Курс: *Архитектура компьютера и ОС*

Лекция 10: Параллелизм и синхронизация (часть 2)

Лектор: Евгений Соколов

Дата: 25.11.2025

Содержание

1 Введение: Сигналы и наблюдаемый параллелизм

В начале лекции был затронут вопрос о сигналах: существует ли для них очередь?

Примечание

По умолчанию **очереди сигналов не существует**. Операционная система поддерживает маску ожидающих (pending) сигналов. Если приходит два одинаковых сигнала (например, от пользователя и от системы), а обработчик ещё не вызван, то в маске просто выставляется бит. Второй сигнал того же типа может быть потерян.

1.1 Наблюдаемый параллелизм и квантование

Даже на одноядерных системах пользователи наблюдают иллюзию параллельного исполнения множества потоков. Это достигается за счёт **Time Slicing** (квантования времени).

- Операционная система нарезает исполнение потоков на небольшие интервалы — *кванты* (единицы или десятки миллисекунд).
- По истечении кванта таймер прерывает исполнение, планировщик ОС сохраняет контекст текущего потока и загружает следующий.
- Для пользователя переключение происходит незаметно, создавая видимость одновременной работы.

Однако увеличение количества потоков сверх физических возможностей процессора не даёт прироста производительности, а лишь увеличивает накладные расходы на переключение контекста.

2 Аппаратная поддержка многопоточности

2.1 Hyper-threading

Процессоры часто простаивают в ожидании данных из памяти (cache miss, ~500 тактов) или при выполнении долгих арифметических операций. Чтобы утилизировать эти ресурсы, была разработана технология Hyper-threading.

Определение: Hyper-threading

Технология, позволяющая одному *физическому* ядру процессора представляться операционной системе как два *логических* ядра.

- **Дублируются:** регистры (архитектурное состояние).
- **Разделяются:** исполнительные устройства (ALU), кэши, шина.

Если один логический поток простаивает (ждет память), физическое ядро переключается на инструкции второго потока. Это увеличивает пропускную способность (throughput), но производительность одного отдельного потока может снизиться из-за конкуренции за ресурсы ядра.

2.2 Привязка к ядрам (CPU Affinity)

Мы можем программно управлять тем, на каких ядрах исполняется поток, используя системные вызовы семейства `sched_`.

Пример закрепления потока за нулевым ядром:

```cpp
cpu_set_t cpuset;
CPU_ZERO(&cpuset);
CPU_SET(0, &cpuset); // Allow execution only on Core 0

pthread_setaffinity_np(thread.native_handle(), sizeof(cpu_set_t), &cpuset);
```

Если запустить два вычислительно тяжёлых потока и привязать их к одному физическому ядру, время выполнения увеличится вдвое по сравнению с их запуском на разных ядрах.

2.3 Закон Амдала

Существует теоретический предел ускорения программы при распараллеливании. Он описывается Закон Амдала.

Пусть:

- P — доля программы, которую можно распараллелить (parallelizable).
- S — доля программы, исполняемая последовательно (serial), $S = 1 - P$.
- T — количество потоков.

Ускорение при использовании T потоков:

$$\text{Speedup}(T) = \frac{1}{S + \frac{P}{T}} \tag{2.1}$$

Максимально возможное ускорение (при $T \to \infty$):

$$\text{Max Speedup} = \frac{1}{S} = 1 + \frac{P}{S} \tag{2.2}$$

Даже если $P = 0.95$ (95% кода параллелится), максимальное ускорение не превысит 20 раз.

3 Thread Safety (Потокобезопасность)

Основная проблема многопоточности — доступ к общим данным.

Определение: Конфликтующая операция (Conflicting Operation)

Операция над одной ячейкой памяти, где участвуют как минимум два потока, и хотя бы один из них выполняет **запись**.

Стандарт C++ гласит: конфликтующие операции, выполняемые одновременно без синхронизации, приводят к **Data Race** и Undefined Behavior.

3.1 Контракты стандартной библиотеки C++

Для контейнеров (например, `std::vector`, `std::map`) действуют следующие правила:

1. **Константные методы** (‘const‘) считаются потокобезопасными (можно вызывать одновременно из разных потоков). Например: ‘v.size()‘, ‘v.capacity()‘.
2. **Неконстантные методы** не являются потокобезопасными. Нельзя вызывать ‘v.push_back()‘ одновременно с ‘v.size()‘ или другим ‘push_back()‘.
3. Разные экземпляры объектов независимы.

Исключения (thread-safe mutable):

- `std::atomic` — все методы потокобезопасны.
- `std::mutex` — ‘lock()‘/‘unlock()‘ меняют состояние, но безопасны.
- `std::cin`, `std::cout` — операторы ‘«‘, ‘»‘ потокобезопасны (не упадут, но вывод может перемешаться).

3.2 Свободные функции

- `printf`, `scanf`: Thread-safe (внутри есть глобальные блокировки).
- `getenv`: Помечена как *MT-Safe env*. Безопасна, если никто одновременно не вызывает модифицирующие функции вроде `setenv`, `clearenv`.

4 Примитивы синхронизации

Помимо ‘std::mutex‘ и ‘std::condition_variable‘, существуют специализированные примитивы.

4.1 Семафор (Semaphore)

Счётчик разрешений (permits). Не имеет владельца (в отличие от мьютекса, который должен освобождать тот же поток, что и захватил).

- **Wait (acquire):** Декрементирует счётчик. Если 0 — блокирует поток.
- **Signal (release):** Инкрементирует счётчик, будит ждущий поток.

Применение: ограничение количества одновременных доступов (rate limiting), очереди. В C++20: `std::counting_semaphore`, `std::binary_semaphore`.

4.2 RWLock (Read-Write Lock)

Позволяет множеству читателей работать одновременно, но писателю дает эксклюзивный доступ. В C++: `std::shared_mutex`.

- `lock_shared()`: Захват для чтения (несколько потоков могут держать одновременно).
- `lock()`: Захват для записи (эксклюзивно: ни читателей, ни писателей).

Проблема: Возможен *starvation* (голодание) писателей, если поток читателей непрерывен, или читателей, если приоритет у писателей.

4.3 Барьер (Barrier)

Синхронизирует прохождение N потоками определённой точки. Пример: нейросети. Нужно посчитать слой i полностью всеми потоками, прежде чем переходить к слою $i+1$.

```
// Example of using std::barrier
std::barrier sync_point(num_threads);

// In the worker thread code:
for (int l = 0; l < layers; ++l) {
    ProcessLayerPart(l);
    sync_point.arrive_and_wait(); // Wait for others
}
```

5 Thread Local Storage (TLS)

Иногда каждому потоку нужна своя копия глобальной переменной (например, буфер или код ошибки). Используется ключевое слово `thread_local`.

```
thread_local int value; // Each thread has its own instance
```

Классический пример: `errno`. В однопоточных системах это была глобальная переменная. В многопоточных она реализована как макрос, возвращающий разыменованный указатель на TLS-переменную:

```
#define errno (*__errno_location())
```

Это позволяет разным потокам иметь разные коды ошибок одновременно.

Итоги раздела

Итоги по инструментам:

- Используйте `const` методы для параллельного чтения.
- Для специфичных задач используйте `semaphore` (лимиты), `shared_mutex` (read-heavy), `barrier` (этапы).
- `thread_local` для изоляции данных потока.

6 Shared Pointer и безопасность

Вопрос: является ли `std::shared_ptr` потокобезопасным? **Ответ:** Частично.

- **Control Block (счётчик ссылок):** Потокобезопасен. Инкремент/декремент счётчика атомарен (используются атомарные инструкции). Можно копировать/удалять ‘shared_ptr‘ из разных потоков.
- **Указываемый объект:** НЕ защищён. Если два потока пишут в данные через разные ‘shared_ptr‘, указывающие на один объект — будет гонка.

Для отладки гонок полезен инструмент **ThreadSanitizer (TSan)**. Он ловит:

- Write-Read races.
- Write-Write races.

Запуск: компиляция с ‘-fsanitize=thread‘.

7 Атомики и модель памяти

7.1 Compare-And-Swap (CAS)

Мощная атомарная операция для реализации lock-free алгоритмов. В C++: `compare_exchange_strc` и `compare_exchange_weak`.

Логика работы:

```
// CAS Pseudocode
bool CAS(atomic<int>& obj, int& expected, int desired) {
    if (obj == expected) {
        obj = desired;
        return true;
    } else {
        expected = obj; // Updates 'expected' with actual value
```

```
        return false;
    }
}
```

Использование в цикле (CAS-loop) для реализации ‘fetch_add‘:

```
std::atomic<int> atom;
int expected = atom.load();
while (!atom.compare_exchange_weak(expected, expected + 1)) {
    // If failed, 'expected' is automatically updated inside the function
    // Loop repeats with new 'expected'
}
```

Weak vs Strong:

- `Weak`: Может вернуть ‘false‘ (спонтанный сбой), даже если значение равно ожидаемому. На платформах вроде ARM/PowerPC это эффективнее (нет вложенного цикла). На x86 (Intel/AMD) разницы в ассемблере нет (обе транслируются в ‘LOCK CMPXCHG‘).
- `Strong`: Гарантирует успех, если значение совпало. Обычно содержит цикл внутри себя на RISC-архитектурах.

8 Кэши и когерентность

8.1 Иерархия памяти и протокол MESI

У каждого ядра есть свои L1/L2 кэши. Если одно ядро пишет в переменную, другие должны узнать об этом, чтобы не читать устаревшие данные. Для этого используется протокол когерентности, например, **MESI**:

- **M (Modified):** Данные изменены, есть только в этом кэше.
- **E (Exclusive):** Данные только в этом кэше, совпадают с памятью.
- **S (Shared):** Данные есть в нескольких кэшах (только чтение).
- **I (Invalid):** Данные устарели.

Коммуникация между ядрами для поддержки когерентности стоит дорого.

8.2 False Sharing (Ложное разделение)

Проблема возникает, когда независимые переменные попадают в одну **кэш-линию** (обычно 64 байта).

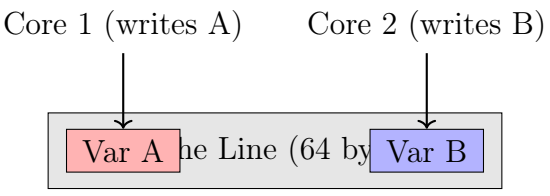

Рис. 1 – Иллюстрация False Sharing

Если Thread 1 пишет в ‘Var A‘, а Thread 2 пишет в ‘Var B‘, и они лежат рядом, ядра будут бесконечно передавать друг другу права на владение всей кэш-линией. Это катастрофически снижает производительность.

Решение: Выравнивание (padding).

```
struct alignas(64) AlignedData {
    int value;
    // Compiler adds padding up to 64 bytes
};
```

9 Процессы и Fork в многопоточной среде

Системный вызов `fork()` создаёт копию процесса. **Правило:** В дочернем процессе продолжает исполнение *только тот поток*, который вызвал `fork`. Остальные потоки исчезают.

Проблема: Если исчезнувший поток держал мьютекс (например, внутри ‘malloc‘), этот мьютекс останется навечно заблокированным в дочернем процессе. Любая попытка вызвать ‘malloc‘ в "ребёнке"приведёт к дедлоку.

Решение: Использовать ‘pthread_atfork‘ для регистрации хуков, которые захватывают нужные блокировки перед форком и отпускают их после (в обоих процессах), обеспечивая консистентное состояние.

10 Futex (Fast Userspace Mutex)

Системный вызов Linux, на котором строятся эффективные мьютексы. Идея: избегать входа в ядро (syscall), если нет конкуренции (fast path). В ядро идем только чтобы уснуть (wait).

- `futex_wait(addr, val)`: "Если по адресу `addr` лежит значение `val`, то усыпи меня". Проверка атомарна внутри ядра.
- `futex_wake(addr, count)`: Разбуди `count` потоков, ждущих на этом адресе.

Пример примитивного мьютекса на атомике и фьютексе:

```
std::atomic<int> flag{0}; // 0 - free, 1 - locked

void lock() {
    while (flag.exchange(1) != 0) { // Try to acquire
        // If busy, go to wait state
        syscall(SYS_futex, &flag, FUTEX_WAIT, 1, ...);
    }
}

void unlock() {
    flag.store(0);
    syscall(SYS_futex, &flag, FUTEX_WAKE, 1, ...);
}
```

Примечание

В реальных реализациях (‘std::mutex‘) используется более сложная логика с тремя состояниями (свободен, занят, занят+есть ждущие), чтобы избежать лишних системных вызовов ‘wake‘.

Итоги раздела

Итоги лекции:

1. Параллелизм ограничивается не только числом ядер, но и синхронизацией (Закон Амдала) и аппаратными эффектами (False Sharing).
2. Hyper-threading позволяет утилизировать простои ядра, но делит ресурсы.
3. Вызов `fork()` в многопоточной программе опасен из-за состояния блокировок.
4. Современные блокировки строятся на атомиках + Futex для эффективного ожидания.