# Курс: *Архитектура компьютера и ОС*

Лекция 2: Оптимизации CPU, числа с плавающей запятой и многопоточность

Лектор: Евгений Соколов

Дата: 08.09.2025

## Содержание

# 1 Оптимизации в современных процессорах

Современные центральный процессор (CPU) применяют множество сложных оптимизаций для достижения высокой производительности. Рассмотрим ключевые из них: организацию кэш-памяти, внеочередное и спекулятивное исполнение инструкций, а также предсказание ветвлений.

## 1.1 Ассоциативность кэша и её влияние на производительность

Кэш — это небольшая, но очень быстрая память, расположенная близко к вычислительным ядрам процессора. Она хранит копии часто используемых данных из основной, более медленной памяти. Эффективность кэша напрямую влияет на скорость работы программ.

**Определение: Ассоциативность кэша**

Ассоциативность определяет, в скольких возможных местах (слотах) кэша может быть размещена определённая строка данных (кэш-линия) из основной памяти. Кэш-линии группируются в множества (sets). В $N$-ассоциативном кэше каждая кэш-линия может быть помещена в любое из $N$ мест внутри своего множества.

Типичные значения ассоциативности: 2, 4, 8 или 16. Прямо-отображаемый кэш (1-ассоциативный) прост, но страдает от коллизий: две кэш-линии, претендующие на одно и то же место, будут постоянно вытеснять друг друга. Увеличение ассоциативности снижает вероятность коллизий, но усложняет аппаратуру.

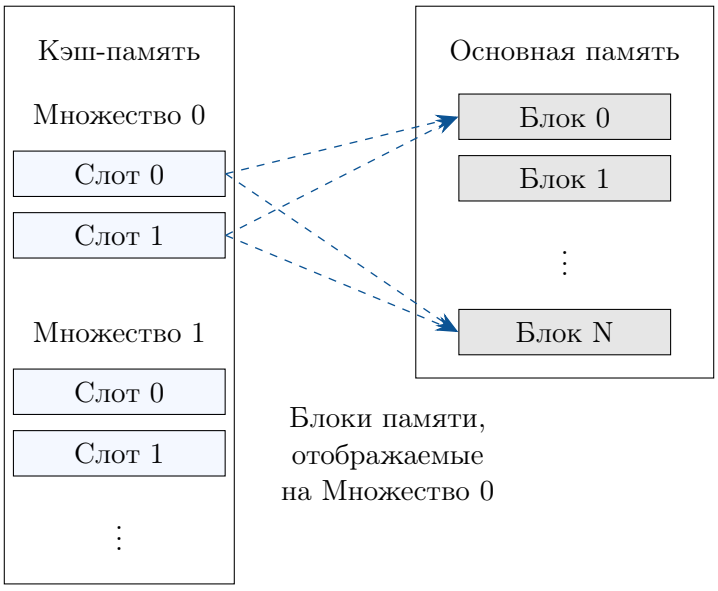


**Рис. 1** – Схема 2-ассоциативного кэша: любой блок памяти, чей адрес отображается на Множество 0, может быть помещён в любой из двух слотов этого множества.

Неправильный паттерн доступа к памяти может привести к «отравлению» кэша. Рассмотрим пример транспонирования матрицы. При обходе матрицы по столбцам адреса соседних элементов отстоят друг от друга на размер строки. Если размер строки кратен большой степени двойки, адреса элементов из разных строк, но одного столбца, могут отображаться на одно и то же или на малое подмножество множеств в кэше.

Это приводит к постоянным промахам (cache miss), так как кэш-линии вытесняют друг друга. Эксперименты показывают, что транспонирование матрицы $512 \times 512$ (где

$512 = 2^9$) выполняется значительно медленнее, чем матриц $511 \times 511$ или $513 \times 513$, именно по этой причине.

## 1.2 Конвейерное и внеочередное исполнение

Для ускорения обработки инструкций CPU использует конвейер. Выполнение каждой инструкции разбивается на стадии (выборка, декодирование, исполнение, доступ к памяти, запись результата). Это позволяет одновременно обрабатывать несколько инструкций на разных стадиях.

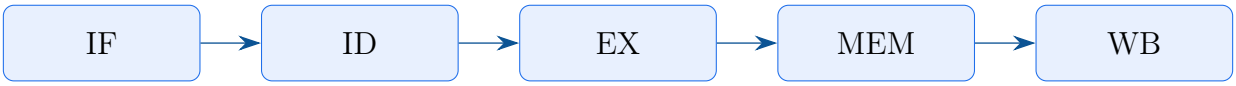


**Рис. 2** – Классический 5-стадийный конвейер обработки инструкций

Современные процессоры идут дальше и реализуют внеочередное исполнение инструкций (Out-of-Order Execution) (OoOE).

**Определение: Внеочередное исполнение (Out-of-Order Execution)**

Это способность CPU исполнять инструкции не в том порядке, в котором они указаны в программе, а в порядке готовности их операндов. Это позволяет обходить задержки (например, при ожидании данных из памяти) и лучше загружать исполнительные устройства процессора.

Процессор анализирует зависимости по данным между инструкциями. Если две инструкции не зависят друг от друга, они могут быть выполнены параллельно или в обратном порядке. Для разрешения конфликтов по регистрам используется **переименование регистров**: архитектурным регистрам (видимым программисту) ставятся в соответствие физические регистры внутри CPU. Это позволяет устранить ложные зависимости.

## 1.3 Спекулятивное исполнение и уязвимости

OoOE тесно связано со спекулятивным исполнением. Процессор может не только переупорядочивать, но и «угадывать» результат условных переходов (ветвлений) и начинать выполнять инструкции из наиболее вероятной ветки кода ещё до того, как условие будет вычислено.

**Примечание**

Спекулятивное исполнение может оставлять следы в кэше. Если процессор спекулятивно выполнил чтение из памяти, к которой у программы нет доступа, данные могут попасть в кэш. Хотя результат операции будет отброшен после обнаружения ошибки доступа, наличие данных в кэше можно определить по времени доступа к ним. На этом принципе были основаны уязвимости класса **Meltdown** и **Spectre**.

## 1.4 Предсказание ветвлений (Branch Prediction)

Эффективность спекулятивного исполнения зависит от точности предсказания ветвлений. Ошибка предсказания (branch misprediction) очень дорога: CPU должен сбросить конвейер, отменить результаты спекулятивно выполненных инструкций и начать выполнение с правильной ветки.

Рассмотрим пример: подсчёт элементов в массиве, которые меньше определённого порога.

- **Отсортированный массив**: Предсказатель легко угадывает результат сравнения. Сначала все элементы будут меньше порога, потом — больше. Переход будет только один. Производительность высокая.
- **Неотсортированный (случайный) массив**: Результат сравнения непредсказуем. Процент ошибок предсказания высок ($\approx 50\%$), что приводит к значительному падению производительности.

Компиляторы знают об этой проблеме и могут применять оптимизации, чтобы избежать ветвлений. Например, условное приращение счётчика ‘if (x $<$ 128) sum++;‘ может быть заменено на инструкцию условного перемещения (conditional move), которая не содержит прыжка и не нагружает предсказатель ветвлений.

**Итоги раздела**

- **Ассоциативность кэша** помогает бороться с коллизиями, но паттерны доступа к памяти с шагом, кратным степени двойки, могут снизить её эффективность.
- **Конвейер** и **OoOE** позволяют исполнять несколько инструкций параллельно, скрывая задержки.
- **Спекулятивное исполнение** на основе предсказания ветвлений ускоряет код, но ошибки предсказания дорого обходятся.
- Компиляторы могут преобразовывать код для минимизации ветвлений и улучшения производительности.

# 2 Представление нецелых чисел

Целочисленные типы не могут представлять дробные значения. Для этого в вычислительной технике используются два основных подхода: числа с фиксированной и с плавающей запятой.

## 2.1 Числа с фиксированной точкой (Fixed-Point)

Идея проста: хранить число как целое, но считать, что дробная точка находится в заранее определённой позиции. Фактически это целое число, делённое на фиксированную степень двойки.

- **Преимущества**: Арифметика быстрая, так как используются целочисленные операции.
- **Недостатки**: Ограниченный и фиксированный диапазон значений. Сложно представлять одновременно очень большие и очень маленькие числа. Точность постоянна по всему диапазону.

Например, число $5.125_{10}$ в двоичном виде равно $101.001_2$. Если мы договоримся хранить 3 знака после запятой, то это число будет храниться как целое $101001_2$.

## 2.2 Стандарт IEEE 754: числа с плавающей запятой

Для гибкого представления широкого диапазона чисел был разработан стандарт стандарт двоичной арифметики с плавающей запятой (IEEE 754). Число представляется в научном формате:

$$\text{fp} = S \cdot M \cdot 2^E \tag{2.1}$$

где:

- $S$ — знак (+1 или -1).
- $M$ — мантисса (значащая часть), нормализованное число в диапазоне $[1.0, 2.0)$.
- $E$ — экспонента (показатель степени).

В двоичном представлении это выглядит так:

| **Знак (1 бит)** | **Экспонента (несколько бит)** | **Мантисса (остальные биты)** |
|---|---|---|

Поскольку нормализованная мантисса всегда начинается с единицы (1....), эта единица не хранится явно («скрытый бит»), что даёт дополнительный бит точности.

Для хранения отрицательных экспонент используется **смещение (bias)**. Хранимое значение экспоненты — это беззнаковое целое, из которого вычитается bias для получения реального показателя степени.

$$E_{\text{real}} = E_{\text{stored}} - \text{bias} \tag{2.2}$$

## 2.3 Специальные случаи

Стандарт IEEE 754 определяет кодирование для особых значений:

- **Денормализованные числа**: Если все биты экспоненты равны 0, скрытый бит считается равным 0 (а не 1). Это позволяет плавно представлять числа, очень близкие к нулю, заполняя «дыру» между нулём и наименьшим нормализованным числом.
- **Бесконечность** ($\pm\infty$): Если все биты экспоненты равны 1, а все биты мантиссы равны 0. Получается при переполнении или делении на ноль (1.0/0.0).

- **Не-число** (**«не число» (Not a Number) (NaN)**): Если все биты экспоненты равны 1, а мантисса не равна нулю. Результат некорректных операций, таких как $\infty - \infty$ или $\sqrt{-1}$.

**Примечание**

NaN обладает особым свойством: любое сравнение с NaN, даже ‘NaN == NaN‘, возвращает ‘false‘. Это требует особой осторожности при проверках.

## 2.4 Погрешности и работа в C++

Арифметика с плавающей запятой неточна. Это приводит к нарушению привычных математических законов:

- **Неассоциативность сложения**: $(a + b) + c$ может не равняться $a + (b + c)$, особенно если числа сильно различаются по величине.
- **Недистрибутивность**: $a \cdot (b + c)$ может не равняться $a \cdot b + a \cdot c$.

Для минимизации ошибок при суммировании большого количества чисел их рекомендуется сортировать и складывать от меньших по модулю к большим.

В C++ есть три основных типа с плавающей запятой:

**Таблица 1** – Типы данных с плавающей запятой в C++

| Тип | Размер (байты) | Биты экспоненты | Биты мантиссы |
|---|---|---|---|
| `float` | 4 | 8 | 23 |
| `double` | 8 | 11 | 52 |
| `long double` | 10 | 15 | 64 |

Для доступа к битовому представлению числа можно использовать ’reinterpret_cast’, ’std::bit_cast’ (в C++ 20) или структуры с битовыми полями, помня об обратном порядке полей на little-endian архитектурах.

```
#include <cstdint>

// Order is reversed for little-endian systems
struct DoubleBits {
    uint64_t mantissa : 52;
    uint64_t exponent : 11;
    uint64_t sign : 1;
};

double d = 1.234;
// In C++20, prefer std::bit_cast
DoubleBits bits = *reinterpret_cast<DoubleBits*>(&d);
// Now bits.sign, bits.exponent, bits.mantissa can be accessed
```

**Листинг 1** – Доступ к битам double через структуру с битовыми полями

**Итоги раздела**

- Числа с **фиксированной точкой** просты и быстры, но имеют ограниченный диапазон.
- Стандарт **IEEE 754** определяет представление чисел с **плавающей запятой** (знак, экспонента, мантисса), позволяя работать с огромным диапазоном значений.
- Существуют специальные значения: **денормализованные числа**, **бесконечности** и **NaN**.
- Арифметика с плавающей запятой неточна и требует аккуратного обращения для минимизации погрешностей.

# 3 Основы многопоточности

Многопоточность — это способ организации вычислений, при котором программа состоит из нескольких потоков управления, выполняющихся параллельно.

## 3.1 Процессы и потоки

**Определение: Процесс и Поток**

**Процесс** — это экземпляр программы, выполняемый операционной системой. Процессы сильно изолированы друг от друга: у каждого своё адресное пространство, свои файловые дескрипторы и т.д. Коммуникация между ними сложна (требует IPC: pipes, shared memory).
**Поток** (thread) — это минимальная единица исполнения внутри процесса. Все потоки одного процесса разделяют общее адресное пространство, файловые дескрипторы и другие ресурсы. Это делает коммуникацию между ними простой, но создаёт проблемы с синхронизацией.

В C++ для создания потоков используется класс ‘std::thread‘.

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

void worker_function() {
    std::cout << "Worker thread is running.\n";
}

int main() {
    std::thread t(worker_function); // Create and start a new thread
    // ... main thread continues execution ...
    t.join(); // Wait for the worker thread to finish
    return 0;
}
```

**Листинг 2** – Создание и запуск потока в C++

## 3.2 Синхронизация и доступ к общей памяти

Основная сложность в многопоточном программировании — корректная работа с общими данными. Когда несколько потоков одновременно читают и пишут в одну и ту же ячейку памяти, возникает **состояние гонки (race condition)**.

Проблема усугубляется тем, что и компилятор, и процессор могут переупорядочивать операции для оптимизации. В однопоточной программе это незаметно, но в многопоточной может привести к непредсказуемому поведению.

**Примечание**

Одновременный доступ (хотя бы одна из операций — запись) к обычной (неатомарной) переменной из разных потоков без синхронизации является **неопределённым поведением** (неопределённое поведение (Undefined Behavior) (UB)) в C++.

## 3.3 Атомарные операции (`std::atomic`)

Для безопасной работы с разделяемыми переменными без блокировок используются атомарные типы (‘std::atomic‘).

**Определение: Атомарная операция**

Это операция, которая выполняется как единое, неделимое целое. Никакой другой поток не может наблюдать её в промежуточном состоянии.

Например, операция ‘value++‘ неатомарна. Она состоит из трёх шагов: чтение, инкремент, запись. Другой поток может вмешаться между этими шагами. Атомарная операция ‘value.fetch_add(1)‘ выполняет то же самое, но гарантированно неделимо.

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

std::atomic<int> counter = 0;

void increment() {
    for (int i = 0; i < 1000000; ++i) {
        counter.fetch_add(1); // Atomic increment
    }
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(increment);
    }
    for (auto& t : threads) {
        t.join();
    }
    // counter will be exactly 10,000,000
    return 0;
}
```

**Листинг 3** – Безопасный инкремент с помощью std::atomic

## 3.4 Примитивы блокирующей синхронизации

Когда требуется защитить не одну переменную, а целый блок кода (критическую секцию), используются блокирующие примитивы.

### 3.4.1 Мьютекс (std::mutex)

Мьютекс обеспечивает взаимное исключение. Только один поток может владеть мьютексом в любой момент времени.

- ‘mutex.lock()‘: Захватывает мьютекс. Если он уже захвачен другим потоком, текущий поток блокируется («засыпает») до его освобождения.
- ‘mutex.unlock()‘: Освобождает мьютекс.

Для безопасного использования рекомендуется RAII-обёртка ‘std::lock_guard‘, которая автоматически вызывает ‘unlock‘ в своём деструкторе.

### 3.4.2 Спинлок (Spinlock)

Альтернатива мьютексу, реализованная на атомарных операциях. Вместо блокировки потока (передачи управления ядру), спинлок входит в цикл активного ожидания (busy-

wait), постоянно проверяя, не освободился ли ресурс.

- **Эффективен**, когда ожидание короткое (меньше, чем накладные расходы на переключение контекста потока).
- **Расточителен**, если ожидание долгое, так как впустую тратит процессорное время.

### 3.4.3 Условные переменные (`std::condition_variable`)

Позволяют одному потоку ждать, пока не выполнится некоторое условие, которое устанавливается другим потоком. Они работают в паре с мьютексом.

- ‘cv.wait(lock, predicate)‘: Атомарно освобождает мьютекс (‘lock‘) и блокирует поток до тех пор, пока другой поток не вызовет ‘notify‘ и ‘predicate‘ не станет истинным. Перед выходом из ‘wait‘ мьютекс снова захватывается.
- ‘cv.notify_one()‘: «Будит» один из ожидающих потоков.

Использование предиката в ‘wait‘ обязательно для борьбы с «ложными пробуждениями» (spurious wakeups).

**Итоги раздела**

- Потоки разделяют память, что требует **синхронизации** для избежания гонок и UB.
- **Атомарные операции** (‘std::atomic‘) обеспечивают неделимый доступ к одиночным переменным.
- **Мьютекс** (‘std::mutex‘) защищает критические секции кода, блокируя потоки при ожидании.
- **Условные переменные** (‘std::condition_variable‘) позволяют потокам эффективно ожидать выполнения произвольных условий.

# 4 Классическая проблема: обедающие философы

Эта задача иллюстрирует проблему взаимоблокировка в системах с разделяемыми ресурсами.

## 4.1 Постановка задачи

Пять философов сидят за круглым столом. Перед каждым — тарелка спагетти, а между каждыми двумя соседними философами лежит по одной вилке. Итого 5 философов и 5 вилок.

Каждый философ попеременно то думает, то ест. Чтобы поесть, ему нужны обе вилки: левая и правая.

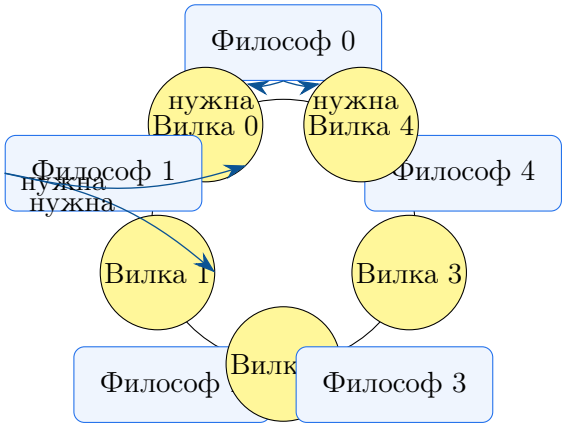


**Рис. 3** – Схема расположения философов и вилок. Каждому философу для еды нужны две соседние вилки.

## 4.2 Взаимоблокировка (Deadlock)

Рассмотрим наивный алгоритм поведения для каждого философа:

1. Взять левую вилку.
2. Взять правую вилку.
3. Поесть.
4. Положить левую вилку.
5. Положить правую вилку.
6. Подумать.

**Примечание**

Что произойдёт, если все философы одновременно решат поесть и каждый возьмёт свою левую вилку? Каждый из них будет вечно ждать, пока его сосед справа освободит правую для него вилку. Но сосед справа тоже ждёт. Возникает **цикл ожидания**, и ни один из потоков не может продолжить выполнение. Это и есть взаимоблокировка.

Проблема взаимоблокировка — одна из фундаментальных в многопоточном программировании. Для её решения существуют различные подходы, например, нарушение одного из условий возникновения взаимоблокировки (в данном случае, введение строгого

порядка захвата ресурсов: например, все философы сначала берут вилку с меньшим номером, а потом с большим).