Курс: *Архитектура компьютера и ОС*

Лекция 10: Синхронизация в ядре: Futex и системные вызовы

Лектор: Евгений Соколов

Дата: 21.12.2025

Содержание

1 Введение в механизмы системной синхронизации

Современные многопоточные приложения требуют эффективных способов координации. Традиционные примитивы синхронизации, реализованные полностью внутри ядра операционной системы, обладают значительными накладными расходами на переключение контекста между пространством пользователя и пространством ядра. В данной главе рассматривается Fast Userspace Mutex (Futex) — фундаментальный механизм Linux, позволяющий минимизировать эти расходы, а также системные проблемы, возникающие при ветвлении многопоточных процессов.

2 Futex: Fast Userspace Mutex

Определение: Futex

Futex (Fast Userspace Mutex) — это системный механизм Linux, предназначенный для реализации эффективных блокировок. Он позволяет потокам выполнять захват и освобождение ресурсов в пространстве пользователя (userspace) без обращения к ядру, пока нет конкуренции за ресурс.

2.1 Механика работы и системные вызовы

Интерфейс Futex представлен системным вызовом `sys_futex`. Основная идея заключается в том, что поток сначала пытается изменить атомарную переменную в памяти процесса. Если попытка успешна (конкуренция отсутствует), системный вызов не требуется. Если же переменная указывает на то, что ресурс занят, поток обращается к ядру для перехода в состояние ожидания.

Две основные операции Futex:

1. `FUTEX_WAIT`: поток засыпает, если значение по указанному адресу равно ожидаемому.
2. `FUTEX_WAKE`: ядро пробуждает N потоков, ожидающих на данном адресе.

```
// Value: 0 - unlocked, 1 - locked
void lock(int *futex_addr) {
    while (__atomic_exchange_n(futex_addr, 1, __ATOMIC_ACQUIRE) == 1) {
        // Atomic check: if value is still 1, then sleep
        syscall(SYS_futex, futex_addr, FUTEX_WAIT, 1, NULL, NULL, 0);
    }
}

void unlock(int *futex_addr) {
```

```
    __atomic_store_n(futex_addr, 0, __ATOMIC_RELEASE);
    // Wake up one waiting thread
    syscall(SYS_futex, futex_addr, FUTEX_WAKE, 1, NULL, NULL, 0);
}
```

Листинг 1 – Упрощенная реализация Mutex на базе Futex

2.2 Проблема Lost Wake-up и атомарность в ядре

Одной из критических проблем при реализации блокировок является Lost Wake-up. Рассмотрим сценарий без атомарной проверки внутри ядра:

1. Поток A видит, что `value == 1`, и готовится вызвать `FUTEX_WAIT`.
2. Поток B вызывает `unlock`, устанавливает `value = 0` и вызывает `FUTEX_WAKE`.
3. Сигнал `WAKE` пропадает, так как Поток A еще не уснул.
4. Поток A вызывает `FUTEX_WAIT` и засыпает навсегда.

Механизм `FUTEX_WAIT` решает эту проблему за счет дополнительного аргумента — ожидаемого значения. Ядро Linux гарантирует, что проверка `*uaddr == val` и постановка потока в очередь ожидания выполняются атомарно относительно операций записи в эту ячейку.

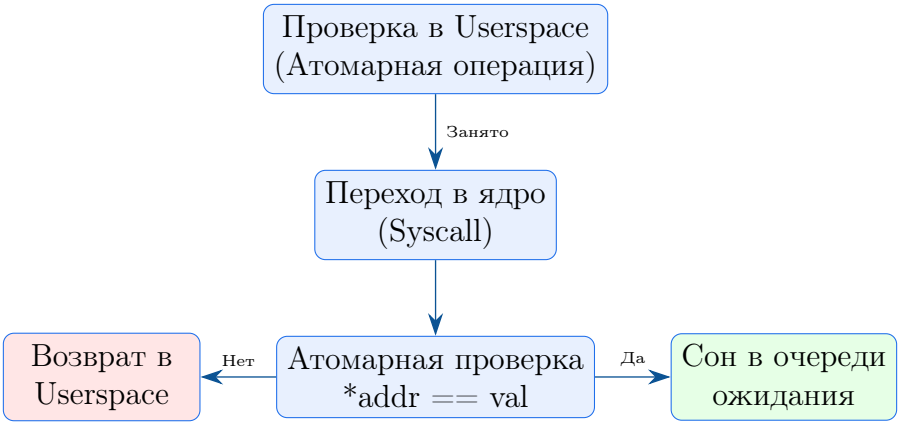

Рис. 1 – Алгоритм работы FUTEX_WAIT

3 Взаимодействие fork() и многопоточности

Системный вызов `fork()` создает копию текущего процесса, однако в контексте многопоточности его поведение обладает специфической особенностью, часто приводящей к трудноуловимым ошибкам.

Примечание

При вызове `fork()` в дочернем процессе продолжает исполнение только тот поток, который инициировал этот вызов. Все остальные потоки родительского процесса в дочернем процессе просто перестают существовать.

3.1 Проблема "мертвых"блокировок

Если в момент вызова `fork()` какой-либо поток (отличный от вызывающего) удерживал `std::mutex`, то в дочернем процессе этот мьютекс останется в захваченном состоянии

навсегда. Поскольку поток-владелец не существует в дочернем процессе, он никогда не вызовет `unlock()`. Любая попытка дочернего процесса захватить этот мьютекс приведет к `deadlock`.

Эта проблема особенно актуальна для библиотечных функций:

- **Аллокаторы памяти:** `malloc` и `free` часто используют внутренние мьютексы для защиты глобальных арен памяти.
- **Функции ввода-вывода:** `printf` и `scanf` также сериализуют доступ к потокам данных через блокировки.

3.2 Решение через pthread_atfork

Для предотвращения подобных ситуаций стандарт POSIX предоставляет функцию `pthread_atfork`. Она позволяет зарегистрировать три обработчика:

1. `prepare`: вызывается в родительском процессе перед `fork`. Обычно здесь захватываются все критические мьютексы.
2. `parent`: вызывается в родительском процессе после `fork`. Мьютексы освобождаются.
3. `child`: вызывается в дочернем процессе после `fork`. Мьютексы сбрасываются или освобождаются.

```
void prepare_locks() { pthread_mutex_lock(&global_lock); }
void parent_unlock() { pthread_mutex_unlock(&global_lock); }
void child_unlock() { pthread_mutex_unlock(&global_lock); }

// Registration in library initialization
pthread_atfork(prepare_locks, parent_unlock, child_unlock);
```

Листинг 2 – Использование pthread_atfork для безопасности аллокаторов

Итоги раздела

- **Futex** — гибридный механизм синхронизации, объединяющий быструю проверку в userspace и блокировку в ядре.
- Атомарная проверка значения в `FUTEX_WAIT` необходима для предотвращения **Lost Wake-up**.
- Вызов `fork()` в многопоточной среде копирует только вызывающий поток, что делает блокировки, захваченные другими потоками, недоступными для освобождения в дочернем процессе.
- Использование `pthread_atfork` позволяет библиотекам корректно обрабатывать состояние блокировок при ветвлении процесса.

4 Основы параллелизма: Планирование ОС и границы ускорения

Наблюдаемый параллелизм в современных вычислительных системах зачастую является абстракцией, реализуемой операционной системой. В данном разделе рассматриваются механизмы квантования времени, аппаратные ограничения сигналов и математические пределы ускорения многопоточных вычислений.

4.1 Квантование времени и аппаратная поддержка планирования

Параллелизм на одноядерных системах реализуется через механизм мультипрограммирования. Исполняемые потоки (threads) или процессы не работают непрерывно; вместо этого они разделяют процессорное время, сменяя друг друга через короткие промежутки.

Определение: Квант времени (Time Quantum)

Квант времени — фиксированный интервал времени (обычно от 1 до 100 мс), в течение которого поток имеет исключительное право на использование вычислительного ресурса ядра процессора до принудительного прерывания планировщиком.

Механизм переключения потоков инициируется на аппаратном уровне:

1. При начале исполнения потока планировщик ОС взводит аппаратный таймер.
2. По истечении кванта времени таймер генерирует прерывание.
3. Процессор переходит в режим ядра (Kernel Mode) и передает управление обработчику прерываний планировщика.
4. Планировщик сохраняет контекст текущего потока (регистры, стек) и выбирает следующий поток из очереди готовых к исполнению (*runqueue*).

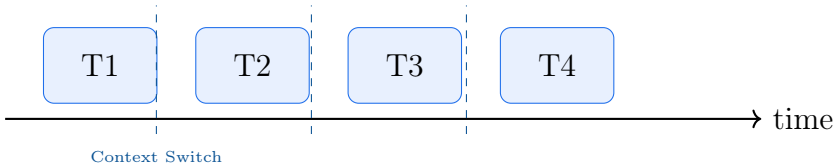

Рис. 2 – Реализация наблюдаемого параллелизма на одном ядре процессора

4.2 Механизмы обработки сигналов и их ограничения

Сигналы в Unix-подобных системах являются средством межпроцессного взаимодействия, однако их реализация по умолчанию не гарантирует надежной доставки всех экземпляров.

Примечание

В стандартной реализации ОС не существует очереди для сигналов. Информация о поступивших сигналах хранится в виде битовой маски (*pending signals mask*). Если ядру поступает новый сигнал того же типа до того, как старый был обработан, бит в маске просто остается взведенным, а информация о втором сигнале теряется.

Это накладывает ограничения на архитектуру систем: сигналы нельзя использовать как надежный механизм передачи данных. Для таких целей следует применять очереди реального времени (POSIX real-time signals), поддерживающие упорядоченную доставку.

4.3 Закон Амдала: пределы масштабируемости

Ускорение вычислений при увеличении числа потоков ограничено наличием последовательных участков кода, которые не могут быть распараллелены (например, инициализация, ввод-вывод или координация потоков).

Определение: Закон Амдала

Пусть P — доля задачи, которую можно распараллелить, а $S = 1 - P$ — последовательная часть. Тогда ускорение U при использовании T потоков вычисляется как:

$$U(T) = \frac{1}{S + \frac{P}{T}} \tag{4.1}$$

При $T \to \infty$ максимальное ускорение стремится к $1/S$.

Если последовательная часть программы составляет 10%, то даже при бесконечном количестве процессоров невозможно получить ускорение более чем в 10 раз. Это подчеркивает важность минимизации критических секций и накладных расходов на синхронизацию.

4.4 Управление привязкой к ядрам (CPU Affinity)

Для оптимизации использования кэша и предсказуемости времени исполнения ОС позволяет закреплять потоки за конкретными логическими ядрами. Это исключает миграцию потока между ядрами и связанные с этим промахи по кэшу (*cache misses*).

```c
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>

void pin_to_core(int core_id) {
    cpu_set_t mask;
    CPU_ZERO(&mask); // Clear the mask
    CPU_SET(core_id, &mask); // Add core_id to mask

    if (sched_setaffinity(0, sizeof(mask), &mask) == -1) {
        perror("sched_setaffinity failed");
    }
}
```

Листинг 3 – Использование sched_setaffinity для закрепления потока за ядром 0

Экспериментально доказано: закрепление двух интенсивных вычислительных потоков на одном физическом ядре приводит к падению производительности каждого до 50% от номинальной, так как они вынуждены делить кванты времени одного исполнительного устройства.

Итоги раздела

- Параллелизм на одном ядре — иллюзия, создаваемая быстрым переключением контекста (квантованием).
- Стандартные сигналы теряются при повторном поступлении из-за использования битовой маски в ядре.
- Максимальное ускорение системы всегда ограничено долей последовательного кода (Закон Амдала).
- CPU Affinity позволяет минимизировать промахи по кэшу, но может привести к деградации при перегрузке конкретных ядер.

5 Аппаратный уровень: Hyper-threading и когерентность кэшей

Производительность многопоточных систем определяется не только алгоритмами планирования ОС, но и микроархитектурными особенностями процессора. В данном разделе рассматриваются механизмы аппаратного переиспользования ресурсов ядра и протоколы обеспечения целостности данных в иерархии кэш-памяти.

5.1 Архитектура Hyper-threading (SMT)

Технология **Hyper-threading** (реализация Simultaneous Multithreading, SMT) направлена на повышение коэффициента полезного действия физического ядра процессора за счет утилизации исполнительных устройств во время простоев.

Определение: Hyper-threading

Hyper-threading — это технология, позволяющая одному физическому ядру процессора функционировать как два логических ядра (процессора). Каждое логическое ядро обладает собственным набором регистров и состоянием (Architecture State), но разделяет с соседним логическим ядром общие исполнительные блоки (ALU, FPU), конвейер и кэши.

Основная цель SMT — скрытие латентности длинных операций. Когда один поток ожидает завершения промаха в кэш (L3 miss может занимать сотни тактов) или выполнения сложной арифметической операции (например, деления), конвейер простаивает. Hyper-threading позволяет моментально переключиться на исполнение инструкций из другого логического потока без накладных расходов на системный вызов или смену контекста ОС.

Примечание

Цена аппаратного параллелизма: логическое ядро всегда медленнее физического при полной загрузке обоих потоков, так как они конкурируют за порты исполнения. В некоторых высоконагруженных или детерминированных системах (HPC, Real-time) Hyper-threading отключают для предотвращения непредсказуемых задержек (*jitter*).

5.2 Протоколы когерентности кэшей: Модель MESI

В многоядерных системах каждое ядро имеет локальные кэши (L1, L2). Если ядро 0 модифицирует данные, ядро 1 должно узнать об этом, чтобы не использовать устаревшее (stale) значение. Для этого используется протокол когерентности, работающий через шину (*Bus Snooping*).

Наиболее распространенным является протокол **MESI**, определяющий четыре состояния кэш-линии:

1. **Modified (M):** Линия присутствует только в текущем кэше, она была изменена (грязная) и не совпадает с оперативной памятью.
2. **Exclusive (E):** Линия присутствует только в текущем кэше, совпадает с памятью.
3. **Shared (S):** Линия присутствует в нескольких кэшах, совпадает с памятью. Доступна только для чтения.
4. **Invalid (I):** Данные в линии не актуальны.

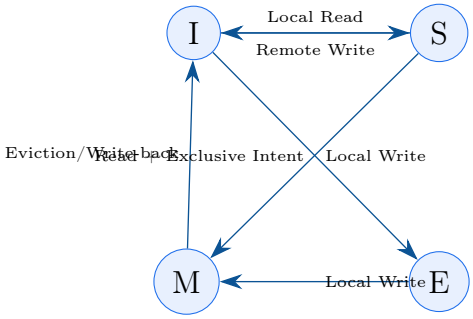

Рис. 3 – Граф переходов состояний протокола MESI

5.3 Проблема False Sharing (Ложное разделение)

Аппаратная единица обмена данными между памятью и кэшем — **кэш-линия** (обычно 64 байта). Это приводит к возникновению побочного эффекта в многопоточном коде.

Определение: False Sharing

False Sharing — ситуация, когда два потока на разных ядрах модифицируют независимые переменные, которые физически расположены в одной и той же кэш-линии.

Механика конфликта:

1. Ядро 0 пишет в переменную `A`. Кэш-линия переходит в состояние **Modified**.
2. Ядро 1 хочет записать в переменную `B`, находящуюся в той же линии. Протокол MESI вынуждает ядро 1 отправить запрос ядру 0 на инвалидацию и получение актуальных данных.
3. Кэш-линия начинает "прыгать"между ядрами (*Cache Line Ping-pong*), вызывая огромные задержки из-за коммуникации по шине.

```
struct CounterStack {
    uint64_t counter_a; // Thread 1 writes here
    uint64_t counter_b; // Thread 2 writes here
    // These 16 bytes sit in the same 64-byte line
};

// Solution: Alignment and Padding
struct CounterFixed {
    alignas(64) uint64_t counter_a;
    alignas(64) uint64_t counter_b;
    // Each counter is now in its own cache line
};
```

Листинг 4 – Пример структуры, подверженной False Sharing

При использовании выравнивания `alignas(64)` или вставки фиктивных полей (*padding*), производительность может возрасти кратно (в 10-20 раз на современных x86 системах), так как ядра перестают конкурировать за одну и ту же единицу памяти.

Итоги раздела

- **Hyper-threading** повышает пропускную способность ядра за счет параллельной загрузки конвейера, но снижает производительность одиночного потока.
- **Протокол MESI** гарантирует когерентность через отслеживание состояний кэш-линий (M, E, S, I).
- **False Sharing** — скрытый враг производительности; возникает из-за гранулярности кэша в 64 байта.
- Решение проблем когерентности в коде требует явного управления расположением данных в памяти (выравнивание).

6 Потокобезопасность в C++ и модель памяти

Проектирование многопоточных систем на языке C++ требует строгого соблюдения правил доступа к разделяемым данным. Нарушение этих правил ведет к неопределенному поведению (Undefined Behavior), которое крайне сложно отлаживать из-за недетерминированности проявлений.

6.1 Определение гонки данных (Data Race)

Согласно стандарту C++, программа содержит гонку данных, если в ней присутствуют две конфликтующие операции в разных потоках, по крайней мере одна из которых является записью, и между ними нет отношения «происходит раньше» (*happens-before*), установленного средствами синхронизации.

Определение: Конфликтующие операции

Две операции над памятью считаются конфликтующими, если они обращаются к одной и той же ячейке памяти (объекту или скалярному типу) одновременно, и хотя бы одна из них модифицирует эту ячейку.

Важные следствия модели памяти C++:

- Чтения из разных потоков никогда не конфликтуют между собой.
- Конфликтующие операции над неатомарными переменными — это **Undefined Behavior**. Процессор может прочитать «мусор», частично записанное значение или вызвать исключение.
- Атомарные операции (через `std::atomic`) упорядочивают доступ к памяти и исключают Data Race на уровне языка.

6.2 Контракт потокобезопасности стандартной библиотеки (STL)

Стандартная библиотека C++ (STL) следует общему правилу относительно потокобезопасности контейнеров (`std::vector`, `std::map` и др.):

1. **Константные методы:** Методы, помеченные как `const`, являются потокобезопасными для одновременного чтения из нескольких потоков. Они не модифицируют внутреннее состояние объекта.
2. **Неконстантные методы:** Любой вызов метода, изменяющего объект (например, `push_back`, `insert`, `operator[]`), требует эксклюзивного доступа. Нельзя вызывать

неконстантный метод одновременно с любым другим методом (даже константным) над тем же объектом без внешней синхронизации (мьютекса).

Примечание

Исключением являются примитивы синхронизации: `std::mutex::lock` и `std::atomic::store` не являются константными методами, но их **можно** вызывать одновременно из разных потоков, так как это их прямое предназначение.

6.3 Анатомия std::shared_ptr в многопоточной среде

Умный указатель `std::shared_ptr` часто вводит разработчиков в заблуждение относительно своей безопасности. Его структура состоит из двух указателей: на сам объект и на так называемый **управляющий блок** (Control Block).

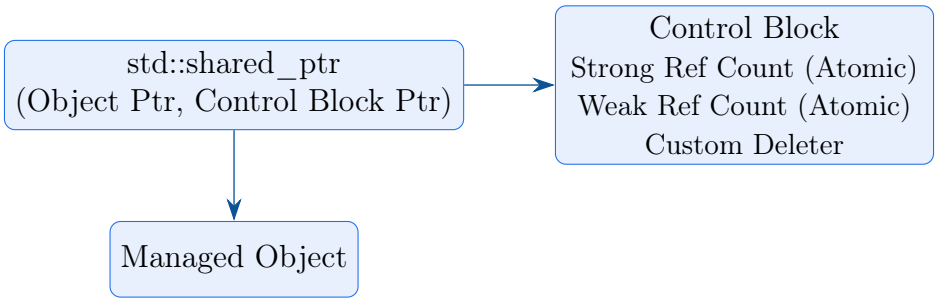

Рис. 4 – Внутренняя структура std::shared_ptr

Что гарантирует стандарт: Счетчик ссылок в управляющем блоке изменяется атомарно. Если два потока одновременно создают копии `shared_ptr` или уничтожают их, счетчик ссылок всегда будет консистентен. Это гарантирует корректное удаление объекта ровно один раз.

Что НЕ гарантирует стандарт: Сам объект `shared_ptr` (пара указателей) не является атомарным. Если один поток записывает в переменную `ptr` новый указатель, а другой поток одновременно читает из этой же переменной `ptr`, возникает Data Race. Аналогично, данные внутри управляемого объекта никак не защищены от гонок.

6.4 Thread Local Storage (TLS)

Для устранения конкуренции за глобальные данные используется механизм локального хранилища потока.

Определение: Thread Local Storage (TLS)

TLS — механизм, позволяющий объявлять переменные, копия которых создается индивидуально для каждого потока. Изменение такой переменной в одном потоке никак не влияет на её значение в других потоках.

Классический пример — переменная `errno`. В однопоточных системах это была глобальная целочисленная переменная. В многопоточной среде это привело бы к тому, что системный вызов в потоке A перезаписал бы код ошибки для потока B. Современная реализация `errno` скрывает за собой вызов функции, возвращающей адрес в TLS-сегменте текущего потока.

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

```cpp

// Each thread gets its own instance of 'counter'
thread_local int counter = 0;

void work() {
    counter++;
    std::cout << "Thread ID: " << std::this_thread::get_id()
              << ", counter: " << counter << "\n";
}

int main() {
    std::thread t1(work);
    std::thread t2(work);
    t1.join(); t2.join();
    // Output: both threads will print 'counter: 1'
}
```

Листинг 5 – Пример использования thread_local

6.5 Диагностика через Thread Sanitizer (TSan)

Для автоматического обнаружения гонок данных используется инструмент **Thread Sanitizer**. Он инструментирует обращения к памяти и отслеживает порядок доступа в рантайме.

При обнаружении гонки TSan генерирует отчет, содержащий:

1. **Write of size N...** — поток, выполнивший запись, и стек его вызовов.
2. **Previous read of size N...** — поток, выполнивший конфликтующее чтение, и его стек.
3. **Location is heap block...** — адрес и происхождение памяти, ставшей причиной конфликта.

Диагностика TSan является критически важной, так как многие гонки данных не проявляются при обычном тестировании, но приводят к фатальным сбоям под высокой нагрузкой.

Итоги раздела

- Гонка данных — это отсутствие синхронизации при обращении к одной ячейке памяти, где есть хотя бы одна запись.
- STL гарантирует безопасность только для одновременных вызовов `const`-методов.
- `std::shared_ptr` атомарно управляет временем жизни объекта, но не защищает сам указатель и данные внутри.
- `thread_local` переменные исключают конкуренцию, создавая изолированные копии данных для каждого потока.

7 Потокобезопасность в C++ и модель памяти

Проектирование многопоточных систем на языке C++ требует строгого соблюдения правил доступа к разделяемым данным. Нарушение этих правил ведет к неопределенному поведению (Undefined Behavior), которое крайне сложно отлаживать из-за

недетерминированности проявлений.

7.1 Определение гонки данных (Data Race)

Согласно стандарту C++, программа содержит гонку данных, если в ней присутствуют две конфликтующие операции в разных потоках, по крайней мере одна из которых является записью, и между ними нет отношения «происходит раньше» (*happens-before*), установленного средствами синхронизации.

Определение: Конфликтующие операции

Две операции над памятью считаются конфликтующими, если они обращаются к одной и той же ячейке памяти (объекту или скалярному типу) одновременно, и хотя бы одна из них модифицирует эту ячейку.

Важные следствия модели памяти C++:

- Чтения из разных потоков никогда не конфликтуют между собой.
- Конфликтующие операции над неатомарными переменными — это **Undefined Behavior**. Процессор может прочитать «мусор», частично записанное значение или вызвать исключение.
- Атомарные операции (через `std::atomic`) упорядочивают доступ к памяти и исключают Data Race на уровне языка.

7.2 Контракт потокобезопасности стандартной библиотеки (STL)

Стандартная библиотека C++ (STL) следует общему правилу относительно потокобезопасности контейнеров (`std::vector`, `std::map` и др.):

1. **Константные методы:** Методы, помеченные как `const`, являются потокобезопасными для одновременного чтения из нескольких потоков. Они не модифицируют внутреннее состояние объекта.
2. **Неконстантные методы:** Любой вызов метода, изменяющего объект (например, `push_back`, `insert`, `operator[]`), требует эксклюзивного доступа. Нельзя вызывать неконстантный метод одновременно с любым другим методом (даже константным) над тем же объектом без внешней синхронизации (мьютекса).

Примечание

Исключением являются примитивы синхронизации: `std::mutex::lock` и `std::atomic::store` не являются константными методами, но их **можно** вызывать одновременно из разных потоков, так как это их прямое предназначение.

7.3 Анатомия std::shared_ptr в многопоточной среде

Умный указатель `std::shared_ptr` часто вводит разработчиков в заблуждение относительно своей безопасности. Его структура состоит из двух указателей: на сам объект и на так называемый **управляющий блок** (Control Block).

Что гарантирует стандарт: Счетчик ссылок в управляющем блоке изменяется атомарно. Если два потока одновременно создают копии `shared_ptr` или уничтожают их, счетчик ссылок всегда будет консистентен. Это гарантирует корректное удаление объекта ровно один раз.

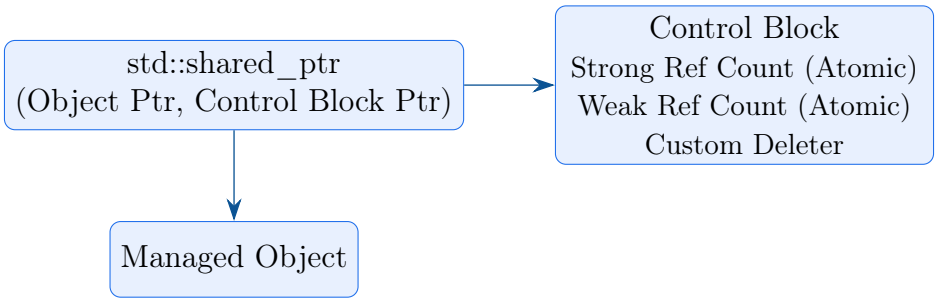

Рис. 5 – Внутренняя структура std::shared_ptr

Что НЕ гарантирует стандарт: Сам объект `shared_ptr` (пара указателей) не является атомарным. Если один поток записывает в переменную `ptr` новый указатель, а другой поток одновременно читает из этой же переменной `ptr`, возникает Data Race. Аналогично, данные внутри управляемого объекта никак не защищены от гонок.

7.4 Thread Local Storage (TLS)

Для устранения конкуренции за глобальные данные используется механизм локального хранилища потока.

Определение: Thread Local Storage (TLS)

TLS — механизм, позволяющий объявлять переменные, копия которых создается индивидуально для каждого потока. Изменение такой переменной в одном потоке никак не влияет на её значение в других потоках.

Классический пример — переменная `errno`. В однопоточных системах это была глобальная целочисленная переменная. В многопоточной среде это привело бы к тому, что системный вызов в потоке A перезаписал бы код ошибки для потока B. Современная реализация `errno` скрывает за собой вызов функции, возвращающей адрес в TLS-сегменте текущего потока.

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

// Each thread gets its own instance of 'counter'
thread_local int counter = 0;

void work() {
    counter++;
    std::cout << "Thread ID: " << std::this_thread::get_id()
              << ", counter: " << counter << "\n";
}

int main() {
    std::thread t1(work);
    std::thread t2(work);
    t1.join(); t2.join();
    // Output: both threads will print 'counter: 1'
}
```

Листинг 6 – Пример использования thread_local

7.5 Диагностика через Thread Sanitizer (TSan)

Для автоматического обнаружения гонок данных используется инструмент **Thread Sanitizer**. Он инструментирует обращения к памяти и отслеживает порядок доступа в рантайме.

При обнаружении гонки TSan генерирует отчет, содержащий:

1. **Write of size N...** — поток, выполнивший запись, и стек его вызовов.
2. **Previous read of size N...** — поток, выполнивший конфликтующее чтение, и его стек.
3. **Location is heap block...** — адрес и происхождение памяти, ставшей причиной конфликта.

Диагностика TSan является критически важной, так как многие гонки данных не проявляются при обычном тестировании, но приводят к фатальным сбоям под высокой нагрузкой.

Итоги раздела

- Гонка данных — это отсутствие синхронизации при обращении к одной ячейке памяти, где есть хотя бы одна запись.
- STL гарантирует безопасность только для одновременных вызовов `const`-методов.
- `std::shared_ptr` атомарно управляет временем жизни объекта, но не защищает сам указатель и данные внутри.
- `thread_local` переменные исключают конкуренцию, создавая изолированные копии данных для каждого потока.

8 Примитивы синхронизации и Lock-free механизмы

В дополнение к базовым мьютексам и условным переменным, системное программирование предлагает специализированные примитивы, оптимизированные под конкретные паттерны доступа и аппаратные возможности процессора. В данном разделе рассматриваются высокоуровневые средства координации и фундамент безблокировочных (lock-free) алгоритмов.

8.1 Семафоры: управление доступом к ресурсам

Семафор является одним из старейших примитивов синхронизации, предложенным Эдсгером Дейкстрой. В отличие от мьютекса, семафор не обладает понятием владения.

Определение: Семафор

Семафор — это целочисленный счетчик (permits), поддерживающий две атомарные операции: декремент (*Wait*/*P*) и инкремент (*Signal*/*V*). Если при попытке декремента счетчик равен нулю, поток блокируется до тех пор, пока значение не станет положительным.

В стандартной библиотеке C++ представлены `std::counting_semaphore` (счетный) и `std::binary_semaphore` (двоичный, аналогичен мьютексу, но без привязки к потоку-владельцу). Основные сценарии использования:

- **Ограничение параллелизма:** Например, лимитирование количества одновременных

запросов к базе данных или внешнему API.

- **Сценарий Producer-Consumer:** Семафор может хранить количество доступных для обработки элементов в буфере.

Примечание

Важное различие: `unlock()` у мьютекса обязан вызывать тот же поток, который вызвал `lock()`. Для семафора это правило не действует — один поток может «взять» разрешение, а другой — «вернуть». Нарушение правила владения мьютекса в C++ ведет к **Undefined Behavior**.

8.2 RW-Lock: оптимизация для сценариев с преобладанием чтения

Многие структуры данных читаются значительно чаще, чем модифицируются. Обычный мьютекс избыточно сериализует читателей, что снижает производительность на многоядерных системах.

Определение: RW-Lock (Read-Writer Lock)

Примитив, разделяющий блокировку на два режима:

1. **Shared (Read):** Позволяет неограниченному числу читателей заходить в критическую секцию одновременно.
2. **Exclusive (Write):** Гарантирует, что только один поток-писатель имеет доступ, блокируя при этом всех читателей и других писателей.

В C++17 это реализовано через `std::shared_mutex`. Главный компромисс при реализации RW-Lock — стратегия разрешения конфликтов между новыми читателями и ожидающими писателями.

Примечание

Проблема Starvation (Голодание): Если отдавать приоритет читателям, постоянный поток новых *Shared*-захватов может бесконечно откладывать выполнение писателя. Если отдавать приоритет писателям, снижается параллелизм чтения. Большинство реализаций ОС стараются соблюдать баланс, запрещая новым читателям захват, если в очереди уже есть ожидающий писатель.

8.3 Барьеры: фазовая синхронизация

Барьеры используются в задачах, разбитых на последовательные этапы, где ни один поток не может начать этап $N+1$, пока все потоки не завершат этап N.

Пример: Вычисление слоев в полносвязной нейронной сети. Умножение матрицы на вектор распараллеливается по строкам, но для перехода к следующему слою необходим полный вектор результатов предыдущего.

```
void worker(std::barrier<>& sync_point, int layer_count) {
    for (int i = 0; i < layer_count; ++i) {
        compute_layer_part(i);
        // All threads must arrive here before any can continue
        sync_point.arrive_and_wait();
    }
}
```

Листинг 7 – Пример использования std::barrier

8.4 Атомарные операции и Compare-and-Swap (CAS)

Фундаментом всех эффективных примитивов синхронизации являются атомарные инструкции процессора. Самой мощной из них является *Compare-and-Swap* (CAS).

Определение: Compare-and-Swap (CAS)

Операция над атомарной переменной, принимающая ожидаемое (*expected*) и желаемое (*desired*) значения. Если текущее значение переменной равно *expected*, оно заменяется на *desired*. Операция возвращает `true` при успехе или обновляет *expected* текущим значением и возвращает `false` при неудаче.

В x86 это транслируется в инструкцию `lock cmpxchg`. На ней строятся циклы перезапуска (*CAS loops*), заменяющие блокировки.

```
void atomic_increment(std::atomic<int>& var) {
    int expected = var.load();
    // Use weak for performance in loops on some architectures
    while (!var.compare_exchange_weak(expected, expected + 1)) {
        // 'expected' is updated automatically by compare_exchange on failure
    }
}
```

Листинг 8 – Реализация атомарного инкремента через CAS loop

8.5 Аппаратная специфика: Weak vs Strong CAS

Стандарт C++ предоставляет две версии: `compare_exchange_strong` и `compare_exchange_weak`.

1. **Strong:** Гарантирует успех, если значения равны.
2. **Weak:** Может вернуть `false`, даже если значения равны (ложный провал или *spurious failure*).

Причины существования `weak` версии кроются в архитектуре процессоров. Архитектуры RISC (ARM, PowerPC) используют механизм *Load-Link / Store-Conditional* (LL/SC). Любое прерывание, переключение контекста или вытеснение кэш-линии между LL и SC приводит к провалу записи, даже если данные не изменились. На x86 `strong` и `weak` обычно идентичны по производительности, но на ARM использование `weak` внутри цикла эффективнее, так как позволяет избежать вложенных циклов в генерируемом машинном коде.

Итоги раздела

- **Семафоры** подходят для ограничения ресурсов и не навязывают владение потоком.
- **RW-Locks** критичны для систем с высокой частотой чтения, но требуют защиты от голодания писателей.
- **Барьеры** обеспечивают фазовую синхронизацию в параллельных алгоритмах.
- **CAS** является базовым блоком для lock-free алгоритмов. Выбор между **weak** и

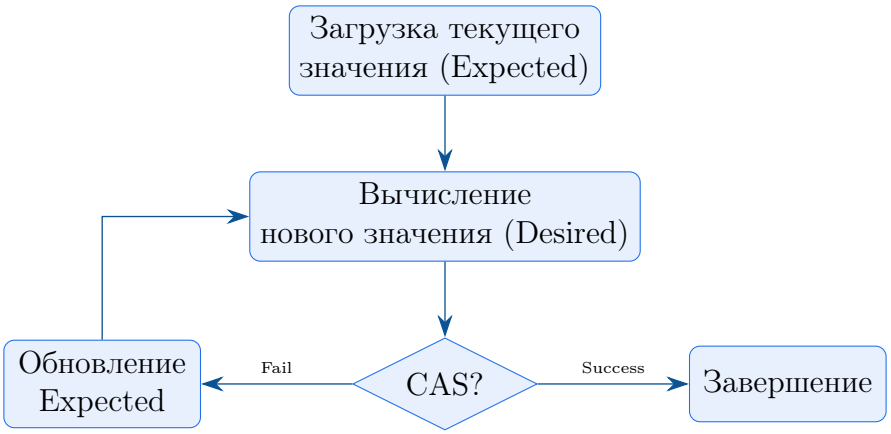

Рис. 6 – Логика CAS-цикла для обновления значения

strong версиями зависит от архитектуры и наличия внешнего цикла.