Анатомия гонки	15
3.3.2	Почему это Undefined Behavior?	15
3.4	ThreadSanitizer (TSan)	16
3.5	Ловушка <code>std::vector&lt;bool&gt;</code>	16
<b>4</b>	<b>Синхронизация: Мьютексы и Взаимные блокировки</b>	<b>18</b>
4.1	<code>std::mutex</code> и Критическая секция	18
4.1.1	Почему "сырые" <code>lock()</code> и <code>unlock()</code> опасны	18
4.2	RAII: <code>lock_guard</code> и <code>unique_lock</code>	19
4.2.1	<code>std::lock_guard</code>	19
4.2.2	<code>std::unique_lock</code>	19
4.3	Shared Mutex (Reader-Writer Lock)	20
4.4	Deadlock (Взаимная блокировка)	20

4.4.3	Решение 2: <code>std::lock</code> и <code>std::scoped_lock</code>	21
<b>5</b>	<b>Атомики и низкоуровневая синхронизация</b>	<b>23</b>
5.1	Миф о <code>volatile</code>	23
5.2	<code>std::atomic</code>	24
5.2.1	Базовые операции: <code>load</code> и <code>store</code>	24
5.2.2	Read-Modify-Write (RMW)	24
5.3	Compare-And-Swap (CAS)	25
5.4	Реализация Spinlock	25
5.5	Spinlock vs Mutex: Цена решения	26
5.6	Модель памяти (Memory Ordering)	27
<b>6</b>	<b>Оптимизация под железо: False Sharing</b>	<b>28</b>
6.1	Физика кэш-линий	28
6.1.1	Протоколы когерентности (MESI)	28
6.2	Феномен False Sharing	29
6.2.1	Сценарий "Пинг-понг"	29
6.3	Демонстрация и диагностика	29
6.3.1	Анализ памяти	30
6.4	Решение: Padding и alignas	31
6.5	Проблема переносимости и ABI	31
6.5.1	Дилемма ABI	31
<b>7</b>	<b>Практикум: Управление памятью и Smart Pointers в многопоточности</b>	<b>33</b>
7.1	Потокобезопасность <code>std::shared_ptr</code>	33
7.1.1	Гарантии стандарта	33
7.2	Thread Local Storage (TLS)	34
7.3	Case Study: Сборщик мусора (Mark and Sweep)	35
7.3.1	Почему <code>shared_ptr</code> не подходит?	35
7.3.2	Алгоритм	35
7.3.3	Инструментарий: Leak Sanitizer	36

## **Часть I**

### **Лекция 08 – Baby thread**

# Глава 1

## Аппаратные основы многопоточности и роль ОС

Современное программирование на C++ невозможно рассматривать в отрыве от аппаратной архитектуры. Производительность кода, особенно в многопоточных приложениях, определяется не столько сложностью алгоритма, сколько эффективностью использования иерархии памяти и конвейера процессора. В этой главе мы рассмотрим эволюцию вычислительных систем от одноядерных процессоров до современных многоядерных архитектур с NUMA, а также разберем фундаментальные физические ограничения, известные как Memory Wall.

### Эволюция архитектуры процессоров

#### От одноядерной парадигмы к SMP

На протяжении десятилетий доминировала парадигма последовательного исполнения кода. Программисты писали инструкции, полагая, что они будут выполнены процессором одна за другой в строгом порядке. Рост производительности обеспечивался увеличением тактовой частоты и усложнением микроархитектуры (суперскалярность, предсказание переходов).

В начале 2000-х годов этот подход достиг физического предела. Дальнейшее повышение частоты приводило к экспоненциальному росту тепловыделения и энергопотребления. Решением стал переход от наращивания частоты одного ядра к увеличению количества ядер на кристалле (Chip Multiprocessors, CMP).

#### SMP и CMP

**SMP (Symmetric Multiprocessing)** — архитектура, в которой два и более одинаковых процессора подключены к общей памяти.

**CMP (Chip Multiprocessor)** — реализация SMP, где несколько ядер размещены на одном кристалле кремния. В 2005 году появились первые массовые многоядерные процессоры.

Переход к многоядерности изменил парадигму разработки программного обеспечения. "Бесплатное" ускорение программ с выходом нового процессора прекратилось. Теперь для утилизации вычислительной мощности требуется явное распараллеливание задач.

## Проблема Memory Wall

Фундаментальным ограничением производительности современных систем является диспропорция между скоростью процессора и скоростью доступа к оперативной памяти. Этот феномен получил название **Memory Wall**.

### Физические ограничения и латентность

За последние 30 лет производительность процессоров выросла на порядки, в то время как латентность (задержка доступа) памяти осталась практически неизменной.

#### Важно!

##### Ключевые показатели латентности:

- **1 такт CPU (3-4 ГГц):**  $\approx 0.25 - 0.3$  нс.
- **Доступ в L1 Cache:**  $\approx 1$  нс (3-4 такта).
- **Доступ в RAM:**  $\approx 70 - 100$  нс.

Это означает, что один промах мимо кэша (cache miss) стоит процессору сотен тактов простаивания. Если данные находятся в оперативной памяти, процессор вынужден ждать их получения, не выполняя полезной работы.

Физическая причина этого ограничения кроется в скорости света. Сигнал в вакууме проходит 30 см за 1 нс. В среде (медь, кремний) скорость ниже. Учитывая размеры кристалла и расстояние до модулей памяти (DIMM), сделать память одновременно большой и быстрой физически невозможно. Большая память требует длинных шин адреса и данных, что увеличивает задержку.

### Иерархия кэш-памяти

Для смягчения эффекта Memory Wall используется многоуровневая иерархия кэш-памяти.

- **L1 Cache (Level 1):** Самый маленький (обычно 32-64 КБ), расположен непосредственно в ядре. Разделен на кэш инструкций (L1i) и кэш данных (L1d). Латентность: единицы тактов.
- **L2 Cache (Level 2):** Больше по объему (256 КБ - 1 МБ), чуть медленнее. Обычно является приватным для ядра.
- **L3 Cache (Level 3):** Большой (десятки МБ), общий для всех ядер на кристалле (Last Level Cache, LLC). Латентность: десятки тактов.

Задача кэша — хранить данные, к которым процессор обращался недавно (временная локальность) или которые находятся рядом с ними (пространственная локальность). Когда данные загружаются из памяти, они загружаются не побайтово, а целыми блоками — **кэш-линиями** (cache lines), обычно размером 64 байта.

#### На заметку

Эффективное программирование на C++ подразумевает работу с данными, которые хорошо укладываются в кэш. Последовательный доступ к массиву (vector) на порядки быстрее, чем доступ к связному списку (list), узлы которого разбросаны по куче.

## Hyper-threading (SMT)

В попытках скрыть латентность памяти и максимально загрузить исполнительные устройства (ALU, FPU) была разработана технология SMT (Simultaneous Multithreading), наиболее известная в реализации Intel как Hyper-threading.

### Концепция виртуальных ядер

Идея SMT заключается в том, что одно физическое ядро представляется операционной системе как два (или более) логических процессора. Ядро дублирует архитектурное состояние (регистры, счетчик команд), но исполнительные устройства (ALU, FPU, L1 кэш) остаются общими.

#### Важно!

**Hyper-threading не удваивает производительность!** Два потока на одном ядре конкурируют за одни и те же вычислительные блоки.

Механизм эффективен в сценариях, когда один поток ожидает данные из памяти (memory stall). В этот момент ядро может переключиться на исполнение инструкций второго потока, загружая простаивающие ALU. Если же оба потока выполняют интенсивные вычисления (compute-bound) и не ждут памяти, выигрыш от SMT будет минимальным или даже отрицательным из-за конкуренции за кэш.

Для операционной системы и программиста это выглядит как наличие  $2N$  ядер, где  $N$  — число физических ядер. Например, 8 физических ядер видны как 16 логических потоков.

## Роль Операционной Системы

Операционная система (ОС) выступает абстракцией над аппаратным обеспечением, предоставляя программам иллюзию монопольного владения процессором.

### Планирование и Квант Времени

ОС использует **вытесняющую многозадачность** (preemptive multitasking). Каждому активному потоку выделяется интервал времени — **квант** (time slice), в течение которого он исполняется на процессоре.

- Длительность кванта в Linux (CFS scheduler) варьируется, но порядок величины — 10-100 мс.
- Это значение огромно по меркам процессора ( $100 \text{ мс} = 10^8 \text{ нс} \approx 4 \cdot 10^8 \text{ тактов}$ ).
- Человеческий глаз не замечает переключений с такой частотой, создавая иллюзию параллельной работы множества приложений.

Когда квант истекает или поток блокируется (например, на вводе-выводе), происходит аппаратное прерывание таймера, и управление передается планировщику ОС.

### Переключение контекста (Context Switch)

Процедура смены исполняемого потока называется переключением контекста.

1. Сохраняется состояние текущего потока: значения регистров общего назначения (RIP, RSP, RAX и др.), флагов, регистров FPU/AVX.
2. Выбирается следующий поток из очереди готовых к исполнению (runqueue).
3. Восстанавливается состояние нового потока (загрузка регистров).

Объем сохраняемых данных невелик (сотни байт), и сама операция быстрая. Однако косвенная стоимость переключения контекста может быть высокой из-за "остывания" кэша: новый поток, скорее всего, будет работать с другими данными, что приведет к серии кэш-промахов.

## Миграция потоков и Processor Affinity

Планировщик ОС старается удерживать поток на одном и том же ядре (Processor Affinity), чтобы сохранить "прогретый" кэш (L1/L2). Миграция потока на другое ядро — дорогая операция, так как данные придется загружать в кэш нового ядра заново (из L3 или RAM).

## NUMA (Non-Uniform Memory Access)

В многопроцессорных системах (серверах с несколькими сокетами) используется архитектура NUMA. Память физически разделена между процессорами.

- У каждого процессора есть "своя" (локальная) память, подключенная напрямую. Доступ к ней быстрый.
- Доступ к памяти другого процессора (удаленная память) происходит через межпроцессорную шину (например, Intel QPI/UPI) и является более медленным.

Игнорирование NUMA-топологии в высокопроизводительных приложениях может привести к существенной деградации производительности из-за трафика по межпроцессорной шине и повышенной латентности.

### Резюме раздела

- Производительность упирается в память (Memory Wall). Доступ к RAM стоит сотни тактов.
- Иерархия кэшей критически важна. Локальность данных определяет скорость программы.
- Hyper-threading позволяет скрыть задержки памяти, утилизируя простаивающие такты ядра.
- ОС переключает потоки раз в квант времени (десятки мс), сохраняя регистровый контекст.
- Миграция потоков между ядрами нежелательна из-за потери горячего кэша.

## Глава 2

# Управление потоками в C++: API и стоимость абстракций

Понимание аппаратной части многопоточности, рассмотренное в предыдущей главе, позволяет нам перейти к программным абстракциям. Стандарт C++11 ввел в язык нативную поддержку потоков, предоставив класс `std::thread` как обертку над системными вызовами (pthreads в Linux, WinAPI Threads в Windows). В этой главе мы разберем не только синтаксис, но и цену, которую приходится платить за создание потоков, а также распространенные архитектурные ошибки, приводящие к деградации производительности.

## Базовый API: `std::thread`

Класс `std::thread` (заголовочный файл `<thread>`) представляет собой объект, владеющий системным потоком исполнения. Конструктор `std::thread` принимает функцию (или любой Callable объект), которую поток начнет выполнять сразу после создания.

## Запуск и передача аргументов

При создании потока аргументы передаются в конструктор. Здесь действует семантика, схожая с `std::bind`: аргументы копируются во внутреннее хранилище потока (обычно в кучу или стек нового потока).

```
1  #include <thread>
2  #include <iostream>
3  #include <string>
4
5  void worker(int id, const std::string& data) {
6      // std::this_thread::get_id() возвращает уникальный идентификатор
7      std::cout << "Thread " << id
8                << " (" << std::this_thread::get_id() << ") processing: "
9                << data << std::endl;
10 }
11
12 int main() {
13     int id = 1;
14     std::string message = "Hello";
15 }
```

```

16     // message будет скопирован!
17     std::thread t1(worker, id, message);
18
19     // Если нужно передать ссылку, используем std::ref / std::cref
20     std::thread t2(worker, 2, std::cref(message));
21
22     t1.join();
23     t2.join();
24     return 0;
25 }

```

### Важно!

По умолчанию аргументы копируются. Если передаваемый объект тяжелый или не копируемый (например, `std::unique_ptr`), его необходимо перемещать с помощью `std::move`, а если функция ожидает ссылку — явно оборачивать в `std::ref`.

## Проблема времени жизни: `join()` и `detach()`

После запуска потока основной поток (родитель) обязан определить судьбу дочернего потока до того, как объект `std::thread` будет уничтожен (выдет из области видимости).

- **`join()`**: Родитель блокируется и ждет завершения дочернего потока. Это точка синхронизации. После возврата из `join()` поток считается завершенным, а ресурсы ОС очищенными.
- **`detach()`**: Поток "отпускается" в свободное плавание. Связь с объектом `std::thread` разрывается. Поток продолжит работать в фоне, пока не завершится сам или пока не завершится программа.

## Ловушка деструктора `std::thread`

Одним из самых спорных решений в дизайне C++11 является поведение деструктора `std::thread`. Если на момент вызова деструктора поток все еще "присоединен" (joinable) — то есть для него не был вызван ни `join()`, ни `detach()` — программа аварийно завершается.

```

1 void dangerous_code() {
2     std::thread t[]{
3         std::this_thread::sleep_for(std::chrono::seconds(1));
4     };
5
6     // Забыли t.join()!
7     // При выходе из функции вызывается ~thread().
8     // Так как t.joinable() = true, вызывается std::terminate().
9 }

```

Это сделано намеренно. Если бы деструктор делал неявный `join()`, это могло бы привести к неочевидным зависаниям программы (деструктор ждет вечно). Если бы он делал `detach()`, поток продолжил бы обращаться к локальным переменным уже уничтоженного стека родительской функции, что привело бы к UB. Комитет стандартизации выбрал "fail fast" — немедленное падение, сигнализирующее об ошибке в логике.

## RAII для потоков: `std::jthread`

В стандарте C++20 был добавлен класс `std::jthread` (joining thread), который исправляет неудобство обычного `std::thread`. Он реализует идиому RAII: в своем деструкторе он автоматически вызывает `join()` (если поток еще работает), либо ничего не делает (если поток уже завершен).

```
1  #include <thread>
2
3  void safe_code() {
4      // std::jthread автоматически присоединится при выходе из scope
5      std::jthread t([]{
6          // ... работа ...
7      });
8  } // Здесь будет вызван t.join()
```

Кроме того, `std::jthread` поддерживает механизм токенов отмены (`std::stop_token`), позволяя вежливо попросить поток завершиться, но это тема отдельного разговора.

## Стоимость создания потока

Частой ошибкой новичков является отношение к потокам как к "легковесным" сущностям (по аналогии с горутинами в Go или файберами). В C++ поток `std::thread` обычно отображается 1:1 на поток операционной системы (kernel thread).

Создание потока — дорогая операция. Рассмотрим, что происходит "под капотом" в Linux (системный вызов `clone`):

1. **Аллокация стека:** По умолчанию размер стека составляет около 2-8 МБ (зависит от `ulimit -s`). Даже если физическая память выделяется лениво (по мере обращения к страницам), это все равно нагрузка на менеджер виртуальной памяти (VMA).
2. **Структуры ядра:** Создается запись в таблице процессов (`task_struct`), выделяются дескрипторы.
3. **Планировщик:** Новый поток нужно добавить в очередь планировщика.

Время создания потока на современном железе измеряется **десятками микросекунд** (на порядок больше, чем просто вызов функции).

## Бенчмарк: Создание потоков в цикле

Рассмотрим пример неэффективного кода, который создает новый поток для каждой микрозадачи (вычисление корня).

```
1  // Плохой паттерн: создание потока на каждую итерацию
2  void SlowSum() {
3      double sum = 0;
4      for (int i = 0; i < 100000; ++i) {
5          std::thread t([i, &sum] {
6              sum += std::sqrt(i);
7          });
8      }
```

```
8         t.join(); // Ждем завершения сразу же
9     }
10 }
```

Запуск такого кода через профилировщик (perf или strace -T) покажет, что программа проводит больше времени в системных вызовах clone и mmap (выделение памяти под стек), чем в полезных вычислениях.

### Важно!

Если ваша задача выполняется быстрее, чем время создания потока (10-50 мкс), распараллеливание через создание новых потоков **замедлит** программу.

## Проблема Oversubscription

Другая крайность — запуск слишком большого количества потоков одновременно. Если количество активных (выполняющих работу) потоков значительно превышает количество физических ядер (std::thread::hardware\_concurrency()), возникает явление **Oversubscription**.

Операционная система вынуждена часто переключать контекст между потоками, чтобы дать каждому квант времени. Как мы обсуждали в Главе 1, переключение контекста портит кэши. В предельном случае (Context Switch Storm) процессор тратит почти все время на переключение задач, а не на их выполнение.

## Ловушка std::async

В C++11 была предпринята попытка сделать высокоуровневую абстракцию для асинхронных вычислений — std::async. Она возвращает объект std::future, через который можно получить результат.

Однако реализации стандартной библиотеки (в частности, libstdc++ в GCC и Clang) часто реализуют политику запуска std::launch::async самым примитивным способом: **всегда создается новый поток**.

```
1 // Опасно: может положить систему
2 std::vector<std::future<double>> futures;
3 for (int i = 0; i < 100000; ++i) {
4     // Каждая итерация порождает новый поток ОС!
5     futures.push_back(std::async(std::launch::async, [i]{
6         return std::sqrt(i);
7     }));
8 }
```

Этот код может привести к исчерпанию ресурсов (ошибка std::system\_error или Resource temporarily unavailable), так как он попытается создать 100 000 потоков одновременно. В MSVC есть внутренний глобальный тред-пул, но полагаться на это в кроссплатформенном коде нельзя.

## Решение: Thread Pool

Чтобы избежать накладных расходов на создание потоков и проблем с oversubscription, в промышленном коде используется паттерн **Thread Pool** (Пул потоков).

Идея проста:

1. При старте программы создается фиксированное количество потоков (обычно равное числу ядер).
2. Потоки крутятся в бесконечном цикле, ожидая задачи из потокобезопасной очереди.
3. Основной поток не создает новые потоки, а кладет задачи (функции/лямбды) в эту очередь.

Это позволяет платить цену за создание потоков только один раз — при инициализации. Мы разберем реализацию очереди и пула в следующих главах, когда изучим примитивы синхронизации.

### Резюме раздела

- Используйте `std::jthread` (C++20) или убедитесь, что всегда вызываете `join()` для `std::thread`.
- Создание потока — тяжелая операция (аллокация стека, `syscall`). Не создавайте потоки в горячих циклах.
- Избегайте `std::async` с политикой `launch::async` в циклах, так как это неявное создание потоков.
- Оптимальное число рабочих потоков для вычислений равно числу ядер процессора.

## Глава 3

# Гонки данных и модель памяти

Введение многопоточности в программу фундаментально меняет модель её исполнения. Если в однопоточном коде поведение строго детерминировано (инструкции выполняются последовательно), то в многопоточном окружении порядок исполнения инструкций разных потоков относительно друг друга не определен. Это приводит к двум классам проблем: состоянию гонки (Race Condition) и гонке данных (Data Race). Хотя эти термины часто используют как синонимы, в C++ между ними существует принципиальная разница: первое — это логическая ошибка, второе — неопределенное поведение (Undefined Behavior).

## Race Condition vs Data Race

Для начала разграничим понятия.

### Race Condition (Состояние гонки)

Ситуация, когда результат работы программы (или её состояние) зависит от того, в каком порядке планировщик ОС решит исполнять потоки. Это семантическая ошибка. Программа технически корректна с точки зрения языка, но делает не то, что ожидал программист.

### Data Race (Гонка данных)

Ситуация, когда два и более потока одновременно обращаются к одной и той же ячейке памяти, при этом:

1. Минимум один из потоков выполняет запись (модификацию).
2. Между обращениями нет синхронизации (atomic, mutex).

В стандарте C++ это строго классифицируется как **Undefined Behavior**.

## Пример 1: Вывод в консоль (Race Condition)

Рассмотрим классический пример, с которого начинают изучение потоков.

```
1 #include <thread>
2 #include <iostream>
3
```

```
4 void print_hello(int id) {
5     // Оператор << вызывается цепочкой
6     std::cout << "Thread " << id << " says hello!" << std::endl;
7 }
8
9 int main() {
10    std::thread t1(print_hello, 1);
11    std::thread t2(print_hello, 2);
12    t1.join();
13    t2.join();
14 }
```

При запуске этой программы вы можете получить корректный вывод, а можете увидеть что-то вроде: Thread 1 says Thread 2 says hello!hello!

Это **Race Condition**. Стандартная библиотека C++ гарантирует, что глобальные объекты вроде `std::cout` потокобезопасны в том смысле, что их внутреннее состояние не будет разрушено (программа не упадет). Однако атомарность гарантируется только на уровне одного вызова функции (одного оператора `<<`).

Выражение `std::cout << "A" << "B"` — это два последовательных вызова функции. Планировщик может прервать первый поток после вывода "A", переключиться на второй, который выведет свои данные, и только потом вернуться к первому.

## Решение: `std::osyncstream`

В C++20 появился синхронизируемый поток вывода `std::osyncstream` (заголовок `<syncstream>`). Он накапливает вывод во внутреннем буфере и атомарно сбрасывает его в выходной поток при уничтожении или вызове `emit()`.

```
1 #include <syncstream>
2
3 void safe_print(int id) {
4     std::osyncstream(std::cout) << "Thread " << id << " says hello!\n";
5     // При выходе из выражения временный объект osyncstream разрушается
6     // и атомарно пишет в cout.
7 }
```

## Пример 2: Инкремент (Data Race)

Теперь рассмотрим ситуацию, которая является фатальной ошибкой. Пусть у нас есть глобальный счетчик, который инкрементируют несколько потоков.

```
1 #include <thread>
2 #include <vector>
3 #include <iostream>
4
5 int counter = 0; // Разделяемый ресурс
6
7 void worker() {
```

```
8     for (int i = 0; i < 100000; ++i) {
9         counter++; // Data Race!
10    }
11 }
12
13 int main() {
14     std::thread t1(worker);
15     std::thread t2(worker);
16     t1.join();
17     t2.join();
18     std::cout << "Result: " << counter << std::endl;
19 }
```

Ожидаемый результат: 200 000. Реальный результат: случайное число, например, 142 583. Или 200 000 (если повезет).

## Анатомия гонки

Операция инкремента `counter++` не является атомарной. На уровне процессора (x86) она раскладывается на три этапа (Read-Modify-Write):

1. `MOV EAX, [addr]` — Загрузить значение из памяти в регистр.
2. `INC EAX` — Увеличить значение в регистре.
3. `MOV [addr], EAX` — Записать значение обратно в память.

Если два потока выполняют эти инструкции одновременно, происходит следующее:

1. Поток 1 читает 0 в свой регистр.
2. Поток 2 читает 0 в свой регистр (так как Поток 1 еще не записал результат).
3. Поток 1 увеличивает 0 до 1 и пишет 1 в память.
4. Поток 2 увеличивает 0 до 1 и пишет 1 в память.

Два инкремента превратились в один. Потеря данных произошла.

## Почему это Undefined Behavior?

Может показаться, что проблема только в "потере" инкрементов. Но стандарт C++ объявляет это UB, что дает компилятору право на агрессивные оптимизации.

Компилятор видит, что переменная `counter` не является атомарной. Согласно модели памяти C++ (которая подразумевает `single-threaded execution`, если нет явной синхронизации), он может предположить, что переменную никто другой не меняет. Он может загрузить `counter` в регистр в начале цикла, прибавить к нему 100 000, и записать обратно только в самом конце. Или вообще выкинуть промежуточные записи.

В многопоточной среде такие оптимизации ломают логику программы полностью. Именно поэтому использование обычных переменных (`int`, `bool`) для синхронизации без мьютексов или атомиков — это всегда ошибка.

## ThreadSanitizer (TSan)

Поиск гонок данных вручную крайне сложен, так как они проявляются недетерминированно. Ошибка может не воспроизводиться на машине разработчика, но "стрелять" в продакшене под нагрузкой.

Для автоматического детектирования используется инструмент **ThreadSanitizer (TSan)**. Это модуль для компиляторов Clang и GCC.

Для использования нужно скомпилировать код с флагом: `-fsanitize=thread -g`

TSan инструментирует код, отслеживая все обращения к памяти во время исполнения. Если он видит два обращения к одному адресу из разных потоков, одно из которых — запись, и между ними не было "happens-before" связи (захвата мьютекса, join потока), он выдает подробный отчет об ошибке.

### Важно!

TSan замедляет выполнение программы в 5-15 раз и потребляет много памяти. Используйте его на этапах тестирования и в CI, но не в релизной сборке для клиентов.

## Ловушка `std::vector<bool>`

Существует особый случай гонки данных, который неочевиден для многих разработчиков. Это использование `std::vector<bool>` в многопоточной среде.

```
1 std::vector<bool> flags(16, false);
2
3 // Поток 1
4 std::thread t1([&]{
5     flags[0] = true;
6 });
7
8 // Поток 2
9 std::thread t2([&]{
10     flags[1] = true;
11 });
```

Казалось бы, потоки обращаются к **разным** элементам вектора (индексы 0 и 1). Логически пересечения нет. Однако, согласно спецификации, `std::vector<bool>` — это псевдоконтейнер, который специализируется для экономии памяти. Он упаковывает 8 булевых значений в один байт.

- Процессор не умеет адресовать отдельные биты. Минимальная адресуемая единица — байт.
- Чтобы записать `true` в 0-й бит, процессор должен: прочитать весь байт, наложить битовую маску (OR), записать байт обратно.
- Поток 1 читает байт, меняет 0-й бит.
- Поток 2 одновременно читает **тот же самый байт**, меняет 1-й бит.

В результате возникает классический Data Race на уровне байта. Изменения одного из потоков могут быть потеряны, или байт превратится в мусор.

### Резюме раздела

---

- **Race Condition** — ошибка логики (порядок действий), **Data Race** — ошибка доступа к памяти (UB).
- Любая запись в общую память без синхронизации — потенциальный Data Race.
- Используйте **ThreadSanitizer** для поиска гонок.
- Остерегайтесь `std::vector<bool>`: разные индексы не гарантируют безопасность потоков, так как они могут делить один байт памяти.

## Глава 4

# Синхронизация: Мьютексы и Взаимные блокировки

Рассмотрев опасности гонок данных, мы приходим к необходимости механизмов защиты разделяемых ресурсов. Самым базовым и распространенным примитивом синхронизации в C++ (и в большинстве других языков) является мьютекс (Mutual Exclusion — взаимное исключение). В этой главе мы изучим не только API `std::mutex`, но и правильные паттерны его использования (RAII), а также разберем одну из самых сложных проблем многопоточности — взаимную блокировку (Deadlock) и способы борьбы с ней.

## `std::mutex` и Критическая секция

Класс `std::mutex` (заголовок `<mutex>`) предоставляет два основных метода:

- `lock()`: Поток пытается захватить мьютекс. Если мьютекс свободен, поток захватывает его и продолжает выполнение. Если мьютекс уже захвачен другим потоком, текущий поток **блокируется** (уходит в состояние ожидания, *sleeping/waiting*), пока мьютекс не освободится.
- `unlock()`: Поток освобождает мьютекс, позволяя одному из ожидающих потоков захватить его.

Участок кода между вызовами `lock()` и `unlock()` называется **критической секцией**. В любой момент времени внутри критической секции может находиться только один поток.

## Почему "сырые" `lock()` и `unlock()` опасны

Использование методов `lock()` и `unlock()` напрямую крайне не рекомендуется в современном C++. Рассмотрим пример:

```
1 std::mutex mtx;
2 int shared_data = 0;
3
4 void bad_practice() {
5     mtx.lock();
6     // ... работа с shared_data ...
7
8     if (some_error) {
```

```
9      // Забыли unlock()! Мьютекс остался захвачен навсегда.  
10     return;  
11 }  
12  
13 // Если здесь вылетит исключение, unlock() тоже не вызовется.  
14 func_that_may_throw();  
15  
16 mtx.unlock();  
17 }
```

Если поток завершится (через `return` или исключение), не вызвав `unlock()`, мьютекс останется в заблокированном состоянии. Все остальные потоки, пытающиеся его захватить, зависнут навечно (Deadlock).

## RAII: `lock_guard` и `unique_lock`

Для решения проблемы управления временем жизни блокировки используется идиома RAII (Resource Acquisition Is Initialization). Стандартная библиотека предоставляет обертки, которые захватывают мьютекс в конструкторе и освобождают в деструкторе.

### `std::lock_guard`

Самая простая и легкая обертка. Захватывает мьютекс при создании, освобождает при выходе из области видимости. Не копируется, не перемещается.

```
1 void good_practice() {  
2     // Конструктор вызывает mtx.lock()  
3     std::lock_guard<std::mutex> guard(mtx);  
4  
5     // ... безопасная работа ...  
6  
7     // При любом выходе (return, exception) вызовется деструктор guard,  
8     // который вызовет mtx.unlock().  
9 }
```

Начиная с C++17, шаблонный тип можно не указывать (CTAD): `std::lock_guard guard(mtx);`.

### `std::unique_lock`

Более мощная и тяжелая обертка. В отличие от `lock_guard`, она позволяет:

- Отложить блокировку (стратегия `std::defer_lock`).
- Временно разблокировать мьютекс (`unlock()`) и снова заблокировать (`lock()`) внутри одного скоупа.
- Передавать владение блокировкой (перемещаемый тип).

`std::unique_lock` необходим при работе с условными переменными (`std::condition_variable`), так как они требуют возможности атомарно отпустить мьютекс при ожидании.

## Shared Mutex (Reader-Writer Lock)

Часто возникает ситуация, когда данные редко меняются, но часто читаются (например, конфигурация, кэш DNS). Использование обычного `std::mutex` заставит читателей выстраиваться в очередь, хотя они могли бы читать данные параллельно, не мешая друг другу.

Для этого сценария в C++17 был добавлен `std::shared_mutex` (заголовок `<shared_mutex>`). Он поддерживает два режима захвата:

1. **Эксклюзивный (Writer):** Аналог обычного `lock()`. Блокирует всех (и читателей, и писателей). Используется для изменения данных.
2. **Разделяемый (Reader):** Метод `lock_shared()`. Позволяет нескольким потокам одновременно владеть мьютексом в режиме чтения. Блокирует только писателей.

Для удобства использования существуют соответствующие RAII-обертки:

- `std::lock_guard<std::shared_mutex>` или `std::unique_lock` — для эксклюзивного захвата (писатель).
- `std::shared_lock<std::shared_mutex>` — для разделяемого захвата (читатель).

```

1  #include <shared_mutex>
2
3  class ThreadSafeConfig {
4      std::map<std::string, std::string> settings;
5      mutable std::shared_mutex mtx; // mutable, чтобы использовать в const методах
6
7  public:
8      std::string get(const std::string& key) const {
9          // Множество потоков могут вызывать get() одновременно
10         std::shared_lock lock(mtx);
11         auto it = settings.find(key);
12         return (it != settings.end()) ? it->second : "";
13     }
14
15     void set(const std::string& key, const std::string& value) {
16         // Только один поток может писать, блокируя всех читателей
17         std::lock_guard lock(mtx);
18         settings[key] = value;
19     }
20 };

```

## Deadlock (Взаимная блокировка)

Использование мьютексов вводит новый класс ошибок — взаимные блокировки. Классический пример — транзакция перевода денег между двумя банковскими счетами. Чтобы операция была атомарной, нужно заблокировать оба аккаунта.

### Проблема порядка блокировки

Рассмотрим наивную реализацию функции `transfer`:

```

1  struct Account {
2      std::mutex m;
3      int balance;
4  };
5
6  void transfer(Account& from, Account& to, int amount) {
7      std::lock_guard<std::mutex> lock1(from.m);
8      // Имитация задержки, увеличивающая вероятность Deadlock
9      std::this_thread::sleep_for(std::chrono::milliseconds(1));
10     std::lock_guard<std::mutex> lock2(to.m);
11
12     from.balance -= amount;
13     to.balance += amount;
14 }

```

Представим, что два потока одновременно выполняют встречные переводы:

- **Поток 1:** `transfer(AccA, AccB, 100)`
- **Поток 2:** `transfer(AccB, AccA, 50)`

Сценарий катастрофы:

1. Поток 1 захватывает мьютекс AccA.
2. Поток 2 захватывает мьютекс AccB.
3. Поток 1 пытается захватить AccB и засыпает, так как он занят Поток 2.
4. Поток 2 пытается захватить AccA и засыпает, так как он занят Поток 1.

Оба потока ждут друг друга. Программа зависает.

## Решение 1: Иерархия блокировок

Можно установить правило: всегда захватывать мьютексы в определенном глобальном порядке. Например, по адресу памяти объекта `Account` (от меньшего к большему) или по уникальному ID аккаунта.

```

1  void safe_transfer_manual(Account& from, Account& to, int amount) {
2      if (&from < &to) {
3          std::lock_guard<std::mutex> lock1(from.m);
4          std::lock_guard<std::mutex> lock2(to.m);
5      } else {
6          std::lock_guard<std::mutex> lock1(to.m); // Сначала меньший адрес
7          std::lock_guard<std::mutex> lock2(from.m);
8      }
9      // ...
10 }

```

Это работает, но сложно поддерживать в больших системах.

## Решение 2: `std::lock` и `std::scoped_lock`

Стандартная библиотека C++ предоставляет алгоритмы для безопасного захвата нескольких мьютексов без риска дедлока (обычно внутри используется тот же алгоритм try-and-

backoff или сортировка по адресам).

В C++11 есть функция `std::lock(m1, m2, ...)`, которая блокирует мьютексы, но не управляет их освобождением (нужно передавать их в `lock_guard` с флагом `std::adopt_lock`).

В C++17 появился идеальный инструмент — **`std::scoped_lock`**. Это вариативный шаблон, который принимает произвольное количество мьютексов, захватывает их безопасным алгоритмом в конструкторе и освобождает в деструкторе.

```
1 void safe_transfer_cpp17(Account& from, Account& to, int amount) {  
2     // Блокирует оба мьютекса безопасно. Порядок аргументов не важен.  
3     std::scoped_lock lock(from.m, to.m);  
4  
5     from.balance -= amount;  
6     to.balance += amount;  
7 }
```

### Резюме раздела

- Никогда не вызывайте `lock()` и `unlock()` вручную. Используйте RAII: `std::lock_guard` или `std::unique_lock`.
- Используйте `std::shared_mutex` и `std::shared_lock` для сценариев "много читателей, один писатель".
- Захват нескольких мьютексов одновременно — риск Deadlock.
- Для захвата нескольких мьютексов всегда используйте `std::scoped_lock` (C++17).

## Глава 5

# Атомики и низкоуровневая синхронизация

Мьютексы, рассмотренные нами ранее, являются механизмами синхронизации уровня операционной системы. Когда поток не может захватить мьютекс, он "усыпляется" ядром ОС, что влечет за собой накладные расходы на системные вызовы и переключение контекста. Однако для простейших операций, таких как инкремент счетчика или установка флага, эти расходы могут быть неприемлемо высоки.

В этой главе мы спустимся на уровень ниже — к атомарным операциям и инструкциям процессора. Мы разберем, как работает `std::atomic`, развеем опасный миф о `volatile` и напишем собственный примитив синхронизации — спинлок (Spinlock).

## Миф о `volatile`

Прежде чем переходить к правильным инструментам, необходимо разобрать одну из самых живучих ошибок в C++ программировании. Многие разработчики, пришедшие из мира микроконтроллеров или старых версий Java, ошибочно полагают, что ключевое слово `volatile` обеспечивает потокобезопасность.

### Важно!

**`volatile` в C++ не имеет отношения к многопоточности!** Оно не гарантирует атомарность. Оно не создает барьеров памяти. Оно не предотвращает гонки данных.

Ключевое слово `volatile` сообщает компилятору лишь одно: "значение этой переменной может измениться извне, поэтому не кэшируй его в регистрах". Это необходимо для работы с **ММО** (Memory-Mapped I/O) — когда адрес в памяти на самом деле является портом ввода-вывода устройства (например, датчика температуры).

Рассмотрим пример Data Race с использованием `volatile`:

```
1 volatile int counter = 0;
2
3 void increment() {
4     // ОШИБКА: Это все еще Read-Modify-Write операция.
5     // volatile лишь заставит процессор каждый раз читать из памяти,
6     // но не запрещает другому потоку вклинуться между чтением и записью.
7     counter++;
8 }
```

Если запустить этот код в несколько потоков, результат будет неверным. Единственное применение `volatile` в современном C++ — это взаимодействие с железом или обработчиками сигналов UNIX. Для потоков используйте `std::atomic`.

## std::atomic

Шаблон `std::atomic<T>` (заголовок `<atomic>`) предоставляет интерфейс для работы с данными, операции над которыми гарантированно выполняются атомарно.

### Базовые операции: load и store

Атомики запрещают компилятору и процессору переупорядочивать инструкции (reordering) опасным образом и гарантируют, что чтение или запись переменной произойдет целиком.

```
1 std::atomic<int> flag = 0;
2
3 void writer() {
4     flag.store(1); // Атомарная запись
5 }
6
7 void reader() {
8     // Атомарное чтение. Гарантируется, что мы не прочитаем
9     // "частично записанное" значение (torn read).
10    int val = flag.load();
11 }
```

### Read-Modify-Write (RMW)

Самая мощная возможность атомиков — это операции, которые читают, изменяют и записывают значение как единое неделимое действие.

- `fetch_add(val)`: прибавляет значение, возвращает старое.
- `fetch_sub(val)`: вычитает значение.
- `exchange(val)`: записывает новое значение, возвращает старое.

```
1 std::atomic<int> counter = 0;
2
3 void worker() {
4     // Эквивалентно fetch_add(1).
5     // На x86 компилируется в инструкцию LOCK XADD.
6     counter++;
7 }
```

Важно понимать разницу между атомарностью самой переменной и последовательности действий.

### На заметку

`atomic_val = atomic_val + 1` — **НЕ** атомарно! Это последовательность: (1) атомарное чтение, (2) локальное сложение, (3) атомарная запись. Между (1) и (3) может вклиниться другой поток. Для атомарного инкремента используйте только `fetch_add` или оператор `++`.

## Compare-And-Swap (CAS)

Фундаментом всех lock-free структур данных (очередей, списков без мьютексов) является операция Compare-And-Swap. В C++ она представлена методами `compare_exchange_strong` и `compare_exchange_weak`.

Логика CAS следующая: "Проверь, равно ли текущее значение `expected`. Если да, запиши `desired` и верни `true`. Если нет, обнови `expected` текущим значением и верни `false`".

```
1 std::atomic<int> current_head;
2
3 void push(int new_val) {
4     int old_head = current_head.load();
5     // Пытаемся обновить head, только если он не изменился с момента чтения.
6     // Если изменился (кто-то другой успел запустить), повторяем цикл.
7     while (!current_head.compare_exchange_weak(old_head, new_val)) {
8         // Тело цикла может быть пустым, old_head обновляется автоматически
9         // ↪ внутри CAS
10    }
```

- **weak**: Может вернуть `false`, даже если значение равно ожидаемому (spurious failure). Это особенность архитектур типа ARM и PowerPC. Используется в циклах.
- **strong**: Гарантирует успех, если значения равны. Используется, когда цикл неудобен, но стоит дороже на некоторых платформах.

## Реализация Spinlock

Используя атомик, мы можем реализовать собственный примитив синхронизации — спинлок. В отличие от мьютекса, спинлок не усыпляет поток, а заставляет его крутиться в цикле ("spin"), ожидая освобождения флага.

Для реализации идеально подходит `std::atomic_flag`. Это единственный тип, для которого стандарт гарантирует lock-free реализацию на любой платформе.

```
1 #include <atomic>
2 #include <thread>
3
4 class Spinlock {
5     // ATOMIC_FLAG_INIT устарело в C++20, конструктор по умолчанию ставит в
6     // ↪ false (clear)
7     std::atomic_flag flag;
```

```
7
8 public:
9     void lock() {
10         // test_and_set атомарно ставит флаг в true и возвращает ПРЕДЫДУЩЕЕ
11         // значение.
12         // Если вернулось true -> значит флаг уже был занят -> продолжаем
13         // крутиться.
14         // Если вернулось false -> мы успешно захватили спинлок.
15         while (flag.test_and_set(std::memory_order_acquire)) {
16             // Hint для процессора, что мы в цикле ожидания (снижает
17             // энергопотребление)
18             // В C++20: std::atomic::wait
19             #if defined(__x86_64__) || defined(_M_X64)
20                 _mm_pause();
21             #endif
22         }
23     }
24
25     void unlock() {
26         flag.clear(std::memory_order_release);
27     }
28 };
```

## Spinlock vs Mutex: Цена решения

Когда стоит использовать спинлок, а когда мьютекс?

### Spinlock:

- **Плюсы:** Не требует системных вызовов. Захват происходит за десятки наносекунд (если свободен). Не сбрасывает кэш процессора (нет переключения контекста).
- **Минусы:** Если спинлок занят долго, ожидающий поток сжигает 100% процессорного времени впустую (busy-wait).
- **Применение:** Критические секции длиной в несколько инструкций (например, инкремент счетчика или перестановка указателя).

### Mutex:

- **Плюсы:** Если ресурс занят, поток уступает ядро другим задачам. Экономит энергию.
- **Минусы:** Захват/освобождение требуют обращения к ядру ОС, что долго.
- **Применение:** Длительные операции (ввод-вывод, сложная логика).

### Важно!

Линус Торвалдс и многие эксперты считают, что использование спинлоков в userspace (пользовательском пространстве) вредно. Причина — **Priority Inversion** и планировщик ОС. Если поток, захвативший спинлок, будет вытеснен планировщиком (кончился квант времени), все остальные потоки будут крутиться впустую целый квант, ожидая его возвращения. Современные реализации `std::mutex` (futex в Linux) гибридные: они немного "крутятся" в userspace перед тем, как уйти в сон.

## Модель памяти (Memory Ordering)

В примерах выше мы видели странные аргументы `std::memory_order_acquire/release`. Это управление моделью памяти. По умолчанию C++ использует **Sequential Consistency** (`seq_cst`). Это самая строгая модель: она гарантирует, что все потоки видят все изменения в одном и том же глобальном порядке. Это безопасно, но может быть медленно, так как требует блокировки шины или сброса буферов записи процессора.

Более слабые модели (`acquire, release, relaxed`) позволяют процессору переупорядочивать инструкции ради производительности, сохраняя лишь необходимые гарантии видимости. Программирование с `weak ordering` — это высший пилотаж C++, где любая ошибка приводит к неуловимым багам. До тех пор, пока профайлер не покажет, что узкое место именно в синхронизации кэшей, используйте дефолтный `seq_cst`.

### Резюме раздела

- **volatile** не для потоков. Никогда не используйте его для синхронизации.
- **std::atomic** — единственный корректный способ работы с общими простыми типами без мьютексов.
- **RMW операции** (`fetch_add, exchange`) позволяют менять данные безопасно.
- **Spinlock** эффективен для сверхкоротких блокировок, но опасен в общем случае из-за сжигания CPU.
- **CAS (Compare-And-Swap)** — основа lock-free алгоритмов.

## Глава 6

# Оптимизация под железо: False Sharing

В предыдущих главах мы добились логической корректности многопоточных программ, используя мьютексы и атомики. Мы устранили гонки данных (Data Race) и неопределенное поведение. Однако в высокопроизводительных системах логической корректности недостаточно.

Существует класс проблем, при которых программа работает абсолютно правильно, не имеет блокировок (lock-free), использует все ядра процессора на 100%, но при этом выполняется в десятки раз медленнее, чем однопоточная версия. Причина кроется в физической организации подсистемы памяти, а именно в явлении, называемом **False Sharing** (Ложное разделение).

## Физика кэш-линий

Процессор никогда не читает из оперативной памяти (RAM) отдельные байты. Это было бы крайне неэффективно с точки зрения пропускной способности шины. Обмен данными между оперативной памятью и кэшами процессора (L1/L2/L3) происходит блоками фиксированного размера, которые называются **кэш-линиями** (Cache Lines).

### На заметку

На большинстве современных архитектур (x86\_64, многие ARM) размер кэш-линии составляет **64 байта**.

Это означает, что если вы читаете переменную типа `int` (4 байта) по адресу `0x1000`, процессор загрузит в кэш весь блок от `0x1000` до `0x103F`. Вместе с нужной вам переменной в кэш попадут и соседние данные ("хвост" длиной 60 байт).

## Протоколы когерентности (MESI)

В многоядерных системах каждый процессор (ядро) имеет свой собственный L1 и L2 кэш. Если два ядра закэшировали одну и ту же область памяти, возникает проблема синхронизации (когерентности). Если одно ядро изменит данные у себя в L1, другое ядро должно узнать об этом, иначе оно будет работать с устаревшим значением.

Для решения этой задачи существуют аппаратные протоколы когерентности, такие как **MESI** (Modified, Exclusive, Shared, Invalid). Упрощенно логика выглядит так:

1. Чтобы ядро могло записать данные в кэш-линию, оно должно получить её в эксклюзивное владение (состояние *Exclusive* или *Modified*).
2. Если эта линия находится в кэшах других ядер (состояние *Shared*), то перед записью инициатор посылает сигнал инвалидации (*Invalidate*).
3. Другие ядра помечают свои копии линии как невалидные (*Invalid*) и выбрасывают их.
4. При следующей попытке чтения эти ядра вынуждены заново загружать линию из памяти (или из кэша пишущего ядра), что очень дорого.

## Феномен False Sharing

Ложное разделение возникает, когда два или более потока одновременно модифицируют **логически разные** переменные, которые случайно оказались на **одном и том же** физическом носителе — в одной кэш-линии.

### Сценарий "Пинг-понг"

Представим структуру, содержащую два атомарных счетчика:

```
1 struct SharedData {  
2     std::atomic<int> a; // 4 байта  
3     std::atomic<int> b; // 4 байта  
4 };  
5 // sizeof(SharedData) = 8 байт. Оба поля лежат в одной кэш-линии.
```

Пусть Поток 1 (Ядро 0) инкрементирует а, а Поток 2 (Ядро 1) инкрементирует b.

1. **Ядро 0** хочет записать в а. Оно запрашивает линию эксклюзивно.
2. **Ядро 1** вынуждено сбросить свою копию линии (*Invalidate*), даже если оно работало только с b.
3. Ядро 0 меняет а. Линия в состоянии *Modified* на Ядре 0.
4. **Ядро 1** хочет записать в b. Промах кэша (L1 Miss). Оно запрашивает линию у Ядра 0.
5. Ядро 0 сбрасывает свои изменения в общий кэш/память (*Write Back*) и отдает владение. Линия на Ядре 0 становится *Invalid*.
6. Ядро 1 меняет b.

Происходит "пинг-понг" кэш-линии между ядрами. Вместо того чтобы работать на скорости L1 кэша (1 нс), ядра ждут синхронизации через шину (десятки нс). Производительность может упасть в 10-50 раз по сравнению с теоретическим максимумом.

Термин "ложное" (*False*) означает, что разделения данных на логическом уровне нет (потоки не трогают данные друг друга), но процессор этого не знает — он видит разделение на уровне 64-байтного блока.

## Демонстрация и диагностика

Рассмотрим пример кода, демонстрирующий этот эффект. Мы запустим несколько потоков, каждый из которых инкрементирует свой собственный счетчик в массиве.

```
1  #include <thread>
2  #include <vector>
3  #include <atomic>
4  #include <new> // для hardware_destructive_interference_size
5
6  // Плохая структура: данные упакованы плотно
7  struct PackedCounter {
8      std::atomic<long> value{0};
9  };
10
11 // Хорошая структура: принудительное выравнивание
12 struct AlignedCounter {
13     alignas(64) std::atomic<long> value{0};
14 };
15
16 template <typename CounterType>
17 void benchmark() {
18     std::vector<CounterType> counters(4);
19     std::vector<std::thread> threads;
20
21     // Каждый поток работает строго со своим индексом!
22     // Логического пересечения нет.
23     for (int i = 0; i < 4; ++i) {
24         threads.emplace_back([&counters, i] {
25             for (int j = 0; j < 100'000'000; ++j) {
26                 counters[i].value.fetch_add(1, std::memory_order_relaxed);
27             }
28         });
29     }
30
31     for (auto& t : threads) t.join();
32 }
```

## Анализ памяти

В случае `PackedCounter`: `sizeof(atomic<long>)` равен 8 байтам. Четыре счетчика займут 32 байта. Они гарантированно поместятся в одну 64-байтную кэш-линию. Четыре ядра будут драться за право владения этой единственной линией.

В случае `AlignedCounter`: спецификатор `alignas(64)` заставляет компилятор размещать каждый объект по адресу, кратному 64. Размер структуры станет 64 байта (8 байт полезных данных + 56 байт padding). Каждый счетчик гарантированно окажется в своей уникальной кэш-линии.

### Важно!

Профилирование такого кода (например, через `perf` с2с в Linux) покажет огромное количество событий **Hitm** (Hit Modified) — ситуаций, когда ядро запрашивает данные, которые были модифицированы в кэше другого ядра.

## Решение: Padding и alignas

Единственный способ борьбы с False Sharing — разнести часто изменяемые данные (hot spots) по разным кэш-линиям. Это делается путем добавления "мусорных" данных (padding) между полезными переменными.

В старом C++ это делалось вручную добавлением массивов `char pad[64]`. В современном C++ (с C++11) используется спецификатор выравнивания `alignas`.

```
1 // Гарантируем, что атомики не попадут в одну линию
2 struct SafeMetrics {
3     alignas(64) std::atomic<int> success_count;
4     alignas(64) std::atomic<int> error_count;
5 };
```

Это увеличивает потребление оперативной памяти (trade-off: память в обмен на скорость процессора), но в контексте глобальных счетчиков или структур синхронизации это ничтожная плата за многократный прирост производительности.

## Проблема переносимости и ABI

Возникает вопрос: почему мы пишем `alignas(64)`? Откуда взялось число 64?

Хотя 64 байта — стандарт де-факто для x86\_64 (Intel/AMD), другие архитектуры могут иметь другие размеры линии (например, 128 байт на некоторых PowerPC или Apple Silicon). Если мы скомпилируем код с `alignas(64)` для процессора с линией 128 байт, False Sharing вернется, так как два 64-байтных блока попадут в одну 128-байтную линию.

В C++17 в заголовок `<new>` были добавлены константы:

- `std::hardware_destructive_interference_size`: минимальный отступ, гарантирующий отсутствие False Sharing (размер кэш-линии).
- `std::hardware_constructive_interference_size`: максимальный размер непрерывных данных, которые гарантированно влезут в одну линию (для оптимизации локальности).

## Дилемма ABI

К сожалению, использование этих констант сопряжено с проблемами ABI (Application Binary Interface). Если библиотека скомпилирована с одним значением этой константы (на старом компиляторе/железе), а приложение с другим, размеры структур не совпадут, что приведет к краху программы.

Из-за этого, например, компилятор GCC долгое время выдавал предупреждение при использовании этих констант, а в некоторых стандартных библиотеках они реализованы просто как жестко заданные значения.

### Резюме раздела

---

- Процессор обменивается данными кэш-линиями (обычно 64 байта).
- **False Sharing** возникает, когда разные потоки пишут в разные переменные, лежащие в одной кэш-линии.
- Симптомом является резкое падение производительности при высокой нагрузке на CPU без видимых блокировок.
- Решение — использование `alignas(64)` (или больше) для разнесения "горячих" данных в памяти.
- Это классический пример трейд-оффа: мы тратим лишнюю память (padding), чтобы выиграть в скорости шины.

## Глава 7

# Практикум: Управление памятью и Smart Pointers в многопоточности

Мы завершаем наш цикл, объединяя знания о многопоточности с фундаментальной темой C++ — управлением памятью. Умные указатели (smart pointers) стали стандартом де-факто для владения ресурсами, но их поведение в многопоточной среде часто понимается превратно. Является ли `std::shared_ptr` потокобезопасным? Ответ "да, но..." требует детального разбора.

Кроме того, мы рассмотрим реальный кейс из разработки интерпретаторов (на примере учебного Scheme) — реализацию сборщика мусора (Garbage Collector) по алгоритму Mark and Sweep, где вопросы достижимости объектов и управления графом становятся критическими.

## Потокобезопасность `std::shared_ptr`

Вопрос "потокобезопасен ли `shared_ptr`?" — один из самых популярных на собеседованиях. Правильный ответ состоит из двух частей, так как `std::shared_ptr` управляет двумя сущностями:

1. **Control Block (Управляющий блок):** Хранит счетчик ссылок (`ref_count`) и счетчик слабых ссылок (`weak_count`).
2. **Managed Object (Управляемый объект):** Тот самый T\*, на который указывает умный указатель.

## Гарантии стандарта

Стандарт C++ дает следующие гарантии:

- **Счетчик ссылок — потокобезопасен.** Операции инкремента (копирование `shared_ptr`) и декремента (уничтожение) являются атомарными. Вы можете безопасно копировать `shared_ptr`, указывающий на один объект, в разных потоках. Гонки данных на счетчике не будет.
- **Управляемый объект — НЕ потокобезопасен.** Если два потока имеют доступ к одному объекту через разные копии `shared_ptr`, и один из них пишет в объект — нужна внешняя синхронизация (мьютекс).

- **Сам экземпляр `shared_ptr` — НЕ потокобезопасен.** Если два потока пытаются изменить **один и тот же экземпляр** `shared_ptr` (например, глобальный указатель, присваивая ему новое значение), это гонка данных.

```

1  std::shared_ptr<int> global_ptr = std::make_shared<int>(42);
2
3  void thread1() {
4      // Безопасно: создается локальная копия.
5      // Счетчик атомарно увеличивается.
6      std::shared_ptr<int> local = global_ptr;
7
8      // Опасно! Чтение объекта без мьютекса (если кто-то пишет в *local)
9      std::cout << *local;
10 }
11
12 void thread2() {
13     // ОПАСНО! Гонка данных на самом global_ptr.
14     // Присваивание изменяет внутренности глобального объекта (ptr и control
15     //   ↪ block).
16     // thread1 может прочитать "разорванное" значение.
17     global_ptr = std::make_shared<int>(100);
18 }

```

Для безопасной модификации самого экземпляра `shared_ptr` в C++20 введены `std::atomic<std::shared_ptr>`. До C++20 приходилось использовать внешние функции `std::atomic_store(&ptr, ...)`, что было неудобно и опасно.

## Thread Local Storage (TLS)

Иногда нам вообще не нужно делить данные между потоками. Напротив, нам нужно, чтобы у каждого потока была своя, уникальная копия переменной. Для этого используется спецификатор `thread_local`.

```

1  #include <random>
2
3  // Глобальный генератор случайных чисел? Плохая идея (мьютекс убьет скорость).
4  // thread_local создает генератор для каждого потока отдельно.
5  int get_random() {
6      thread_local std::mt19937 gen(std::clock());
7      std::uniform_int_distribution<int> dist(0, 100);
8      return dist(gen);
9  }

```

Переменные `thread_local`:

- Инициализируются при первом обращении к ним в потоке (lazy initialization).
- Уничтожаются при завершении потока.
- Живут в специальной области памяти (TLS segment), доступ к которой обычно осуществляется через регистр сегмента (например, `fs` на `x86_64`), что очень быстро.

Это идеальное решение для кэшей, буферов (например, строковых билдеров) и генераторов случайных чисел в многопоточных сервисах.

## Case Study: Сборщик мусора (Mark and Sweep)

В курсе мы разрабатываем интерпретатор Scheme. Языки Lisp-семейства славятся тем, что в них легко создаются циклические структуры данных (списки, ссылающиеся сами на себя, замыкания).

### Почему `shared_ptr` не подходит?

Если использовать `std::shared_ptr` для узлов графа объектов Scheme, мы столкнемся с проблемой циклических ссылок. Объект A ссылается на B, B ссылается на A. Их `ref_count` никогда не станет нулем, даже если они оба недостижимы из остальной программы. Это классическая утечка памяти.

Решение — реализовать собственный Garbage Collector (GC). Самый простой и надежный алгоритм — **Mark and Sweep** (Пометь и Удали).

### Алгоритм

Идея алгоритма базируется на понятии **достижимости** (reachability). Объект "жив", если до него можно добраться по ссылкам от "корней" (Roots).

1. **Root Set (Корневое множество):** Это объекты, которые гарантированно живы. В нашем случае это Глобальный Скоуп (Global Scope) интерпретатора и, возможно, временные переменные на стеке вычислений.
2. **Фаза Mark (Пометка):** Мы запускаем обход графа (DFS/BFS) от корней. Каждый посещенный объект помечается флагом `is_marked = true`. Если мы встречаем уже помеченный объект, останавливаемся (защита от заикливания).
3. **Фаза Sweep (Удаление):** Мы проходим по **всем** выделенным объектам в куче (нам нужен список всех аллокаций).
  - Если у объекта `is_marked = true`, мы сбрасываем флаг в `false` (готовим к следующему циклу) и оставляем его жить.
  - Если у объекта `is_marked = false`, значит он недостижим (мусор). Мы вызываем `delete` и удаляем его из списка аллокаций.

```

1  // Упрощенная модель объекта
2  struct Object {
3      bool is_marked = false;
4      std::vector<Object*> children; // Ссылки на другие объекты
5      virtual ~Object() = default;
6  };
7
8  void mark(Object* obj) {
9      if (!obj || obj->is_marked) return;
10
11     obj->is_marked = true;
12     for (auto* child : obj->children) {
13         mark(child);
14     }
15 }
16

```

```

17 void sweep(std::vector<Object*>& all_objects) {
18     auto it = all_objects.begin();
19     while (it != all_objects.end()) {
20         if ((*it)->is_marked) {
21             (*it)->is_marked = false; // Сброс для следующего раза
22             ++it;
23         } else {
24             delete *it; // Удаление мусора
25             // Удаление из списка всех объектов (swap-and-pop для вектора)
26             *it = all_objects.back();
27             all_objects.pop_back();
28             // Итератор не инкрементируем, проверяем новый элемент на этом месте
29         }
30     }
31 }

```

Этот подход решает проблему циклов: если группа объектов замкнута сама на себя, но недостижима от Global Scope, алгоритм `mark` до них не дойдет, их флаги останутся `false`, и `sweep` их удалит.

## Инструментарий: Leak Sanitizer

Для проверки корректности работы GC в задачах мы используем **Leak Sanitizer** (LSan). Это часть набора санитайзеров (как и TSan/ASan). Он запускается в конце работы программы и проверяет, остались ли в куче аллокации, которые не были освобождены.

Если ваш GC работает корректно, на выходе LSan молчит. Если вы забыли удалить циклические ссылки, LSan выдаст отчет об утечке, показав стек вызовов, где была выделена память.

### Резюме раздела

- `std::shared_ptr` гарантирует атомарность счетчика, но не безопасность данных внутри.
- Модификация одного экземпляра `shared_ptr` из разных потоков — гонка данных.
- `thread_local` позволяет создавать уникальные копии данных для потоков без мьютексов.
- Для сложных структур с циклами (графы, интерпретаторы) `shared_ptr` недостаточен. Требуется полноценный Garbage Collection (Mark and Sweep).