Курс: *Архитектура компьютера и ОС*

Лекция 1: Введение в ОС и системные вызовы

Лектор: Олег

Оглавление

Глава 1

1 Лекция

1.1 Введение и организационные моменты

Этот курс посвящён изучению пользовательской части Операционная система (ОС) и архитектуры компьютера. Основная цель — понять, как программы выполняют свои действия на низком уровне, «под капотом» стандартных библиотечных функций. Курс рассчитан на один семестр и является обязательным, с возможностью выбрать продолжение во втором семестре.

1.1.1 Формула оценки

Итоговая оценка за курс формируется по следующей формуле:

$$\text{Оценка} = \min\left(10, 0.6 \cdot O_{\text{дз}} + 0.2 \cdot O_{\text{кр}} + 0.2 \cdot O_{\text{экз}} + 0.1 \cdot O_{\text{сем}}\right) \tag{1.1.1}$$

где:

- $O_{\text{дз}}$ — оценка за домашние задания.
- $O_{\text{кр}}$ — оценка за контрольные работы.
- $O_{\text{экз}}$ — оценка за экзамен.
- $O_{\text{сем}}$ — оценка за работу на семинарах.

Примечание

Сумма весовых коэффициентов в формуле (1.1.1) равна 1.1. Это означает, что с учётом бонусов за домашние задания можно набрать более 10 баллов, но итоговая оценка ограничивается 10 баллами.

1.1.2 Работа с домашними заданиями

Домашние задания будут выдаваться примерно раз в неделю со сроком выполнения 1–2 недели. Дедлайны «мягкие»: баллы за задание начинают убывать постепенно и достигают 10–20% от первоначальной стоимости через 3 недели после выдачи. Это сделано для того, чтобы студенты не жертвовали сном ради сдачи заданий в последний момент. Все задания будут выполняться в среде GEDLab.

1.2 Зачем нужна операционная система?

Рассмотрим простейшую программу на C++, которая считывает два числа и выводит их сумму (листинг 1.1).

```cpp
#include <iostream>

int main() {
    int a, b;
    std::cin >> a >> b;
    std::cout << a + b;
    return 0;
}
```

Листинг 1.1 – Программа для сложения двух чисел

На первый взгляд, все операции ввода-вывода выполняются благодаря библиотеке `iostream`. Однако в самом языке C++ нет встроенных механизмов для прямого взаимодействия с устройствами, такими как экран или клавиатура. Как же тогда текст появляется на мониторе?

1.2.1 Проблема прямого доступа к оборудованию

Представим модель, в которой программа напрямую взаимодействует с аппаратными компонентами компьютера: Центральный процессор (CPU), Оперативная память (RAM), жестким диском, сетевой картой и т.д. (рис. 1.1).

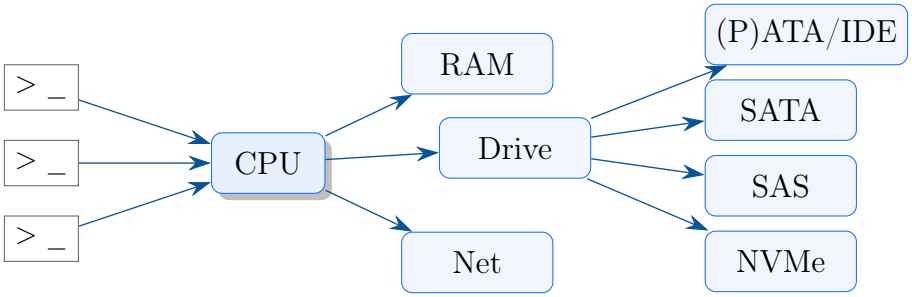


Рис. 1.1 – Модель прямого взаимодействия программы с оборудованием

Такая модель порождает две ключевые проблемы:

1. **Сложность и непереносимость.** Существует множество протоколов для взаимодействия с одним и тем же типом устройств. Например, для работы с дисками программа должна была бы поддерживать интерфейсы PATA, SATA, SAS, NVMe и другие. Аналогично, каждый производитель сетевых карт может предлагать свой уникальный протокол. Чтобы программа работала на разных компьютерах, ей пришлось бы реализовывать поддержку всех этих интерфейсов, что практически невозможно.
2. **Разделение ресурсов.** В современных системах одновременно запущены сотни и тысячи программ, в то время как количество ядер CPU ограничено единицами или десятками. Необходимо эффективно распределять процессорное время и другие ресурсы (память, доступ к дискам) между всеми программами. При прямом доступе программы к оборудованию сделать это было бы крайне затруднительно.

1.2.2 Решение: операционная система как абстракция

Для решения этих проблем была придумана ОС.

Определение: Операционная система

ОС — это программный слой, который выступает посредником между пользовательскими программами и аппаратным обеспечением компьютера.

ОС решает обе проблемы:

- Она **предоставляет унифицированный интерфейс** для работы с оборудованием. Программа работает не с конкретным жестким диском, а с абстракцией «файловой системы». ОС сама берёт на себя реализацию всех низкоуровневых протоколов.
- Она **управляет ресурсами**. ОС решает, какой программе и на какое время предоставить CPU, распределяет память, организует доступ к устройствам, предотвращая конфликты.

Эта модель показана на рис. 1.2.

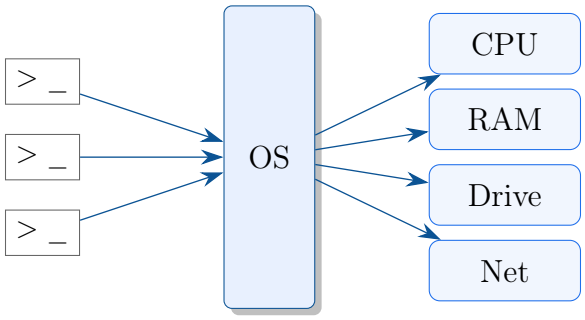


Рис. 1.2 – Операционная система как посредник

Итоги раздела

- Прямое взаимодействие программ с оборудованием сложно, непереносимо и не позволяет эффективно разделять ресурсы.
- ОС решает эти проблемы, предоставляя программам абстракции (файлы, сокеты) и управляя доступом к аппаратуре.

1.3 Работа с памятью в C++: краткое повторение

Взаимодействие с ОС в значительной степени происходит через память. Программа записывает данные в свою область памяти и затем просит ОС что-то с этими данными сделать. Поэтому важно освежить знания о модели памяти в C++.

1.3.1 Линейно адресуемая память и указатели

Память можно представить как большой массив байтов, где у каждого байта есть уникальный числовой адрес. Это называется моделью линейно адресуемой памяти.

Для работы с памятью используются **указатели** — переменные, которые хранят адрес. Разыменование указателя (`*ptr`) означает обращение к данным, лежащим по этому адресу. Тип указателя определяет, сколько байт будет прочитано и как они будут интерпретированы. Например, указатель типа `int64_t*` при разыменовании прочитает 8 байт и представит их как 64-битное целое число.

1.3.2 Динамическая память

Иногда память нужно выделить так, чтобы она «пережила» функцию, в которой была создана. Для этого используется динамическое выделение памяти в «куче» (heap).

```
// Allocating a single object
// Operator `new` allocates sizeof(T) bytes and constructs an object
T* ptr = new T{};

// Deleting an object and freeing memory
delete ptr;

// Allocating an array of 10 objects
T* arr = new T[10];

// Deleting all objects in the array and freeing memory
delete[] arr;
```

Листинг 1.2 – Работа с динамической памятью

Примечание

Важно соблюдать парность операторов: память, выделенную через `new`, нужно освобождать через `delete`. Память, выделенную через `new[]`, — через `delete[]`. Нарушение этого правила приводит к неопределённому поведению.

1.3.3 Разделение аллокации и конструирования

Оператор `new T` на самом деле выполняет две операции:

1. Выделение «сырой» (неинициализированной) памяти нужного размера.
2. Конструирование объекта типа `T` в этой памяти.

Эти шаги можно выполнить раздельно.

```
// 1. Allocate sizeof(T) raw bytes. operator new returns void*
void* raw_ptr = operator new(sizeof(T));

// 2. Construct an object of type T at the given address (placement new)
T* ptr = new (raw_ptr) T{};
```

```

// --- object `ptr` is ready to use ---

// 3. Explicitly call the destructor to destroy the object
ptr->~T();

// 4. Free the raw memory
operator delete(raw_ptr);
```

Листинг 1.3 – Явное управление памятью и объектами

Такой подход даёт больше контроля, но требует аккуратного ручного управления временем жизни объекта и памяти.

1.3.4 Арифметика указателей

В C++ арифметика указателей типизирована. Прибавление к указателю `ptr` единицы (`ptr + 1`) сдвигает его адрес не на 1 байт, а на `sizeof(*ptr)` байт, то есть к адресу следующего элемента в массиве.

- `ptr[i]` эквивалентно `*(ptr + i)`.
- Разность двух указателей одного типа `ptr1 - ptr2` даёт количество элементов (а не байт) между ними.

1.4 Взаимодействие с ОС: системные вызовы

Теперь, когда мы освежили знания о памяти, перейдём к основному механизму взаимодействия программы с ОС.

Определение: Системный вызов

Системный вызов — это основной интерфейс между пользовательскими программами и ядром ОС. Программа использует Системный вызов, чтобы попросить ядро выполнить действие, которое она не может выполнить сама (например, работать с файлом или сетью).

Рассмотрим два базовых системных вызова для ввода-вывода: `read` и `write`.

1.4.1 Системный вызов read

Функция `read` читает данные из источника, идентифицируемого Файловый дескриптор, в буфер.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

// Example
char buf[10];
// Read from stdin (fd=0) into buf, at most 9 bytes
ssize_t bytes_read = read(0, buf, 9);

if (bytes_read == -1) {
    // Error occurred
} else if (bytes_read == 0) {
    // End of input (EOF)
```

```
} else {
    // Successfully read `bytes_read` bytes
}
```

Листинг 1.4 – Сигнатура и использование read

Аргументы:

- `int fd`: Файловый дескриптор источника данных. По соглашению, 0 — это Стандартный поток ввода.
- `void *buf`: Указатель на буфер, куда будут записаны данные.
- `size_t count`: Максимальное количество байт для чтения.

Возвращаемое значение (`ssize_t`):

- Положительное число: количество успешно прочитанных байт. **Важно:** `read` не гарантирует, что прочитает ровно `count` байт, даже если они доступны. Он может прочитать меньше.
- `0`: достигнут конец файла (EOF) или потока. Больше данных для чтения нет.
- `-1`: произошла ошибка. Код ошибки сохраняется в глобальной переменной errno.

1.4.2 Системный вызов write

Функция `write` записывает данные из буфера в приёмник, идентифицируемый Файловый дескриптор.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);

// Example
const char msg[] = "Hello, world!\n";
// Write to stdout (fd=1), strlen(msg) bytes
// We use sizeof(msg) - 1 to exclude the terminating null byte ('\0')
ssize_t bytes_written = write(1, msg, sizeof(msg) - 1);

// Write the same message to stderr (fd=2)
write(2, msg, sizeof(msg) - 1);

if (bytes_written == -1) {
    // Error occurred
}
```

Листинг 1.5 – Сигнатура и использование write

Аргументы:

- `int fd`: Файловый дескриптор приёмника данных. 1 — Стандартный поток вывода, 2 — Стандартный поток ошибок.
- `const void *buf`: Указатель на буфер с данными для записи. Буфер константный, так как `write` его не изменяет.
- `size_t count`: Количество байт для записи.

Возвращаемое значение:

- Положительное число: количество успешно записанных байт. Как и `read`, `write` может записать меньше байт, чем было запрошено (например, если на диске закончилось место).
- `-1`: произошла ошибка, код которой записан в errno.

1.4.3 Обработка ошибок и частичных операций

Поскольку `read` и `write` могут обработать меньше данных, чем запрошено, для надёжной передачи всего объёма данных необходимо использовать циклы.

```cpp
// Writes exactly `count` bytes from `buf` to `fd`.
// Returns `true` on success, `false` on error.
bool WriteAll(int fd, const char* buf, size_t count) {
    size_t written = 0;
    while (written < count) {
        ssize_t res = write(fd, buf + written, count - written);
        if (res == -1) {
            return false; // An error occurred
        }
        written += res;
    }
    return true;
}
```

Листинг 1.6 – Надёжная функция для записи всех данных

Аналогичная функция `ReadAll` должна быть реализована для чтения, но с дополнительной проверкой на возврат 0 (EOF).

Для получения текстового описания ошибки по её коду из errno можно использовать функцию `strerror` из заголовка `<cstring>`.

```cpp
#include <cerrno>
#include <cstring>
#include <iostream>

// ... inside a function
if (!WriteAll(1, "Hello", 5)) {
    // errno is set by the last failed `write` call
    std::cerr << "Error writing data: " << strerror(errno) << std::endl;
    return 1; // Exit with error code
}
```

Листинг 1.7 – Обработка ошибок с выводом сообщения

Итоги раздела

- Системные вызовы — это API операционной системы.
- `read` и `write` — базовые вызовы для неформатированного ввода-вывода.
- Файловые дескрипторы 0, 1, 2 зарезервированы для стандартных потоков stdin, stdout, stderr.
- Всегда проверяйте возвращаемые значения системных вызовов на ошибки (-1) и обрабатывайте частичные операции.
- Для получения информации об ошибке используйте переменную errno.

1.5 Работа с файлами

Стандартные потоки — это лишь частный случай. Основное применение Файловый дескриптор — работа с файлами на диске.

1.5.1 Системные вызовы open и close

Чтобы работать с файлом, его сначала нужно открыть с помощью системного вызова `open`.

Определение: Файловый дескриптор

Файловый дескриптор — это неотрицательное целое число, которое ОС возвращает процессу при открытии файла. Процесс использует этот дескриптор во всех последующих операциях с файлом (`read`, `write`, `close`).

```
#include <fcntl.h> // For flags
#include <unistd.h>

int open(const char *pathname, int flags, ... /* mode_t mode */);
```

Листинг 1.8 – Сигнатура системного вызова open

Аргументы:

- `const char *pathname`: Путь к файлу.
- `int flags`: Флаги, определяющие режим доступа (например, `O_RDONLY` — только для чтения, `O_WRONLY` — только для записи, `O_RDWR` — для чтения и записи). Флаги можно комбинировать с помощью побитового ИЛИ (`|`).
- `mode_t mode`: (Опционально) Права доступа, которые устанавливаются, если файл создаётся с флагом `O_CREAT`.

В случае успеха `open` возвращает новый Файловый дескриптор (обычно наименьший из доступных). В случае ошибки возвращается -1, а errno устанавливается.

После завершения работы с файлом его дескриптор необходимо освободить с помощью системного вызова `close`.

```
#include <unistd.h>

int close(int fd);
```

Листинг 1.9 – Сигнатура системного вызова close

Если не закрывать файлы, это приведёт к утечке ресурсов (файловых дескрипторов), так как их количество для одного процесса ограничено.

1.5.2 Пример чтения из файла

В листинг 1.10 показан полный цикл работы: открытие файла, чтение из него, вывод содержимого в стандартный поток и закрытие.

```
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <cerrno>
#include <cstring>

int main() {
    const char* filename = "output.txt";
    int fd = open(filename, O_RDONLY);
```

```
    if (fd == -1) {
        std::cerr << "Failed to open file " << filename << ": "
                  << strerror(errno) << std::endl;
        return 1;
    }

    char buffer[1024];
    ssize_t bytes_read;

    // Read from file in a loop until EOF
    while ((bytes_read = read(fd, buffer, sizeof(buffer))) > 0) {
        // Write the read data to stdout
        if (!WriteAll(1, buffer, bytes_read)) {
            std::cerr << "Failed to write to stdout: "
                      << strerror(errno) << std::endl;
            close(fd);
            return 1;
        }
    }

    if (bytes_read == -1) {
        std::cerr << "Error reading from file: " << strerror(errno) << std::endl;
    }

    close(fd); // Don't forget to close the file!
    return 0;
}
// Assume WriteAll is defined as in listing 4.4
```

Листинг 1.10 – Чтение из файла и вывод в stdout

Итоги раздела

- Работа с файлом начинается с его открытия вызовом `open`, который возвращает Файловый дескриптор.
- Полученный Файловый дескриптор используется в вызовах `read` и `write` для взаимодействия с файлом.
- После окончания работы файл необходимо закрыть вызовом `close`, чтобы освободить ресурсы.
- Каждый `open` должен иметь парный `close`, подобно паре `new`/`delete`.

Глава 2

2 Лекция

2.1 Взаимодействие с носителями информации

На прошлой лекции мы установили, что программы взаимодействуют с внешним миром через Системный вызов. Сегодня мы продолжим эту тему и углубимся во взаимодействие с файловая система.

2.1.1 Почему не работать с диском напрямую?

Казалось бы, зачем нужна файловая система, если можно работать с жёстким диском напрямую? Тому есть две ключевые причины: сложность Application Programming Interface (интерфейс прикладного программирования) (API) и низкая производительность.

1. **Примитивный интерфейс.** Диск предоставляет очень аскетичное API: он позволяет читать и писать только «сырые» данные по указанным адресам (с такого-то по такой-то байт). В таком интерфейсе отсутствуют высокоуровневые концепции, такие как файлы, директории, права доступа и структура данных.
2. **Особенности производительности.** Жёсткий диск (HDD) — механическое устройство. Он состоит из вращающихся магнитных пластин («блинов») и считывающих головок (рис. 2.1).

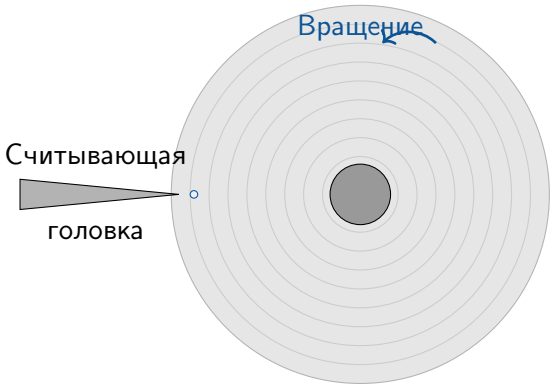


Рис. 2.1 – Упрощённая схема устройства жёсткого диска (HDD)

Скорость вращения современных дисков составляет 5000–7000 оборотов в минуту. Чтобы прочитать данные, необходимо выполнить две операции с большими задержками:

- **Позиционирование головки (seek time):** Механическое перемещение головки к нужной дорожке.
- **Ожидание вращения (rotational latency):** Ожидание, пока нужный сектор на дорожке окажется под головкой.

В среднем, ожидание нужного сектора может занимать до 5 мс. Это означает, что при чтении из случайных мест диска можно выполнить всего около 200 операций в секунду, что на порядки медленнее, чем миллиарды операций, выполняемых процессором. Для эффективной работы данные нужно располагать последовательно, минимизируя перемещения головки, но реализация такой логики — крайне сложная задача.

Определение: Файловая система

Файловая система — это уровень абстракции, предоставляемый операционной системой для организации, хранения и именования данных на носителях информа-

ции. Она скрывает сложности работы с оборудованием и предоставляет удобный и эффективный интерфейс для пользователя и программ.

2.2 Права доступа в Linux

файловая система в Linux представляет собой древовидную структуру из директорий и файлов. Для управления доступом к этим объектам используется модель прав, основанная на пользователях и группах.

2.2.1 Чтение вывода `ls -l`

Команда `ls -l` выводит подробную информацию о файлах и директориях:

```
-rw-rw-r-- 1 arch arch 4 sen 13 11:58 out
drwxr-xr-x 2 arch arch 4096 sen 13 12:00 test
```

Рассмотрим структуру вывода:

- `-rw-rw-r-`: Права доступа.
- `1`: Количество жёстких ссылок.
- `arch`: Пользователь-владелец.
- `arch`: Группа-владелец.
- `4`: Размер в байтах.
- `Сен 13 11:58`: Дата последнего изменения.
- `out`: Имя файла.

Первый символ указывает на тип: `-` для обычного файла, `d` для директории, `l` для символическая ссылка.

2.2.2 Пользователь, группа и остальные

Следующие 9 символов прав доступа делятся на три группы по три:

1. **Для владельца (user):** Права пользователя, которому принадлежит файл.
2. **Для группы (group):** Права для всех пользователей, состоящих в группе, которой принадлежит файл.
3. **Для остальных (others):** Права для всех остальных пользователей.

Каждая тройка состоит из символов `r`, `w`, `x`:

- `r` (read): Право на чтение.
- `w` (write): Право на запись (изменение).
- `x` (execute): Право на исполнение (для программ и скриптов).

Если право отсутствует, на его месте ставится прочерк (`-`).

2.2.3 Команда `chmod`

Для изменения прав доступа используется команда `chmod` (change mode). Она поддерживает два основных синтаксиса: символический и восьмеричный.

Символический синтаксис:

```
# Add execute permission for the user (owner)
chmod u+x filename

# Remove write permission for group and others
chmod go-w filename

# Set permissions: read/write for user, read-only for group/others
chmod u=rw,go=r filename
```

Восьмеричный синтаксис: Права представляются в виде трёх восьмеричных цифр, где каждая цифра — это сумма значений для `r`, `w`, `x`:

- `r` = 4
- `w` = 2
- `x` = 1

Например, `rw-` соответствует $4 + 2 + 0 = 6$, а `r-x` — $4 + 0 + 1 = 5$.

```
# Corresponds to rw-rw-r-- (664)
chmod 664 out

# Corresponds to rwxr-xr-x (755)
chmod 755 script.sh
```

Примечание

Права для директорий. Права `rwx` для директорий имеют особый смысл:

- `r`: Позволяет просмотреть список файлов в директории (выполнить `ls`).
- `w`: Позволяет создавать, удалять и переименовывать файлы в директории.
- `x`: Позволяет войти в директорию (сделать `cd`) и получить доступ к файлам внутри неё (при наличии прав на сами файлы).

2.3 Файловые дескрипторы и системные вызовы

Для работы с файлами из программы операционная система предоставляет набор Системный вызов. Ключевой абстракцией здесь является Файловый дескриптор.

Определение: Файловый дескриптор

Файловый дескриптор — это неотрицательное целое число, которое процесс использует для идентификации открытого файла или другого ресурса ввода-вывода. Вместо того чтобы каждый раз передавать ядру полный путь к файлу, программа один раз вызывает `open` и получает Файловый дескриптор, который затем использует в вызовах `read`, `write`, `close` и др..

По умолчанию каждый процесс в Linux при запуске имеет три открытых Файловый дескриптор:

- `0` — стандартный поток ввода (*stdin*).
- `1` — стандартный поток вывода (*stdout*).

- 2 — стандартный поток ошибок (*stderr*).

2.3.1 Системный вызов `open`

Для открытия или создания файла используется Системный вызов `open`.

```c
#include <fcntl.h>
#include <sys/stat.h>

int open(const char *path, int flags, mode_t mode);
```

- `path`: Путь к файлу.
- `flags`: Битовая маска, определяющая режим доступа.
- `mode`: Права доступа, которые будут установлены, если файл создаётся.

Функция возвращает новый Файловый дескриптор или `-1` в случае ошибки.

Основные флаги (`flags`):

- `O_RDONLY`, `O_WRONLY`, `O_RDWR`: Открыть только для чтения, только для записи или для чтения и записи. Один из этих флагов должен быть указан.
- `O_CREAT`: Создать файл, если он не существует.
- `O_EXCL`: Использовать вместе с `O_CREAT`. Вызов завершится ошибкой, если файл уже существует. Это позволяет атомарно создать файл и убедиться в его отсутствии до вызова.
- `O_APPEND`: Все операции записи будут производиться в конец файла.
- `O_TRUNC`: Если файл существует и открывается на запись, его содержимое усекается до нуля байт.

Создание файла и `umask`

При создании файла (с флагом `O_CREAT`) его итоговые права доступа определяются формулой:

$$\text{final\_mode} = \text{mode} \ \& \ \sim \text{umask}$$

где `mode` — это права, переданные в `open`, а `umask` — это маска процесса. Umask определяет, какие права доступа нужно «выключить» по умолчанию. Например, если `umask` равна `0002` ($---w----$), то у всех создаваемых файлов будет отбираться право на запись для «остальных».

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

int main() {
    // Create "file" if it does not exist.
    // Error if it already exists.
    // Permissions: read for all (0444).
    // Mode: read and write for our process.
    int fd = open(
        "file",
        O_RDWR | O_CREAT | O_EXCL,
        S_IRUSR | S_IRGRP | S_IROTH /* 0444 */
```

```
    );

    if (fd == -1) {
        // handle error
        return 1;
    }

    // ... work with the file ...

    close(fd);
    return 0;
}
```

Листинг 2.1 – Пример использования open

Примечание

Открытый файл — это ресурс, который ядро выделяет для процесса. Как и любую другую выделенную память, его необходимо освобождать. Для этого используется Системный вызов `close(int fd)`. Если этого не делать, произойдёт утечка ресурсов (файловых дескрипторов).

2.3.2 Структура открытого файла в ядре

С каждым открытым Файловый дескриптор ядро ассоциирует структуру, содержащую как минимум:

- **Флаги открытия:** Режим, в котором файл был открыт (`O_RDONLY` и т.д.).
- **Текущее смещение:** Позиция в файле, с которой будет происходить следующая операция чтения/записи.
- **Ссылка на inode:** Указатель на структуру файла в файловая система.

Важно, что права доступа проверяются только один раз — во время вызова `open`. Все последующие операции с Файловый дескриптор (`read`, `write`) не требуют повторной проверки прав.

2.3.3 Другие важные системные вызовы

`lseek`: Изменение смещения

Системный вызов `lseek` позволяет изменить текущее смещение в файле.

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

- `fd`: Файловый дескриптор, для которого меняется смещение.
- `offset`: Значение смещения в байтах.
- `whence`: Точка отсчёта:
  - `SEEK_SET`: смещение отсчитывается от начала файла.
  - `SEEK_CUR`: смещение отсчитывается от текущей позиции.
  - `SEEK_END`: смещение отсчитывается от конца файла.

С помощью `lseek` можно перемещаться за конец файла. Если после такого перемещения произвести запись, то пространство между старым концом файла и новой позицией записи будет заполнено нулевыми байтами, создавая разреженный файл.

`dup` и `dup2`: Копирование файловых дескрипторов

Эти вызовы создают копию Файловый дескриптор.

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

`dup` создаёт копию `oldfd`, используя первый свободный номер Файловый дескриптор. `dup2` создаёт копию `oldfd` с конкретным номером `newfd`. Если `newfd` уже был открыт, он атомарно закрывается.

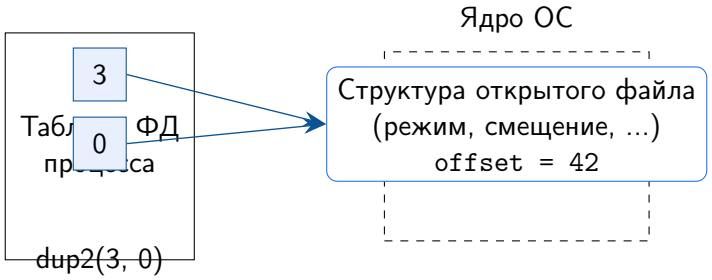


Рис. 2.2 – Схема работы `dup2`. Оба дескриптора (старый и новый) указывают на одну и ту же структуру открытого файла в ядре и разделяют общее смещение.

Ключевой особенностью является то, что новый и старый Файловый дескриптор ссылаются на одну и ту же запись в таблице открытых файлов ядра (рис. 2.2). Это означает, что они **разделяют общее смещение**: изменение позиции через один Файловый дескриптор немедленно отражается на другом.

`pipe`: Создание каналов

Системный вызов `pipe` создаёт однонаправленный канал (pipe) для межпроцессного взаимодействия.

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

Вызов создаёт пару связанных Файловый дескриптор и помещает их в массив `pipefd`:

- `pipefd[0]`: Файловый дескриптор для чтения из канала.
- `pipefd[1]`: Файловый дескриптор для записи в канал.

Данные, записанные в `pipefd[1]`, можно прочитать из `pipefd[0]` в том же порядке (FIFO).

- Файловый дескриптор — это числовой идентификатор открытого ресурса.
- `open` открывает/создаёт файл и возвращает Файловый дескриптор.
- `lseek` позволяет перемещаться по файлу, изменяя смещение.

- `dup2` копирует Файловый дескриптор, что является основой для перенаправления ввода-вывода.
- `pipe` создаёт пару Файловый дескриптор для однонаправленной передачи данных между процессами.
- Все открытые ресурсы должны быть закрыты с помощью `close`.

2.4 Практика: Перенаправление ввода-вывода

Одной из самых мощных возможностей, которую даёт `dup2`, является перенаправление стандартных потоков ввода-вывода. Рассмотрим программу, которая читает число из *stdin* и выводит его инкремент в *stdout*.

```cpp
#include <iostream>

int main() {
    int a;
    std::cin >> a;
    std::cout << a + 1 << std::endl;
    return 0;
}
```

Листинг 2.2 – Программа с простым вводом-выводом

Мы можем перехватить её ввод и вывод, не изменяя исходный код. Для этого нужно открыть файлы для чтения и записи, а затем с помощью `dup2` подменить стандартные Файловый дескриптор (0 и 1) нашими.

```cpp
#include <fcntl.h>
#include <unistd.h>
#include <string_view>
#include <iostream>

// Utility for error handling
[[noreturn]] void Fail(std::string_view msg) {
    perror(msg.data());
    std::abort();
}

void Redirect() {
    // Open a file for reading
    int fin = open("input.txt", O_RDONLY);
    if (fin == -1) {
        Fail("open input.txt");
    }

    // Open a file for writing, create if it does not exist
    int fout = open("out.txt", O_WRONLY | O_CREAT, 0666);
    if (fout == -1) {
        Fail("open out.txt");
    }

    // Replace stdin (fd 0) with our file fin
    if (dup2(fin, 0) == -1) {
```

```
        Fail("dup2 fin -> 0");
    }

    // Replace stdout (fd 1) with our file fout
    if (dup2(fout, 1) == -1) {
        Fail("dup2 fout -> 1");
    }

    // The original fds fin and fout can be closed,
    // as their copies now exist as fd 0 and 1.
    close(fin);
    close(fout);
}

int main() {
    Redirect();

    int a;
    std::cin >> a; // Now reads from input.txt
    std::cout << a + 1 << std::endl; // Now writes to out.txt

    return 0;
}
```

Листинг 2.3 – Функция перенаправления ввода-вывода

Если в файле `input.txt` будет число 123, то после выполнения программы в файле `out.txt` появится 124. Программа `main` ничего не знает о подмене; для неё `std::cin` и `std::cout` продолжают работать со стандартными Файловый дескриптор 0 и 1, но ядро теперь направляет эти операции в файлы.

Примечание

Проблемы буферизации. Стандартные потоки C++ (и C) буферизуют вывод для повышения производительности. Данные не отправляются ядру немедленно, а накапливаются во внутреннем буфере. Сброс буфера (flush) происходит:

- При его заполнении.
- При выводе специального символа, например, при использовании `std::endl`.
- При чтении из `std::cin` (обычно *stdout* сбрасывается).
- При завершении программы.

Интересно, что `libc` может менять свою стратегию буферизации. При выводе в терминал буфер часто сбрасывается при каждом символе новой строки (`'\n'`). При выводе в файл (который не является интерактивным устройством) буферизация становится полной, и сброс происходит только при заполнении буфера или явном вызове `flush`. Это может приводить к неожиданному поведению, когда вывод, видимый в терминале, не сразу появляется в файле при перенаправлении.

2.5 Работа с директориями

Для просмотра содержимого директории используются функции из стандартной библиотеки C, которые являются обёрткой над соответствующими Системный вызов.

```c
#include <dirent.h>

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

Листинг 2.4 – Интерфейс для чтения директорий

- `opendir` открывает директорию и возвращает указатель на структуру `DIR`, которая используется для дальнейших операций.
- `readdir` при каждом вызове возвращает указатель на структуру `dirent`, описывающую следующий элемент в директории. Когда элементы заканчиваются или происходит ошибка, возвращается `NULL`.
- `closedir` закрывает директорию.

Структура `dirent` содержит как минимум два поля: `d_name` (имя файла) и `d_type` (тип файла, например, `DT_REG` для файла, `DT_DIR` для директории).

```c
#include <dirent.h>
#include <stdio.h>
#include <errno.h>

int main() {
    DIR* dir = opendir(".");
    if (!dir) {
        perror("opendir failed");
        return 1;
    }

    errno = 0; // To distinguish end-of-stream from error
    struct dirent* entry;
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }

    if (errno != 0) {
        perror("readdir failed");
    }

    closedir(dir);
    return 0;
}
```

Листинг 2.5 – Пример простой реализации ls

Примечание

В каждой директории в Linux есть два специальных вхождения:

- .: Ссылка на саму директорию.
- ..: Ссылка на родительскую директорию.

Они также будут перечислены при вызове `readdir`.

Глава 3

3 Лекция

3.1 Дополнительные инструменты для работы с файловой системой

На прошлом занятии мы рассмотрели основы работы с файловой системой. Сегодня мы завершим эту тему, изучив несколько оставшихся, но важных инструментов, которые могут пригодиться в практических задачах.

3.1.1 Новые флаги для системного вызова `open`

Системный вызов `open` имеет несколько полезных флагов, которые мы не обсуждали ранее.

- `O_TRUNC`: Этот флаг позволяет при открытии файла немедленно обрезать его размер до нуля. Это удобная альтернатива последовательному вызову `open` и `ftruncate`, если содержимое файла нужно полностью перезаписать.
- `O_PATH`: Позволяет получить файловый дескриптор, который ссылается не на сам файл, а на его путь в файловой системе. Такой дескриптор имеет ограниченное применение (например, из него нельзя читать или в него писать), но он полезен для передачи в другие системные вызовы, такие как `fstat`, для получения информации об объекте файловой системы (включая директории), не открывая его для операций ввода-вывода.
- `O_NOFOLLOW`: Если путь, передаваемый в `open`, является символической ссылкой, то с этим флагом вызов не будет переходить по ней, а вернёт ошибку. Это важно для безопасности, чтобы избежать работы с непредусмотренным файлом.

3.1.2 Получение метаданных о файлах: семейство `stat`

Для получения подробной информации о файле или директории используется семейство системных вызовов `stat`. Они заполняют структуру `struct stat`, содержащую метаданные об объекте.

```c
#include <sys/stat.h>

int stat(const char* path, struct stat* statbuf);
int lstat(const char* path, struct stat* statbuf);
int fstat(int fd, struct stat* statbuf);
```

Листинг 3.1 – Function signatures of the stat family

Ключевые различия между вызовами:

- `stat`: Принимает путь к файлу. Если путь указывает на символическую ссылку, `stat` переходит по ней и возвращает информацию о файле, на который она указывает.
- `lstat`: Аналогичен `stat`, но **не** переходит по символическим ссылкам. Вместо этого он возвращает информацию о самой ссылке.
- `fstat`: Принимает файловый дескриптор, полученный ранее через `open`.

Структура `struct stat` содержит множество полезных полей:

- `st_mode`: Тип файла (обычный файл, директория, символическая ссылка и т.д.) и права доступа к нему (чтение, запись, исполнение для владельца, группы и остальных).
- `st_uid` и `st_gid`: ID пользователя и группы-владельца файла.
- `st_size`: Размер файла в байтах.
- `st_blocks`: Количество дисковых блоков, занимаемых файлом.

- `st_atim`, `st_mtim`, `st_ctim`: Временные метки последнего доступа, последней модификации содержимого и последней модификации метаданных соответственно.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

// ...

int fd = open(path, O_RDONLY | O_PATH | O_NOFOLLOW);
if (fd == -1) { /* handle error */ }

struct stat stats;
if (fstat(fd, &stats) == -1) { /* handle error */ }

close(fd);
if (S_ISDIR(stats.st_mode)) {
    // Directory
} else if (S_ISLNK(stats.st_mode)) {
    // Symbol link
} else if (S_ISREG(stats.st_mode)) {
    // Just file
}
```

Листинг 3.2 – Example of using fstat to determine the object type

В этом примере используется флаг `O_PATH`, чтобы безопасно получить дескриптор для проверки типа объекта, не открывая его для полноценной работы.

3.2 Управление памятью: виртуальная адресация

3.2.1 Проблема модели линейной памяти

Мы привыкли думать о памяти как о большом непрерывном массиве байтов. Однако эта модель не соответствует действительности. Проведём простой эксперимент: создадим две переменные — одну в куча (через `new`), а другую на стек (локальная переменная) — и выведем их адреса.

```
#include <iostream>

int main() {
    int* heap_var = new int(10);
int stack_var = 20;

    std::cout << "Heap address: " << (void*)heap_var << std::endl;
std::cout << "Stack address: " << (void*)&stack_var << std::endl;

    long long diff = (long long)&stack_var - (long long)heap_var;
std::cout << "Difference (bytes): " << diff << std::endl;
    // On a 64-bit system, the difference can be tens of terabytes

    delete heap_var;
return 0;
}
```

Листинг 3.3 – Comparing stack and heap addresses

Разница между этими адресами может составлять десятки терабайт, что очевидно превышает объём физической оперативной памяти любого современного компьютера. Это наблюдение доказывает, что адреса, с которыми мы работаем в программе, не являются прямыми физическими адресами.

3.2.2 Виртуальная и физическая память

Для решения проблемы изоляции и безопасности процессов операционные системы вводят абстракцию — виртуальная память.

Определение: Виртуальная и физическая память

- **физическая память** — это реальные микросхемы оперативной памяти (RAM) в компьютере. Её адреса последовательны и ограничены её физическим объёмом.
- **виртуальная память** — это логическое адресное пространство, которое ОС предоставляет каждому процессу. Каждый процесс «видит» свой собственный, изолированный массив памяти, начинающийся с нуля. Адреса в этом пространстве называются **виртуальными**.

Процессор с помощью специального модуля (MMU — Memory Management Unit) и при содействии операционной системы преобразует виртуальные адреса в физические при каждом обращении к памяти. Это преобразование прозрачно для программиста.

3.2.3 Страничная организация памяти

Преобразование адресов происходит не для каждого байта в отдельности, а для блоков памяти фиксированного размера, называемых **страница памяти**.

Примечание

На большинстве современных систем (x86-64) размер страницы составляет 4 килобайта (4096 bytes, или $0x1000$ в шестнадцатеричной системе). Узнать точный размер страницы в системе можно с помощью вызова `sysconf(_SC_PAGESIZE)`.

Операционная система поддерживает для каждого процесса таблицу страниц, которая устанавливает соответствие между страницами виртуальной и физической памяти.

При обращении к адресу, например, `0x2345`:

1. Процессор разделяет его на номер страницы и смещение. Для страниц размером $0x1000$ адрес `0x2345` — это смещение `0x345` внутри страницы `2`.
2. С помощью таблицы страниц находится физический фрейм, соответствующий виртуальной странице `2` (на рис. 3.1 это фрейм A).
3. Процессор обращается к физической памяти по адресу, равному начальному адресу фрейма A плюс смещение `0x345`.

Итоги раздела

Виртуальная память обеспечивает изоляцию процессов, позволяет программам работать с большим адресным пространством, чем доступно физической памяти,

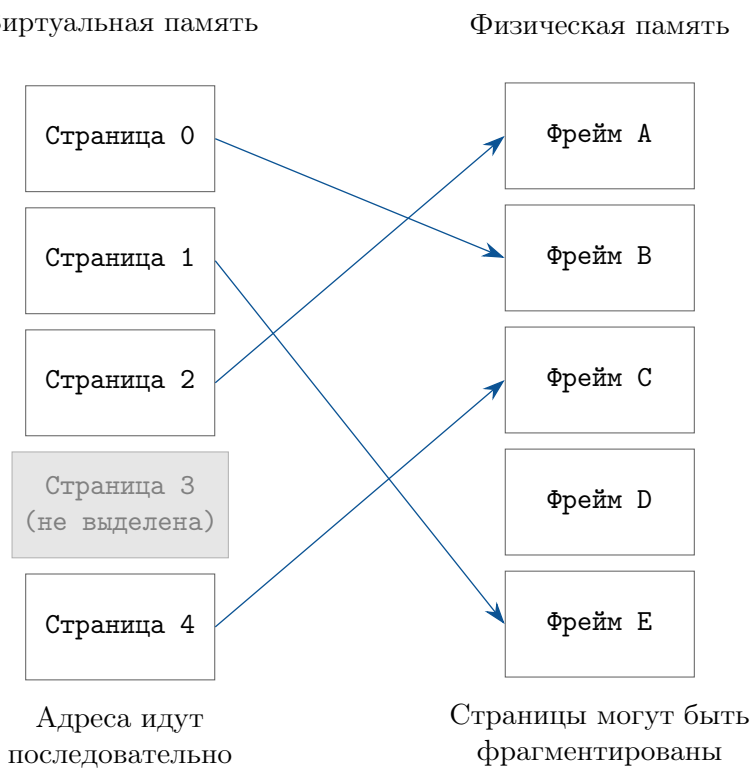


Рис. 3.1 – Схема отображения виртуальных страниц на физические фреймы памяти. Две соседние виртуальные страницы не обязательно отображаются в соседние физические.

и упрощает управление памятью для ОС. Это достигается за счёт постраничного отображения виртуальных адресов на физические.

3.3 Системные вызовы для управления памятью: mmap

Для управления виртуальным адресным пространством процесса в POSIX-системах используется системный вызов mmap и его пара munmap.

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

Листинг 3.4 – Signatures of mmap and munmap

mmap — это мощный, но сложный инструмент, который выполняет две основные функции:

1. **Анонимное отображение**: выделение новых страниц оперативной памяти для процесса.
2. **Файловое отображение**: отображение содержимого файла (или его части) в виртуальное адресное пространство процесса.

3.3.1 Аргументы и флаги mmap

Рассмотрим ключевые параметры mmap:

- `addr`: Желаемый стартовый адрес для отображения. Обычно передаётся `nullptr`, чтобы ОС сама выбрала подходящий адрес.
- `length`: Размер отображаемой области в байтах.
- `prot` (protection): Права доступа к памяти.
  - `PROT_READ`: память можно читать.
  - `PROT_WRITE`: в память можно писать.
  - `PROT_EXEC`: содержимое памяти можно исполнять как код.
  - `PROT_NONE`: к памяти нет доступа.
- `flags`: Определяют тип и поведение отображения.
  - `MAP_SHARED` или `MAP_PRIVATE`: Один из этих флагов обязателен. `MAP_SHARED` означает, что изменения, сделанные в памяти, будут видны другим процессам, отображающим тот же объект, и (в случае файла) будут записаны обратно в файл. `MAP_PRIVATE` создаёт copy-on-write отображение: изменения видны только текущему процессу и не затрагивают исходный файл.
  - `MAP_ANONYMOUS`: Создаёт анонимное отображение. Память инициализируется нулями и не связана ни с каким файлом. При использовании этого флага аргумент `fd` должен быть `-1`.
  - `MAP_FIXED`: Требует от ОС использовать точно адрес, указанный в `addr`. Это опасный флаг, так как он может без предупреждения перезаписать существующие отображения.
- `fd`, `offset`: Файловый дескриптор и смещение от начала файла для файловых отображений.

В случае успеха `mmap` возвращает указатель на начало выделенной области. В случае ошибки — `MAP_FAILED`.

3.3.2 Примеры использования `mmap`

Анонимное отображение

Это основной способ, которым аллокаторы (`malloc`, `new`) запрашивают большие блоки памяти у операционной системы.

```
#include <sys/mman.h>
#include <unistd.h> // For sysconf

// ...

// Request one page of memory
size_t page_size = sysconf(_SC_PAGESIZE);
void* raw_mem = mmap(nullptr, page_size,
                     PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS,
                     -1, 0);
if (raw_mem == MAP_FAILED) {
    // Error handling
}

```

```
char* data = static_cast<char*>(raw_mem);
// Now 'data' can be used as a regular array
data[0] = 'H';
data[1] = 'i';

// Free the memory
munmap(raw_mem, page_size);
```

Листинг 3.5 – Allocating one page of memory using mmap

Отображение файла в память

Отображение файла позволяет работать с его содержимым как с обычным массивом в памяти, что может быть эффективнее, чем многократные вызовы `read` и `write`, особенно при произвольном доступе.

```
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

const size_t FILE_SIZE = 128;
int fd = open("storage.bin", O_RDWR | O_CREAT, 0644);
ftruncate(fd, FILE_SIZE);
// Set the file size

void* raw_mem = mmap(nullptr, FILE_SIZE,
                     PROT_READ | PROT_WRITE,
                     MAP_SHARED, // Changes will be written to the file
                     fd, 0);
close(fd); // The file descriptor can be closed after mmap

if (raw_mem == MAP_FAILED) { /* ... */ }

char* data = static_cast<char*>(raw_mem);
for (size_t i = 0; i < FILE_SIZE; ++i) {
    data[i] = static_cast<char>(i);
}

// The operating system will write the changes to the disk
// (not necessarily immediately)

munmap(raw_mem, FILE_SIZE);
```

Листинг 3.6 – Working with a file via mmap

3.3.3 Освобождение памяти: `munmap`

Вызов `munmap` удаляет отображение для указанного диапазона виртуальных адресов. Крайне важно освобождать память, выделенную через `mmap`, чтобы избежать утечек ресурсов. Аналогично паре `new`/`delete`, каждому успешному вызову `mmap` должен соответствовать вызов `munmap`.

3.4 Аргументы командной строки и переменные окружения

Кроме ввода-вывода, программа может получать информацию извне при запуске. Рассмотрим два основных механизма: аргументы командной строки и переменные окружения.

3.4.1 Аргументы командной строки

При запуске программы из терминала можно передать ей параметры. Они доступны в функции `main` через её аргументы.

```
int main(int argc, char* argv[]) {
    // ...
}
```

Листинг 3.7 – The main function interface

- `argc` (argument count): количество переданных аргументов.
- `argv` (argument vector): массив указателей на C-строки.

Важно помнить, что `argv[0]` — это всегда имя самой запущенной программы. Реальные аргументы начинаются с `argv[1]`. Например, для команды `./myprog hello world` будет:

- `argc` = 3
- `argv[0]` = `"./myprog"`
- `argv[1]` = `"hello"`
- `argv[2]` = `"world"`

3.4.2 Переменные окружения

Переменные окружения — это набор пар "ключ-значение которые наследуются дочерними процессами от родительских. Они используются для передачи контекста и настроек программам (например, `PATH` для поиска исполняемых файлов, `HOME` для пути к домашней директории). В Linux переменные окружения физически располагаются в памяти процесса сразу после массива `argv`, отделённые от него указателем `nullptr`. Для безопасного доступа к ним из C++ используется функция `getenv`.

```
#include <iostream>
#include <cstdlib> // For getenv

int main() {
    const char* user = std::getenv("USER");
if (user != nullptr) {
        std::cout << "Hello, " << user << "!"
<< std::endl;
    } else {
        std::cout << "USER environment variable is not set."
<< std::endl;
    }
    return 0;
}
```

Листинг 3.8 – Reading an environment variable

Примечание

Переменные окружения часто используются для передачи конфиденциальной информации (ключей API, паролей), так как они, в отличие от аргументов командной строки, не видны другим пользователям системы через команды типа `ps`.

Итоги раздела

- Аргументы командной строки (`argc`, `argv`) позволяют передавать простые параметры при запуске.
- Переменные окружения — это наследуемые пары "ключ-значение"для передачи настроек и контекста.
- Для доступа к переменным окружения следует использовать `getenv`, что является более безопасным и портируемым способом.

Глава 4

4 Лекция

4.1 Углублённая работа с памятью

На прошлой лекции мы познакомились с концепцией виртуальная память и системным вызовом `mmap`, который управляет отображением виртуальных адресов на физическая память. Однако модель, в которой `mmap` немедленно выделяет реальные физические страницы, является упрощением. На практике современные ОС используют более сложный и эффективный механизм.

4.1.1 Механизм Page Fault и ленивое выделение памяти

При попытке программы обратиться по виртуальному адресу, который не сопоставлен ни одной физической странице, процессор генерирует специальное прерывание.

Определение: Страничная ошибка (Page Fault)

страничная ошибка — это прерывание, которое генерируется аппаратно (процессором) при попытке доступа к странице виртуальная память, не имеющей корректного отображения в физическая память. При возникновении страничная ошибка исполнение текущего кода программы приостанавливается, и управление передаётся обработчику в ОС.

ОС анализирует причину страничная ошибка. Если обращение было к некорректному адресу (например, разыменование нулевого указателя), ОС принудительно завершает программу, как правило, с ошибкой **Segmentation Fault**.

Однако этот же механизм используется для реализации **ленивого выделения памяти** (постраничная подкачка по требованию). Когда программа вызывает `mmap`, ОС на самом деле не выделяет физические страницы. Она лишь запоминает, что данный диапазон виртуальных адресов теперь является валидным для процесса. Реальное выделение физической страницы происходит только при **первом обращении** к ней. Это обращение вызывает страничная ошибка, который ОС обрабатывает:

1. Находит свободную физическую страницу.
2. Устанавливает отображение между виртуальной страницей, вызвавшей прерывание, и новой физической страницей.
3. Возобновляет исполнение программы с прерванной инструкции.

Для программы этот процесс прозрачен, за исключением небольшой задержки.

Примечание

Плюсы и минусы ленивого выделения:

- **Плюс:** Эффективное использование ресурсов. Программы часто запрашивают больше памяти, чем реально используют. Ленивый подход позволяет системе выделять только ту физическую память, которая действительно нужна, и поддерживать так называемый *memory overcommitment* (когда суммарный объем запрошенной памяти превышает имеющуюся физическую).
- **Минус:** Усложнение обработки ошибок нехватки памяти. Вместо проверки кода возврата `mmap`, программа может быть внезапно "убита"ОС в произвольный момент при обращении к памяти, если свободные физические страницы закончились.
- **Минус:** Непредсказуемые задержки. Обращение к "новой"странице памяти вызывает страничная ошибка, что приводит к задержке, так как управление передается

ОС. Это может быть критично для приложений реального времени.

4.1.2 Структура таблиц страниц (Page Tables)

Для трансляции виртуальных адресов в физические ОС и процессор используют таблица страниц. Хранить простое линейное отображение для всего 64-битного адресного пространства (даже с учётом реальных ограничений современных процессоров в 256 ТБ) неэффективно.

В архитектуре x86-64 используется **четырехуровневая древовидная структура** таблиц страниц. Виртуальный адрес делится на несколько частей:

- **Смещение (offset):** Младшие 12 бит, указывающие на байт внутри страницы ($2^{12} = 4096$ байт).
- **Индексы в таблицах:** Четыре группы по 9 бит каждая, которые используются для последовательного обхода четырехуровневого дерева таблиц (L3, L2, L1, L0).

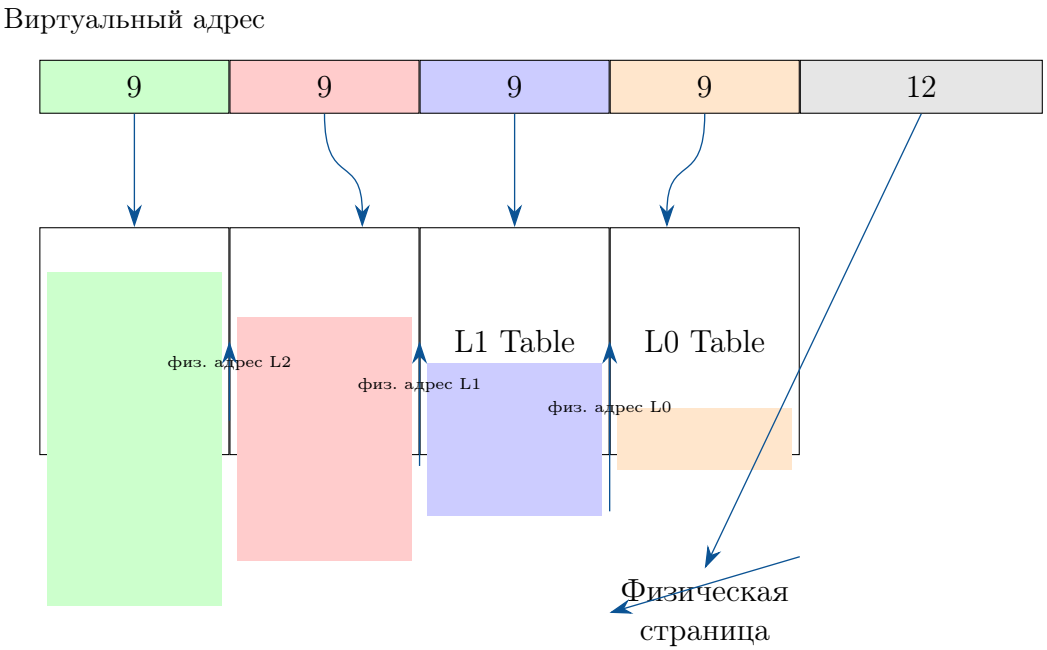


Рис. 4.1 – Трансляция виртуального адреса в физический через четырехуровневые таблицы страниц.

Процессор аппаратно выполняет обход этой структуры при каждом доступе к памяти: использует 9 бит адреса как индекс в таблице L3, находит там физический адрес таблицы L2, затем следующие 9 бит — как индекс в L2, и так далее, пока не дойдет до таблицы L0, где хранится адрес искомой физической страницы.

4.1.3 Дополнительные системные вызовы для работы с памятью

`mprotect(addr, size, prot)` изменяет права доступа (чтение, запись, исполнение) для уже выделенного диапазона виртуальной памяти `[addr, addr+size)`.

`mremap(old_addr, old_size, new_size, flags, ...)` позволяет изменять размер существующего отображения, а также перемещать его на новое место в виртуальном адресном пространстве.

`mlock(addr, size)` "закрепляет"указанный диапазон страниц в физической памяти,

запрещая ОС выгружать их в своп (swap). Это важно для приложений, работающих с чувствительными данными (пароли, ключи шифрования) или требующих предсказуемых задержек. `munlock` отменяет это действие.

Особый режим `mremap` позволяет создать "копию"участка памяти, где два разных диапазона виртуальных адресов указывают на **одни и те же физические страницы**. Любая запись в один диапазон немедленно видна в другом.

```cpp
// Allocate original mapping
void* from = mmap(nullptr, PAGE_SIZE, PROT_READ | PROT_WRITE,
                  MAP_ANONYMOUS | MAP_SHARED, -1, 0);

// Reserve space for the "copy"
void* to_placeholder = mmap(nullptr, PAGE_SIZE, PROT_NONE,
                            MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);

// Create the shared mapping (remap `from` onto `to_placeholder`)
// MREMAP_FIXED tells mremap to use the address we provide.
// old_size = 0 is a special value for this copy operation.
void* to = mremap(from, 0 /* old_size */, PAGE_SIZE,
                  MREMAP_MAYMOVE | MREMAP_FIXED, to_placeholder);

// Now, `from` and `to` point to the same physical page.
volatile int* p_from = static_cast<volatile int*>(from);
volatile int* p_to = static_cast<volatile int*>(to);

*p_from = 123;
// Reading from p_to will now yield 123.
printf("Value at 'to': %d\n", *p_to); // Prints 123
```

Листинг 4.1 – Пример использования mremap для создания разделяемого отображения.

Примечание

В листинг 4.1 используется ключевое слово `volatile`. Оно сообщает компилятору, что значение в памяти, на которую указывает указатель, может измениться в любой момент без его ведома (например, через другой указатель, как в нашем случае). Это запрещает компилятору кэшировать значение переменной в регистре и заставляет его каждый раз честно читать значение из памяти, предотвращая неверные оптимизации.

4.1.4 Swap (своп) и его проблемы

Когда физическая память заканчивается, ОС может использовать своп (swap): выгрузить содержимое некоторых "неактивных"физических страниц на жесткий диск, чтобы освободить место для более актуальных данных. Когда программа обратится к такой выгруженной странице, произойдет страничная ошибка, и ОС загрузит её обратно с диска.

Проблемы свопинга:

- **Производительность:** Диск значительно медленнее оперативной памяти, что приводит к большим задержкам.
- **Безопасность:** Секретные данные (ключи, пароли) могут оказаться на диске в незашифрованном виде и остаться там даже после выключения питания, создавая уязви-

мость.

Итоги раздела

- Обращение к неотмеченной в таблица страниц странице вызывает страничная ошибка.
- ОС использует страничная ошибка для реализации ленивого выделения памяти, что экономит физическую память.
- Трансляция адресов в x86-64 реализована через многоуровневые таблицы страниц.
- Системные вызовы `mprotect`, `mremap`, `mlock` предоставляют тонкий контроль над отображениями памяти.
- своп (swap) помогает при нехватке памяти, но ценой производительности и потенциальных рисков безопасности.

4.2 Управление процессами

До сих пор мы рассматривали работу в рамках одного процесса. Теперь изучим, как создавать новые процессы и управлять ими.

Определение: Процесс

Процесс — это экземпляр запущенной программы. Каждый процесс является изолированной сущностью и обладает собственными ресурсами:

- Уникальным Process ID (идентификатор процесса) (PID).
- Отдельным виртуальным адресным пространством.
- Собственной таблицей файловых дескрипторов.

Процессы могут выполняться параллельно на многоядерных системах.

4.2.1 Подмена процесса: семейство `exec`

Системные вызовы семейства `exec` (`execlp`, `execvpe` и др.) **не создают** новый процесс. Они полностью **заменяют** текущий процесс новым, загружая и запуская указанный исполняемый файл.

```
#include <unistd.h>
#include <cstdio>

int main() {
    printf("Before exec...\n");

    // Replace the current process with "ls -l"
    // The first argument is the command,
    // subsequent args are for its argv.
    // The list must be terminated by a NULL pointer.
    execlp("ls", "ls", "-l", nullptr);

    // This line will never be reached if execlp succeeds.
    perror("execlp failed");
    return 1;
}
```

Листинг 4.2 – Запуск утилиты ls с помощью execlp.

При успешном вызове `execlp` код после него никогда не выполняется. Новый процесс (в данном случае, `ls`) наследует некоторые атрибуты старого, например, таблицу файловых дескрипторов, но получает новое адресное пространство.

Примечание

При запуске сторонних программ важно избегать утечки файловых дескрипторов. Если библиотека внутри вашего кода открыла файл, он останется открытым и в запущенном через `exec` процессе. Стандартное решение — открывать все файловые дескрипторы с флагом `O_CLOEXEC`, который предписывает ядру автоматически закрыть этот дескриптор при вызове `exec`.

4.2.2 Создание процесса: `fork`

Для создания нового процесса используется системный вызов fork.

Определение: Системный вызов fork

`fork()` создаёт точную копию текущего процесса. Уникальность fork в том, что он **вызывается один раз, а возвращается дважды**:

- В **родительском** процессе `fork()` возвращает PID нового (дочернего) процесса.
- В **дочернем** процессе `fork()` возвращает **0**.
- В случае ошибки возвращается -1.

Дочерний процесс является почти полной копией родителя: он получает копию адресного пространства, стека вызовов и таблицы файловых дескрипторов. Исполнение в обоих процессах продолжается с точки сразу после вызова `fork`.

```
#include <unistd.h>
#include <sys/wait.h>
#include <cstdio>

int main() {
    pid_t child_pid = fork();

    if (child_pid == -1) {
        perror("fork failed");
        return 1;
    } else if (child_pid == 0) {
        // We are in the child process
        printf("I am the child! My PID is %d\n", getpid());
    } else {
        // We are in the parent process
        printf("I am the parent! My child's PID is %d\n", child_pid);
        wait(nullptr); // Wait for the child to finish
        printf("Parent knows child has finished.\n");
    }

    return 0;
}
```

Листинг 4.3 – Базовое использование fork.

4.2.3 Жизненный цикл процесса

Завершение процесса: `exit` vs `_Exit`

- `std::exit(code)` — функция стандартной библиотеки. Она не только завершает процесс с кодом `code`, но и выполняет ряд "очищающих"действий: сбрасывает буферы потоков ввода-вывода (например, `cout`), вызывает обработчики, зарегистрированные через `atexit`, и т.д..
- `std::_Exit(code)` (или системный вызов `_exit`) — немедленно завершает процесс без какой-либо очистки. В дочерних процессах после `fork` предпочтительнее использовать именно `_Exit`, чтобы избежать нежелательных побочных эффектов, например, двойного сброса буферов, которые были скопированы от родителя.

Ожидание дочерних процессов: `wait` и `waitpid`

Родительский процесс обязан "собирать"информацию о завершении своих дочерних процессов с помощью `wait()` или `waitpid()`. Эти вызовы блокируют родителя до тех пор, пока один из его детей не завершится, и позволяют получить его код завершения.

Определение: Процесс-зомби

процесс-зомби — это процесс, который уже завершил своё выполнение, но запись о нём (PID, код завершения) всё ещё остаётся в таблице процессов ядра. Он находится в этом состоянии до тех пор, пока родитель не "прочитает"его статус с помощью `wait`. Если родитель не делает `wait`, зомби накапливаются и "утекают"системные ресурсы (в частности, PID).

Определение: Процесс-сирота

процесс-сирота — это процесс, родитель которого завершился раньше него. Такие процессы не остаются "бесхозными"— их "усыновляет"специальный системный процесс `init` (с PID 1), который периодически вызывает `wait` и очищает зомби.

4.2.4 Паттерн `fork-exec`

Комбинация `fork` и `exec` — это стандартный способ в Unix-системах запустить новую программу, не прекращая работу текущей.

1. Родительский процесс вызывает `fork()`, создавая свою копию.
2. В дочернем процессе (где `fork()` вернул 0) выполняются необходимые настройки (например, перенаправление ввода-вывода с помощью `dup2`).
3. Дочерний процесс вызывает один из вызовов семейства `exec`, заменяя себя новой программой.
4. Родительский процесс (где `fork()` вернул PID > 0) может продолжить свою работу или дождаться завершения дочернего с помощью `waitpid()`.

Итоги раздела

- Процесс — это изолированный экземпляр запущенной программы.
- exec **заменяет** текущий процесс, fork **создаёт** его копию.
- Паттерн `fork-exec` является основой для запуска программ в Unix-подобных системах.
- Родитель **обязан** дожидаться завершения дочерних процессов с помощью `wait` или `waitpid`, чтобы избежать появления процесс-зомби.

4.3 Межпроцессное взаимодействие: Pipelines

Одним из самых мощных механизмов в Unix является канал (pipe), который позволяет связать стандартный вывод одного процесса со стандартным вводом другого. Рассмотрим, как реализовать аналог команды `ps aux | grep zsh` программно.

Для этого нам понадобится системный вызов `pipe()`, который создаёт однонаправленный канал данных и возвращает два файловых дескриптора: `pipefd[0]` для чтения и `pipefd[1]` для записи.

Алгоритм реализации пайплайна `cmd1 | cmd2`:

1. Создать канал (pipe) с помощью `pipe(pipefd)`.
2. Вызвать `fork()` для создания первого дочернего процесса (`child1` для `cmd1`).
3. **В `child1`:**
   - Закрыть ненужный конец канала: `close(pipefd[0])`.
   - Перенаправить стандартный вывод на пишущий конец канала: `dup2(pipefd[1], STDOUT_FILENO)`.
   - Закрыть оригинальный дескриптор: `close(pipefd[1])`.
   - Вызвать `exec` для запуска `cmd1`.
4. Вызвать `fork()` для создания второго дочернего процесса (`child2` для `cmd2`).
5. **В `child2`:**
   - Закрыть ненужный конец канала: `close(pipefd[1])`.
   - Перенаправить стандартный ввод на читающий конец канала: `dup2(pipefd[0], STDIN_FILENO)`.
   - Закрыть оригинальный дескриптор: `close(pipefd[0])`.
   - Вызвать `exec` для запуска `cmd2`.
6. **В родительском процессе:**
   - **Критически важно:** закрыть **оба** конца канала: `close(pipefd[0])` и `close(pipefd[1])`. Если этого не сделать, читающий процесс никогда не получит EOF и зависнет.
   - Дождаться завершения обоих дочерних процессов с помощью `waitpid()`.

```
#include <unistd.h>
#include <sys/wait.h>
#include <cstdio>
#include <cstdlib>

int main() {
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        perror("pipe failed");
        return 1;
    }

    pid_t child1 = fork();
    if (child1 == 0) { // First child: ps aux
        close(pipefd[0]); // Close read end
```

```
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]); // Close original write end
        execlp("ps", "ps", "aux", nullptr);
        _exit(1); // exit if exec fails
    }

    pid_t child2 = fork();
    if (child2 == 0) { // Second child: grep zsh
        close(pipefd[1]); // Close write end
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]); // Close original read end
        execlp("grep", "grep", "zsh", nullptr);
        _exit(1); // exit if exec fails
    }

    // Parent process
    close(pipefd[0]); // ESSENTIAL: close both pipe ends in parent
    close(pipefd[1]);

    waitpid(child1, nullptr, 0);
    waitpid(child2, nullptr, 0);

    return 0;
}
```

Листинг 4.4 – Программная реализация пайплайна ps aux | grep zsh.

Примечание

Что происходит, если читатель завершается раньше писателя? (например, в `ps aux | head -n 5`) Когда все читающие концы канала закрываются, а писатель пытается в него что-то записать, ОС посылает писателю сигнал `SIGPIPE`. По умолчанию, действие для этого сигнала — аварийное завершение процесса. Это элегантно решает проблему "бесконечной"работы процессов в начале пайплайна, если их вывод больше никому не нужен.

Итоги раздела

- канал (pipe) создаёт однонаправленный канал для данных между процессами.
- Комбинация `pipe`, `fork`, `dup2` и `exec` позволяет строить сложные конвейеры обработки данных.
- В родительском процессе необходимо закрывать оба конца канала, чтобы избежать взаимоблокировок.
- Сигнал `SIGPIPE` автоматически завершает процессы, которые пытаются писать в "сломанный"канал (без читателей).

Глава 5

5 Лекция

5.1 Управление процессами: группы и сигналы

Продолжая изучение процессов как единиц параллелизма с изолированными адресными пространствами, мы рассмотрим механизмы их организации и взаимодействия. Для эффективного управления множеством связанных процессов операционные системы, включая Linux, предоставляют инструменты для их группировки и асинхронного уведомления.

5.1.1 Группы процессов (Process Groups)

Для упрощения управления несколькими процессами одновременно они могут быть объединены в группы. Каждый процесс в системе принадлежит определённой группе.

Определение: Группа процессов

PGID (Process Group ID) — это числовой идентификатор, общий для нескольких процессов. Он позволяет применять операции, такие как отправка сигналов, ко всей группе сразу, а не к каждому процессу по отдельности. Каждый процесс также принадлежит SID (Session ID), которая объединяет группы процессов.

Для работы с PGID (Process Group ID) существуют системные вызовы:

- `getpgid(pid_t pid)`: получает PGID (Process Group ID) процесса с указанным `pid`. Вызов `getpgid(0)` вернёт PGID (Process Group ID) текущего процесса.
- `setpgid(pid_t pid, pid_t pgid)`: устанавливает PGID (Process Group ID) для процесса. Чтобы создать новую группу, обычно процесс вызывает `setpgid(0, 0)`, что создаёт новую группу с PGID (Process Group ID), равным `PID` этого процесса.

5.1.2 Сигналы (Signals)

Определение: Сигнал

сигнал — это простой механизм IPC (Inter-Process Communication), представляющий собой асинхронное уведомление, которое может быть отправлено процессу операционной системой или другим процессом. В отличие от пайпов, для отправки сигнала не требуется наличие родственной связи между процессами.

Сигналы, как и `PID`, являются просто числами. При получении сигнала процесс может отреагировать одним из нескольких способов:

- **Term (Termination):** Завершение процесса. Это действие по умолчанию для большинства сигналов.
- **Ign (Ignore):** Игнорирование сигнала.
- **Core:** Завершение процесса с генерацией core dump. Это файл, содержащий полный снимок адресного пространства процесса в момент сбоя, что позволяет проводить посмертную отладку (post-mortem debugging) с помощью таких инструментов, как GDB.
- **Stop:** Приостановка выполнения процесса.
- **Cont (Continue):** Возобновление выполнения приостановленного процесса.

Процесс может переопределить стандартную реакцию на большинство сигналов, установив собственный обработчик.

Таблица 5.1 – Некоторые распространённые сигналы и их действия по умолчанию

Сигнал	Номер	Действие	Комментарий
`SIGINT`	2	Term	Отправляется при нажатии `Ctrl+C` в терминале.
`SIGQUIT`	3	Core	Отправляется при нажатии `Ctrl+\`.
`SIGKILL`	9	Term	Гарантированно завершает процесс. Этот сигнал нельзя перехватить или проигнорировать.
`SIGSEGV`	11	Core	Segmentation Fault. Отправляется при попытке доступа к неразрешённой области памяти.
`SIGTSTP`	20	Stop	Отправляется при нажатии `Ctrl+Z` в терминале, приостанавливая процесс.

5.1.3 Отправка сигналов и ожидание процессов

Для отправки сигналов используется системный вызов `kill`. Несмотря на название, он может отправлять любой сигнал, а не только те, что завершают процесс.

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

Листинг 5.1 – Сигнатура системного вызова `kill`

Аргумент `pid` интерпретируется следующим образом:

- `pid > 0`: Сигнал `sig` отправляется процессу с `PID`, равным `pid`.
- `pid < -1`: Сигнал отправляется всем процессам в группе с PGID (Process Group ID), равным `-pid`.
- `pid == 0`: Сигнал отправляется всем процессам в группе текущего процесса.
- `pid == -1`: Сигнал отправляется всем процессам, которым текущий пользователь имеет право отправлять сигналы (за исключением некоторых системных процессов).

Механизмы ожидания дочерних процессов, такие как `wait` и `waitpid`, позволяют не только дождаться завершения, но и получить информацию о причине.

- `WIFEXITED(status)`: Возвращает `true`, если процесс завершился штатно через вызов `exit()`. Код возврата можно получить с помощью `WEXITSTATUS(status)`.
- `WIFSIGNALED(status)`: Возвращает `true`, если процесс был завершён сигналом. Номер сигнала можно получить через `WTERMSIG(status)`.

Вызов `waitpid` также интегрирован с группами процессов:

- `waitpid(-1, ...)`: Ждёт любого дочернего процесса (стандартное поведение).
- `waitpid(-pgid, ...)`: Ждёт завершения любого дочернего процесса из группы с PGID (Process Group ID).

5.1.4 Управление памятью при `fork()`

Ранее мы говорили, что `fork()` создаёт полную копию адресного пространства родителя для дочернего процесса. Для современных процессов, занимающих гигабайты памяти, полное копирование было бы крайне неэффективным.

Определение: Copy-on-Write (COW)

Copy-on-Write (копирование при записи) — это оптимизация, применяемая при вызове `fork()`. Вместо реального копирования всех страниц памяти, операционная система создаёт для дочернего процесса новые таблицы страниц, которые указывают на те же физические страницы, что и у родителя. Все эти страницы помечаются как доступные только для чтения. При попытке записи в такую страницу (и родителем, и ребёнком) происходит аппаратное прерывание. ОС перехватывает его, создаёт реальную копию только этой конкретной страницы, и уже в неё производится запись. Это позволяет копировать только те данные, которые действительно изменяются, экономя время и память.

Общая память через `mmap`

Хотя Copy-on-Write (копирование при записи) обеспечивает изоляцию, иногда процессам нужна общая область памяти для эффективного взаимодействия. Этого можно достичь с помощью системного вызова `mmap` с флагом `MAP_SHARED`.

Примечание

Если участок памяти был создан с флагом `MAP_PRIVATE`, то при `fork()` к нему применяется Copy-on-Write (копирование при записи). Если же был использован флаг `MAP_SHARED`, то этот участок памяти будет общим для родителя и ребёнка после `fork()`: изменения, сделанные одним процессом, будут видны другому.

Важно помнить, что при работе с общей памятью возникает проблема синхронизации. Нет гарантий относительно порядка выполнения инструкций в родителе и ребёнке. Чтобы корректно читать данные, записанные другим процессом, необходимо использовать механизмы синхронизации, например, пайпы или сигналы, чтобы уведомить читающий процесс о готовности данных.

Итоги раздела

- **Группы процессов** (PGID (Process Group ID)) упрощают управление множеством процессов, позволяя применять операции ко всей группе сразу.
- **Сигналы** — это механизм асинхронных уведомлений для межпроцессного взаимодействия.
- Вызов `kill` позволяет отправлять сигналы процессам и группам процессов.
- `waitpid` может ожидать завершения процессов из определённой группы.
- **Copy-on-Write** (Copy-on-Write (копирование при записи)) оптимизирует `fork()`, откладывая реальное копирование страниц памяти до первой операции записи.
- `mmap` с флагом `MAP_SHARED` создаёт общую область памяти для родителя и дочерних процессов, но требует явной синхронизации доступа.

5.2 Представление данных

Любые данные в компьютерных системах — будь то файлы, сетевые пакеты или потоки ввода-вывода — в конечном счёте представляются в виде последовательности байт. Способ преобразования структурированных данных в байты и обратно определяет формат данных.

5.2.1 Текстовые и бинарные форматы

Форматы данных условно делятся на две большие категории.

- **Текстовые форматы** (JSON, YAML, XML, TXT) оптимизированы для чтения и редактирования человеком. Они, как правило, избыточны и требуют больше ресурсов для парсинга (синтаксического анализа).
- **Бинарные форматы** (исполняемые файлы ELF, архивы ZIP, форматы сериализации BSON, Protocol Buffers) оптимизированы для машинной обработки, компактности или скорости. Они нечитаемы для человека без специальных инструментов, но обрабатываются программами гораздо эффективнее.

Некоторые бинарные форматы, например, исполняемые файлы ELF (Executable and Linkable Format), спроектированы так, что их можно загрузить в память простым отображением файла с помощью `mmap`, без дополнительной обработки.

Тонкости бинарных форматов: порядок байтов

При работе с бинарными данными, содержащими числа размером более одного байта, возникает проблема порядок байтов (endianness).

Определение: Порядок байтов (Endianness)

порядок байтов (endianness) определяет, как байты многобайтового числа располагаются в оперативной памяти.

- **Little-endian**: Младший байт числа хранится по младшему адресу памяти. Этот порядок используется в большинстве современных архитектур, включая x86-64.
- **Big-endian**: Старший байт числа хранится по младшему адресу. Этот порядок часто используется в сетевых протоколах (network byte order).

Проблема возникает при передаче бинарных данных между системами с разным порядком байтов. Если данные записываются и читаются на одной и той же машине, беспокоиться о порядке байтов не нужно.

5.2.2 Кодировки текста: от ASCII до Unicode

Представление текста — одна из фундаментальных задач. Исторически первой и наиболее влиятельной кодировкой стала ASCII (American Standard Code for Information Interchange).

Определение: ASCII

ASCII (American Standard Code for Information Interchange) — стандарт, определяющий соответствие между числами (кодами от 0 до 127) и символами. Он использует только 7 бит, восьмой бит всегда равен нулю. ASCII включает в себя символы английского алфавита, цифры, знаки препинания и управляющие символы.

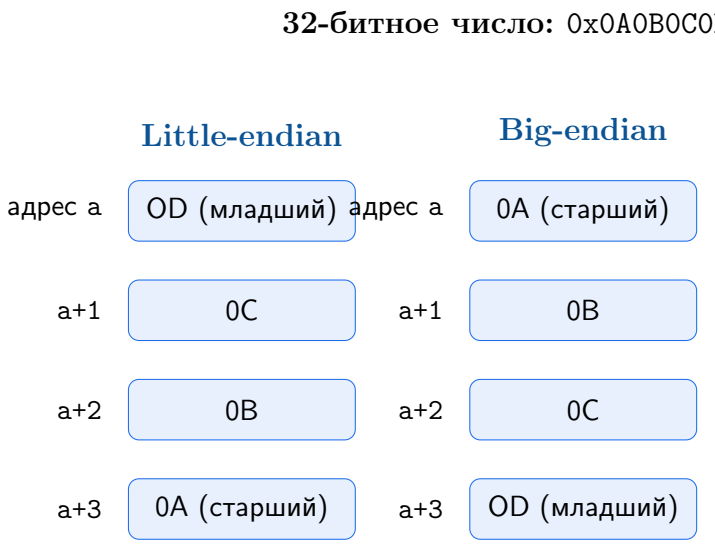


Рис. 5.1 – Расположение байтов числа `0x0A0B0C0D` в памяти при разном порядке байтов

Среди управляющих символов ASCII есть:

- `0x0A (\n)` — перевод строки (Line Feed).
- `0x0D (\r)` — возврат каретки (Carriage Return).
- `0x1B (ESC)` — Escape, используется для начала управляющих последовательностей, например, для управления цветом текста в терминале.

Примечание

Исторически сложилось два основных способа кодирования конца строки в текстовых файлах:

- **Unix/Linux**: используется один символ `\n`.
- **DOS/Windows**: используется последовательность из двух символов `\r\n` (возврат каретки и перевод строки).

Это различие может вызывать проблемы, например, при запуске скриптов с Windows-окончаниями строк в Linux, так как интерпретатор может неверно обработать `\r` в конце строки shebang.

Очевидным недостатком ASCII (American Standard Code for Information Interchange) является отсутствие поддержки символов других языков. Это привело к появлению множества несовместимых 8-битных кодировок (семейства ISO-8859, CP1251, KOI8-R и др.), которые использовали старший бит для кодирования национальных алфавитов. Проблема усугублялась в языках с иероглифической письменностью, где требовались многобайтовые кодировки со сложной логикой переключения режимов (например, EUC-JP).

5.2.3 Unicode и его кодировки

Для решения проблемы хаоса кодировок был создан стандарт Unicode.

Определение: Unicode

Unicode — это стандарт, который сопоставляет каждому символу из большинства мировых письменностей (включая мёртвые языки и эмодзи) уникальное целое число, называемое **Unicode**. Всего стандарт определяет более миллиона кодовых позиций (от `U+0000` до `U+10FFFF`).

Ключевые свойства Unicode:

- Первые 128 кодовых позиций (`U+0000` – `U+007F`) полностью совпадают с ASCII (American Standard Code for Information Interchange), обеспечивая частичную совместимость.
- Unicode сам по себе не является кодировкой. Он лишь определяет соответствие «символ ↔ число». Для представления этих чисел в виде байтов используются специальные кодировки (encodings).
- Стандарт имеет свои сложности: один и тот же видимый символ (глиф) может быть представлен либо одним кодпоинтом, либо комбинацией из базового символа и модифицирующего знака (например, диакритики). Например, символ 'ё' можно представить как `U+0451` или как комбинацию 'е' (`U+0435`) и диерезиса (`U+0308`).

Кодировка UTF-8

Существует несколько способов кодирования кодпоинтов Unicode в байты: UCS-2, UCS-4, UTF-16. Наиболее популярным и де-факто стандартом в вебе и современных ОС стал UTF-8 (Unicode Transformation Format, 8-bit).

Определение: UTF-8

UTF-8 (Unicode Transformation Format, 8-bit) — это кодировка Unicode с переменной длиной символа. Она кодирует кодпоинты в последовательности от 1 до 4 байт.

Основные преимущества UTF-8:

- **Совместимость с ASCII:** Любой текст в кодировке ASCII является корректным текстом в UTF-8, так как кодпоинты до `U+007F` кодируются одним байтом, полностью совпадающим с их ASCII-представлением.
- **Эффективность:** Маленькие значения кодпоинтов кодируются меньшим числом байт. Это экономит место для текстов на латинице.
- **Надёжность:** Нулевой байт (`0x00`) используется только для кодирования нулевого кодпоинта (`U+0000`). Это позволяет использовать стандартные C-функции для работы со строками, оканчивающимися нулём.
- **Самосинхронизация:** По значению любого байта можно определить, является ли он началом символа или частью многобайтовой последовательности. Это позволяет легко находить границы символов в потоке байт.

Принцип кодирования в UTF-8 основан на использовании старших битов первого байта для указания общей длины последовательности (см. Table 5.2).

5.2.4 Работа с Unicode в C/C++

Для поддержки Unicode в языках C и C++ существует тип `wchar_t` («широкий символ»).

Таблица 5.2 – Схема кодирования символов в UTF-8

Длина	Диапазон кодпоинтов	Схема байтов (x — биты кодпоинта)
1 байт	U+0000 – U+007F	`0xxxxxxx`
2 байта	U+0080 – U+07FF	`110xxxxx 10xxxxxx`
3 байта	U+0800 – U+FFFF	`1110xxxx 10xxxxxx 10xxxxxx`
4 байта	U+10000 – U+10FFFF	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

- `wchar_t`: Тип для хранения одного кодпоинта. Его размер зависит от платформы, но часто составляет 4 байта (32 бита), что достаточно для любого символа Unicode (аналогично кодировке UCS-4 в памяти).
- `L'a'`: Литерал типа `wchar_t`.
- `L"строка"`: Широкий строковый литерал (массив `const wchar_t`).
- Стандартная библиотека предоставляет аналоги для работы с широкими строками: `std::wstring`, `std::wcin`, `std::wcout` в C++ и функции `wprintf`, `wscanf` в C.

Локали

Чтобы стандартная библиотека знала, как преобразовывать широкие символы (`wchar_t`) в байтовые последовательности (например, в UTF-8) при вводе-выводе, ей необходимо сообщить текущие региональные настройки.

Определение: Локаль

локаль — это набор параметров, описывающих языковые и культурные особенности пользователя. локаль определяет кодировку текста (`LC_CTYPE`), формат чисел, валюты, даты и времени. Она обычно задаётся через переменные окружения, такие как `LANG` или `LC_ALL`.

Перед началом работы с широким вводом-выводом в программе на C++ необходимо установить глобальную локаль, чтобы она была унаследована от настроек операционной системы.

```cpp
#include <locale>
#include <iostream>
#include <string>

int main() {
    // Set the global locale based on the system's environment variables.
    // An empty string "" means "take from environment".
    std::locale::global(std::locale(""));

    // Now std::wcin and std::wcout will work correctly
    // with the encoding specified in the locale (e.g., UTF-8).
    std::wstring s;
    std::wcout << L"input text: ";
    std::wcin >> s;
    std::wcout << L"you input: " << s << L", len: " << s.size() << L" symb" << std
        ::endl;

    return 0;
```

```
18 }
```

Листинг 5.2 – Установка локали для корректной работы с Unicode в C++

Примечание

Смешивать обычные потоки (`std::cout`) и широкие (`std::wcout`) в одной программе не рекомендуется. Такое смешивание может привести к непредсказуемому поведению и некорректному выводу, так как внутреннее состояние потоков может быть нарушено.

Итоги раздела

- Данные могут быть представлены в **текстовом** (человекочитаемом) или **бинарном** (машиночитаемом) формате.
- При работе с бинарными данными важно учитывать **порядок байтов** (порядок байтов (endianness)), особенно при передаче данных между разными системами.
- **ASCII** — исторически важная, но ограниченная 7-битная кодировка.
- **Unicode** является универсальным стандартом, присваивающим уникальный номер (Unicode) каждому символу.
- **UTF-8** — самая популярная кодировка для Unicode, эффективная и обратно совместимая с ASCII.
- В C/C++ для работы с Unicode используется тип `wchar_t` и связанные с ним строковые классы и функции ввода-вывода.
- Для корректного ввода-вывода Unicode-текста необходимо настроить **локаль** программы, чтобы она соответствовала системным настройкам.

Глава 6

6 Лекция

6.1 Представление целых чисел

В прошлых лекциях мы обсуждали представление текстовых данных. Теперь рассмотрим, как в памяти кодируются целые числа.

6.1.1 Беззнаковые числа

С беззнаковыми (unsigned) числами всё просто. Они представляются напрямую своим двоичным эквивалентом. Если у нас есть N бит, мы можем представить числа от 0 до $2^N - 1$. Например, для 3-битного числа:

- `000` $\rightarrow 0$
- `001` $\rightarrow 1$
- `010` $\rightarrow 2$
- `111` $\rightarrow 7$

6.1.2 Знаковые числа: Прямой код (Sign-Magnitude)

Первая и самая прямолинейная идея для представления знаковых чисел — использовать один бит (обычно старший) для кодирования знака, а остальные биты — для кодирования абсолютного значения (величины).

Например, для 3-битного числа (1 бит на знак, 2 на значение):

- `001` $\rightarrow +1$
- `010` $\rightarrow +2$
- `101` $\rightarrow$ -1
- `110` $\rightarrow$ -2

У этого подхода есть два существенных недостатка:

1. **Проблема двух нулей:** Существует два представления для нуля: `000` ($+0$) и `100` (-0). Это избыточно и усложняет проверки.
2. **Сложная арифметика:** Обычный двоичный сумматор "ломается". Сложение $+1$ (`001`) и -1 (`101`) в лоб даст `110`, что равно -2, а не 0. Для выполнения арифметических операций требуются сложные проверки знаков.

6.1.3 Знаковые числа: Дополняющий код (Two's Complement)

Современные компьютеры решают эти проблемы, используя Дополняющий код.

Определение: Дополняющий код

Дополняющий код — это способ представления знаковых чисел, основанный на арифметике по модулю 2^N, где N — количество бит.

- Положительные числа (и 0) представляются так же, как и беззнаковые (в диапазоне от 0 до $2^{N-1} - 1$).
- Отрицательные числа x (в диапазоне от -2^{N-1} до -1) представляются как беззнаковое число $2^N + x$.

Рассмотрим 3-битные числа (модуль $2^3 = 8$):

- `000` $\rightarrow 0$

- `001` $\to 1$
- `010` $\to 2$
- `011` $\to 3$
- `100` $\to 4$ (или $4 - 8 = -4$)
- `101` $\to 5$ (или $5 - 8 = -3$)
- `110` $\to 6$ (или $6 - 8 = -2$)
- `111` $\to 7$ (или $7 - 8 = -1$)

Преимущества дополняющего кода:

- **Один ноль:** Значение `000` уникально.
- **Простая арифметика:** Обычный двоичный сумматор корректно работает как для знаковых, так и для беззнаковых чисел.

Примечание

Пример арифметики: Сложим 1 (`001`) и -2 (`110`) как знаковые.

$$\texttt{001} + \texttt{110} = \texttt{111}$$

Результат `111` в дополняющем коде — это -1. Сложение работает.
Теперь сложим 1 (`001`) и 6 (`110`) как беззнаковые.

$$\texttt{001} + \texttt{110} = \texttt{111}$$

Результат `111` в беззнаковом коде — это 7. Сложение также работает.

Получение отрицательного числа

Практическое правило для получения представления числа $-x$ из x в дополняющем коде:

1. Инвертировать все биты x (операция ~x, побитовое НЕ).
2. Прибавить к результату 1.

Формула: $-x = \sim x + 1$.

Пример: Найти представление -3 (для 3-битного числа).

1. Берём 3: `011`
2. Инвертируем (~): `100`
3. Прибавляем 1: `100` + 1 = `101`

Результат `101` — это -3, что совпадает с нашей таблицей.

6.2 Выравнивание данных в памяти

Определение: Выравнивание данных (Data Alignment)

Выравнивание данных — это ограничение, согласно которому данные определённого типа и размера должны размещаться в памяти по адресам, кратным некоторой степени двойки.
Например, 8-байтный `int64_t` должен иметь адрес, который делится на 8 (т.е. `address % 8 == 0`).

6.2.1 Зачем нужно выравнивание?

Выравнивание — это не просто прихоть компилятора, а требование, диктуемое аппаратным обеспечением (процессором).

- **Эффективность:** Процессоры читают данные из памяти не по одному байту, а "блоками"(например, по 4, 8 или 16 байт). Если 8-байтовое число "пересекает"границу такого блока (например, начинается с адреса 4 и заканчивается на 11), процессору придётся выполнить два чтения из памяти вместо одного.
- **Корректность:** На некоторых архитектурах (не x86) обращение по невыровненному адресу может привести к немедленному падению программы (аппаратному прерыванию). На x86 это "всего лишь"приводит к сильному замедлению.
- **Атомарность:** Операции чтения/записи по выровненным адресам, как правило, атомарны (неделимы). Невыровненная запись (например, 8 байт) может быть выполнена процессором как две отдельные записи по 4 байта.

Примечание

Проблема неатомарности особенно важна при работе с разделяемой памятью (shared memory). Представим, что два процесса (например, полученные через `fork()` с памятью `mmap(MAP_SHARED)`) работают с одним 8-байтным числом по невыровненному адресу.
Процесс А пишет новое значение. Он может успеть записать первые 4 байта, но не вторые. В этот момент Процесс Б читает это число и видит "мусор"— половину старого значения и половину нового.

Из-за этих требований компиляторы (C, C++, Rust, Go) автоматически вставляют "пропуски"(padding) в структуры, а стандартные аллокаторы (`malloc`, `operator new`) возвращают память, выровненную по максимальному требованию для стандартных типов (например, 16 байт на x86-64).

6.3 Процесс сборки программы

Рассмотрим, почему в C/C++ принято разделять код на заголовочные файлы (`.h`) и файлы реализации (`.cpp`).

6.3.1 Препроцессор и `#include`

Первый этап сборки — Препроцессинг. Директивы, начинающиеся с `#`, обрабатываются на этом этапе.

Директива `#include "header.h"` — это простая текстовая операция. Она заменяет эту строку содержимым файла `header.h`. Это можно проверить, запустив компилятор с флагом `-E`:

```
# gcc -E main.c
```

Листинг 6.1 – Запуск только препроцессора

Проблема многократного включения

Если один `.h` файл включается несколько раз (например, `a.h` и `b.h` оба включают `common.h`, а `main.cpp` включает `a.h` и `b.h`), мы получим дублирование кода и ошибки компиляции.

Для решения этой проблемы используются Страж включения:

- **Классический способ (Стражи):**

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// ... soderzhimoe zagolovka ...

#endif // MY_HEADER_H
```

Листинг 6.2 – Использование ifndef/define

- **Современный способ:**

```
#pragma once

// ... soderzhimoe zagolovka ...
```

Листинг 6.3 – Использование pragma once

Оба способа гарантируют, что препроцессор включит тело файла только один раз.

6.3.2 Единицы трансляции и ускорение сборки

Основная причина разделения кода на `.h` и `.cpp` — это **ускорение сборки** больших проектов за счёт параллелизма.

Определение: Единица трансляции (Translation Unit)

Единица трансляции — это один `.c` или `.cpp` файл после того, как препроцессор "вклеил"в него содержимое всех `#include`.

Процесс сборки можно разбить на два этапа:

1. **Компиляция (Compilation):** Компилятор (например, `gcc -c`) *независимо и параллельно* обрабатывает каждую единицу трансляции, превращая её в Объектный файл (<code>.o</code>). Этот этап включает синтаксический анализ, оптимизации (<code>-O2</code>) и генерацию машинного кода.
2. **Линковка (Linking):** Линковщик (компоновщик) берёт все <code>.o</code> файлы и "сшивает"их в один исполняемый файл. Этот этап, как правило, последовательный, но он выполняется быстрее, чем полная перекомпиляция всего проекта.

Если мы меняем один `.cpp` файл, нам нужно перекомпилировать только его, а затем быстро перелинковать проект. Если бы весь код был в одном файле, любое изменение требовало бы полной перекомпиляции.

6.3.3 Объектные файлы и символы

Объектный файл (<code>.o</code>) — это "полуфабрикат". Он содержит машинный код, но в нём ещё нет информации о том, где находятся функции и переменные из *других* <code>.o</code> файлов.

Связь между файлами осуществляется через **символы**. С помощью утилиты `nm` можно посмотреть таблицу символов объектного файла.

```
# nm main.o
0000000000000000 T main
                 U isEven
                 U printf
                 U scanf
```

Листинг 6.4 – Анализ символов с помощью nm

- **T (Text):** Символ *определён* (defined) в этом файле. Здесь определён `main`.
- **U (Undefined):** Символ *используется*, но не определён. Линковщик должен будет найти его в другом <code>.o</code> файле или библиотеке.

6.3.4 Линковка и релокации

Когда компилятор генерирует `main.o` и видит вызов `isEven()`, он не знает адреса этой функции. Вместо адреса он оставляет "дырку"— специальную запись, называемую релокация.

Задача линковщика:

1. Найти `main.o`, у которого `isEven` помечен как `U`.
2. Найти другой <code>.o</code> файл (например, `even.o`), у которого `isEven` помечен как `T`.
3. "Заполнить дырку"(выполнить релокацию) в `main.o`, подставив реальный адрес `isEven` из `even.o`.

Если линкощик не может найти определение для `U`-символа (или находит *несколько* определений), он выдаёт ошибку ("Undefined reference"или "Multiple definition").

Примечание

Поскольку <code>.o</code> файлы содержат только машинный код и таблицу символов (а не C++ или C код), они языково-независимы. Это позволяет компоновать программу из частей, написанных на разных языках (например, скомпилировать функцию на Rust, а вызвать её из C).

6.3.5 Формат ELF

В Linux исполняемые файлы и объектные файлы хранятся в формате ELF (Executable and Linkable Format). Он состоит из **секций**, которые сообщают загрузчику ОС, как создать образ процесса в памяти.

Основные секции:

- `.text`: Исполняемый код (инструкции CPU). Загружается с правами "чтение + исполнение".

- `.rodata`: (Read-Only Data) Константные данные, например, строковые литералы (<code>"Hello«/code>). Загружается с правами "только чтение".
- `.data`: Инициализированные глобальные и статические переменные (<code>int x = 10;</code>). Загружается с правами "чтение + запись".
- `.bss`: Неинициализированные глобальные и статические переменные (<code>int y;</code>). Эта секция *не занимает места в файле*, она просто говорит загрузчику: "выдели X байт памяти и заполни их нулями".

6.4 Особенности C++: Имена и переменные

6.4.1 Искажение имён (Name Mangling)

В C++ можно объявлять функции с одинаковыми именами, но разными аргументами (перегрузка) или в разных пространствах имён:

```
void f();
void f(int);
namespace A { void f(); }
```

Линковщик C не справился бы с этим, так как он видит только один символ `f`. Компилятор C++ решает эту проблему, кодируя полную сигнатуру функции в имя символа. Этот процесс называется Искажение имён (Name Mangling).

Например, `A::f()` может превратиться в `_Z1A1fv`.

6.4.2 Extern C

Чтобы C++ мог вызывать функции из C (или из ассемблера, который следует C-соглашениям) или наоборот, нужно отключить Искажение имён (Name Mangling). Для этого используется extern "C":

```
// Ob'yavlyaem, chto eta funktsiya ispol'zuet C ABI
// (bez iskazheniya imen)
extern "C" void my_c_function(int x);
```

6.4.3 Глобальные переменные: `extern` против `static`

Как и в случае с функциями, глобальные переменные нужно *объявлять* (в `.h`) и *определять* (в `.cpp`).

- **Правильный способ (Общая переменная):**

```
// Govorim kompilyatoru, chto peremennaya *gde-to* sushchestvuet
extern int shared_value;
```

```
// Vydelyaem pamyat' i zadaem znachenie
int shared_value = 123;
```

Все `.cpp` файлы, включившие `def.h`, будут ссылаться на *одну и ту же* копию `shared_value`.

- **Неправильный способ (Локальные копии)**

```
// 'static' v global'noy oblasti vidimosti
// delaet peremennuyu lokal'noy dlya edinitsy translyatsii
static int value = 0;
```

Если `main.cpp` и `other.cpp` включат `static.h`, *каждый* из них получит свою *собственную, независимую* копию `value`. Линкер не выдаст ошибки, но программа будет работать некорректно.

6.5 Основы ассемблера и архитектуры

6.5.1 Архитектура фон Неймана

Современные процессоры в основном следуют Архитектура фон Неймана .

Ключевая особенность — единый блок памяти, в котором хранятся и данные, и инструкции (код) программы. Процессор выполняет инструкции последовательно, используя Instruction Pointer, регистр-указатель на следующую инструкцию (RIP) (Instruction Pointer) для отслеживания адреса текущей инструкции.

6.5.2 Регистры и память

Доступ к оперативной памяти (RAM) — медленная операция (порядка 100 ns). Чтобы процессор не простаивал, он содержит Регистр — сверхбыстрые ячейки памяти.

В архитектуре x86-64 (которую мы используем) есть 16 64-битных регистров общего назначения (<code>RAX</code>, <code>RCX</code>, <code>RDX</code>, <code>RSI</code>, <code>RDI</code>, <code>R8</code>...<code>R15</code> и т.д.).

Два регистра имеют особо важное значение:

- **RIP:** Указатель на инструкцию.
- **Stack Pointer, регистр-указатель на вершину стека (RSP):** Указатель на вершину стека.

6.5.3 Стек и вызовы функций

Для реализации вызовов функций используется стек.

1. `call f` (Вызов функции):
   - Процессор помещает адрес *следующей* за `call` инструкции (адрес возврата) на вершину стека (<code>push return_addr</code>).
   - Процессор совершает безусловный переход на адрес функции `f` (<code>jmp f</code>).
2. `ret` (Возврат из функции):
   - Процессор снимает адрес возврата с вершины стека (<code>pop return_addr</code>).
   - Процессор совершает безусловный переход на этот адрес (<code>jmp return_addr</code>).

6.5.4 Соглашение о вызовах (ABI)

Как функции передают аргументы и возвращают значения? Процессор об этом "не знает". Это определяется программным соглашением — ABI (Application Binary Interface).

Для Linux x86-64 (System V ABI) действуют следующие правила:

Определение: Соглашение о вызовах (x86-64 System V ABI)

- **Передача аргументов (целочисленных):**
  - 1-й аргумент: RDI
  - 2-й аргумент: RSI

- 3-й аргумент: <code>RDX</code>
- 4-й аргумент: <code>RCX</code>
- 5-й аргумент: <code>R8</code>
- 6-й аргумент: <code>R9</code>

- **Передача аргументов (7-й и далее):**
  - Передаются через стек. Вызывающая сторона (caller) кладёт их на стек в обратном порядке *до* выполнения инструкции `call`.
- **Возвращаемое значение:**
  - RAX

Примечание

Аргументы на стеке. Поскольку `call` кладёт на стек адрес возврата, внутри вызываемой функции (callee) аргументы, переданные через стек, оказываются смещены:

- `[rsp]` — Адрес возврата (положен инструкцией `call`)
- `[rsp+8]` — 7-й аргумент
- `[rsp+16]` — 8-й аргумент
- и т.д.

(Здесь `[addr]` означает "прочитать 8 байт из памяти по адресу `addr`").

6.6 Практика: написание функций на ассемблере

Мы будем использовать синтаксис Intel. Файл `.S` должен начинаться с директив:

```
.intel_syntax noprefix # Ustanavlivaem sintaksis
.text # Nachalo sektsii koda
```

Чтобы сделать функцию `my_func` видимой для линковщика (C/C++), её нужно объявить глобальной:

```
.global my_func
my_func:
    # ... instruktsii ...
    ret
```

6.6.1 Пример 1: Возврат константы

```
// C++: extern "C" long constant();
```

```
.global constant
constant:
    mov rax, 42 # Vozvrashchaemoe znachenie - v RAX
    ret
```

6.6.2 Пример 2: Identity (аргумент -> возврат)

```
// C++: extern "C" long identity(long x);
```

```
.global identity
identity:
    # 1-y argument 'x' prihodit v RDI
    mov rax, rdi # Peremeshchaem RDI v RAX
    ret
```

6.6.3 Пример 3: Сложение (два аргумента)

```
// C++: extern "C" long add(long x, long y);
```

```
.global add
add:
    # x v RDI, y v RSI
    add rdi, rsi # Skladyvaem: RDI = RDI + RSI
    mov rax, rdi # Peremeshchaem rezul'tat (v RDI) v RAX
    ret
```

6.6.4 Пример 4: Условный переход (If/Else)

```
/* C++: extern "C" long select(long cond, long a, long b);
 * if (cond == 0) return b;
 * else return a;
 */
```

```
.global select
select:
    # cond v RDI, a v RSI, b v RDX

    # Proveryaem RDI na nol'.
    # Operatsiya 'add' menyaet RFLAGS, v t.ch. Zero Flag (ZF)
    add rdi, 0

    # jz (Jump if Zero) - perehod, esli ZF=1 (rezul'tat byl 0)
    jz .L_return_b

.L_return_a:
    # cond != 0
    mov rax, rsi # return a
    ret

.L_return_b:
    # cond == 0
    mov rax, rdx # return b
    ret
```

6.6.5 Пример 5: Цикл (Sum)

```
/* C++: extern "C" long sum(long n);
 * long s = 0;
 * for (long i = n; i > 0; i--) { s += i; }
 * return s;
 */
```

```
.global sum
sum:
    # n v RDI
    xor rax, rax # rax (summa) = 0

.L_loop_start:
    add rax, rdi # summa += n
    add rdi, -1 # n--

    # 'add rdi, -1' (n--):
    # - Esli n > 0, perenosa (carry) ne budet.
    # - Esli n == 0, to 0 + (-1) daet perenos (borrow).

    # jnc (Jump if No Carry) - prygnut', esli n > 0
    jnc .L_loop_start

    # n == 0, tsikl zavershen
    ret
```

Итоги раздела

Итоги раздела "Ассемблер":

- Код на ассемблере — это прямое представление машинных инструкций (мнемоники).
- Взаимодействие с C/C++ происходит через ABI (Application Binary Interface) (соглашение о вызовах).
- В Linux x86-64 аргументы передаются через регистры (`RDI`, `RSI` и т.д.), а возвращаемое значение — через `RAX`.
- Аргументы, не поместившиеся в регистры (7-й и далее), передаются через стек и доступны по адресу `[rsp+8]`, `[rsp+16]` и т.д.
- Управление потоком (if, loop) реализуется через RFLAGS и инструкции условных переходов (`jz`, `jnc` и др.).

Глава 7

7 Лекция

7.1 Адресация памяти в x86-64

Продолжаем изучение низкоуровневый язык программирования, близкий к машинному коду (Ассемблер). Ключевой темой является работа с памятью. В прошлый раз мы установили, что для обращения к памяти (разыменования) используется синтаксис с квадратными скобками.

7.1.1 Синтаксис Scale-Index-Base (SIB)

Общий синтаксис адресации памяти в 64-битном режиме (в Intel-синтаксис) выглядит следующим образом:

$$[rbase + rindex \times scale + displacement]$$

где:

- `rbase` — базовый регистр.
- `rindex` — регистр-индекс.
- `scale` — множитель (масштаб) для индекса. Допустимые значения: $scale \in \{1, 2, 4, 8\}$.
- `displacement` — константное смещение (сдвиг).

Этот синтаксис был разработан для удобной работы с массивами и структурами. Например, `rbase` может хранить адрес начала массива, `rindex` — индекс элемента, `scale` — размер одного элемента (e.g., 8 байт для `uint64_t`), а `displacement` — сдвиг до нужного поля внутри структуры.

```
; * (uint64_t*)(rax + 8 * rdx) = rcx
; (rax = base, rdx = index, 8 = scale)
mov [rax + rdx * 8], rcx

; * (uint64_t*)(rbx + rbp + 32) = rax
; (rbx = base, rbp = index, 1 = scale (default), 32 = displacement)
mov [rbx + rbp + 32], rax
```

Листинг 7.1 – Примеры SIB-адресации

7.1.2 Указание размера операнда

В листинг 7.1 ассемблер мог угадать размер операции (64 бита) по размеру регистра `rcx` или `rax`. Однако при работе с константами возникает неоднозначность.

```
mov [rax], 0 ; OSHIBKA: Neizvesten razmer: 1, 2, 4 ili 8 bayt?
```

Листинг 7.2 – Неоднозначность размера

Компилятор ассемблера не знает, какой размер данных вы намереваетесь записать. Для явного указания размера используются специальные директивы:

- `BYTE PTR` — 8 бит (1 байт).
- `WORD PTR` — 16 бит (2 байта).
- `DWORD PTR` — 32 бита (4 байта).
- `QWORD PTR` — 64 бита (8 байт).

```
; * (uint32_t*)rax = 0
mov DWORD PTR [rax], 0
```

Листинг 7.3 – Явное указание размера (32 бита)

Итоги раздела

- Адресация SIB ($[base + index \times scale + disp]$) — основной механизм доступа к памяти.
- $scale$ ограничен значениями $\{1, 2, 4, 8\}$.
- При неоднозначности (например, при записи константы) размер операции нужно указывать явно (e.g., `DWORD PTR`).

7.2 Инструкция LEA (Load Effective Address)

Инструкция Load Effective Address, инструкция загрузки вычисленного адреса (LEA) — один из самых полезных и часто используемых инструментов в Ассемблер.

Определение: LEA (Load Effective Address)

Инструкция `lea` **вычисляет** адрес, используя синтаксис SIB, но **не разыменовывает** его. Вместо этого она записывает вычисленный адрес в регистр-приемник.

```
; MOV: Prochitat' 8 bayt po adresu [rax] i polozhit' v rdx
; rdx = * (uint64_t*)rax
mov rdx, [rax]

; LEA: Vychislit' adres (v etom sluchae prosto rax) i polozhit' v rdx
; rdx = rax
lea rdx, [rax]
```

Листинг 7.4 – Сравнение MOV и LEA

Основное применение LEA — это вычисление адресов, но благодаря своей способности выполнять сложение и умножение (на 1, 2, 4, 8), она стала мощным инструментом для арифметических вычислений.

```
; rdx = rax + 4 * rbx + 16
lea rdx, [rax + rbx * 4 + 0x10]
```

Листинг 7.5 – LEA для вычисления адреса

7.2.1 LEA как оптимизация компилятора

Компиляторы часто используют LEA для выполнения простых арифметических операций, так как LEA часто выполняется быстрее, чем инструкции умножения (такие как `imul`). Например, для компиляции функции `a * 3`:

```
uint64_t Mul3(uint64_t a) {
    return a * 3;
}
```

Листинг 7.6 – C++ код для умножения на 3

Компилятор (`g++ -O2`) сгенерирует следующий код (листинг 7.7), используя LEA вместо умножения. В Linux (System V AMD64 ABI) первый аргумент (`a`) передается в регистре `rdi`, а возвращаемое значение — в `rax`.

$$a \times 3 = a \times (1 + 2) = a + a \times 2$$

Этот паттерн идеально ложится в SIB-адресацию: $[rdi + rdi \times 2]$.

```
0000000000000000 <Mul3(unsigned long)>:
   0: f3 0f 1e fa endbr64 ; Zashchitnaya instruktsiya
   4: 48 8d 04 7f lea rax,[rdi+rdi*2] ; rax = rdi + rdi * 2
   8: c3 ret
```

Листинг 7.7 – Результат компиляции Mul3 (objdump)

Итоги раздела

- `lea` вычисляет адрес, но не читает память.
- Это мощный инструмент для компактных арифметических вычислений, часто используемый компиляторами.

7.3 Работа со стеком и локальными переменными

У процессора ограниченное количество регистров. При вызове функции (инструкция `call`) возникает проблема: как сохранить значения локальных переменных, если вызываемая функция может перезаписать ("испортить") регистры?

7.3.1 Проблема: Callee-clobbered регистры

Согласно соглашениям о вызовах (Calling Conventions), большинство регистров (как `rax`, `rdx`, `rdi` и т.д.) являются *callee-clobbered* — вызываемая функция (*callee*) имеет право изменять их без восстановления.

Рассмотрим код, где мы храним `a` и `b` в регистрах:

```
mov rax, 1 ; a = 1
mov rdx, 2 ; b = 2
call f ; f()
add rax, rdx ; ??? (rdx mozhet byt' isporchen funktsiey f)
ret
```

Листинг 7.8 – Проблема сохранения локальных переменных

После возврата из `f`, мы не можем полагаться на то, что в `rdx` все еще лежит 2.

7.3.2 Решение 1: Сохранение на стеке

Основной механизм для сохранения локальных переменных — это стековый фрейм. Мы можем "зарезервировать"место на стеке, сдвинув указатель стека RSP, и сохранить туда наши значения.

```
mov rax, 1 ; a = 1
mov rdx, 2 ; b = 2

sub rsp, 16 ; Rezerviruem 16 bayt na steke
mov [rsp + 8], rdx ; Sokhranyaem b (po smeshcheniyu 8)
mov [rsp], rax ; Sokhranyaem a (na vershinu steka)
```

```
7
8  call f ; f()
9
10 ; VOSSTANOVLENIE
11 mov rax, [rsp] ; Vosstanavlivaem a
12 mov rdx, [rsp + 8] ; Vosstanavlivaem b
13 add rsp, 16 ; Osvobozhdaem mesto na steke
14
15 add rax, rdx ; Teper' bezopasno
16 ret
```

Листинг 7.9 – Использование стека для локальных переменных

7.3.3 Решение 2: Callee-saved регистры

Некоторые регистры, напротив, являются *callee-saved* (например, `rbx`, `rbp`). Это означает, что если вызываемая функция хочет их использовать, она *обязана* сохранить их значение (обычно на стеке) и восстановить перед выходом (`ret`). Компиляторы используют это для оптимизации.

Рассмотрим C++ код:

```
1 uint64_t f();
2 uint64_t g();
3 uint64_t Sum() {
4     return f() + g();
5 }
```

Листинг 7.10 – C++ код Sum()

Чтобы вычислить `g()`, нужно сначала вызвать `f()`, но результат `f()` (который вернется в `rax`) будет перезаписан результатом `g()`. Компилятор (листинг 7.11) решает эту проблему, сохраняя результат `f()` в callee-saved регистре `rbx`.

```
1 <Sum()>:
2    push rbx ; 1. Sokhranit' staroe znachenie rbx
3    call <f()> ; 2. Vyzvat' f(). Rezul'tat v rax
4    mov rbx, rax ; 3. Spryatat' rezul'tat f() v rbx
5    call <g()> ; 4. Vyzvat' g(). Rezul'tat v rax
6    add rax, rbx ; 5. rax = rax + rbx (rezul'tat g() + rezul'tat f())
7    pop rbx ; 6. Vosstanovit' staroe znachenie rbx
8    ret
```

Листинг 7.11 – Дизассемблированный код Sum() (g++ -O2)

Примечание

В листинг 7.11 мы видим `call` на адрес вроде `<Sum()+0xa>` (в реальном `objdump` это часто `call 0`). Это релокация. На этапе компиляции адрес функции `f` еще неизвестен. Компилятор оставляет "дырку"(часто 0), а компоновщик (linker) на финальном этапе сборки подставляет в это место реальный адрес функции.

7.4 Фреймовые указатели (Frame Pointers)

В листинг 7.9 мы вручную двигали RSP (`sub rsp, 16`) и обращались к переменным относительно RSP (`[rsp + 8]`). Это работает, но усложняет отладку и трассировку стека.

Определение: Фреймовый указатель (RBP)

Base Pointer, регистр-указатель на базу стекового фрейма (RBP) (Base Pointer) — это регистр, который по соглашению используется для хранения адреса *начала* текущего стековый фрейм. Это обеспечивает "стабильный"якорь для доступа к локальным переменным, даже если RSP постоянно движется (например, при `push`/`pop`).

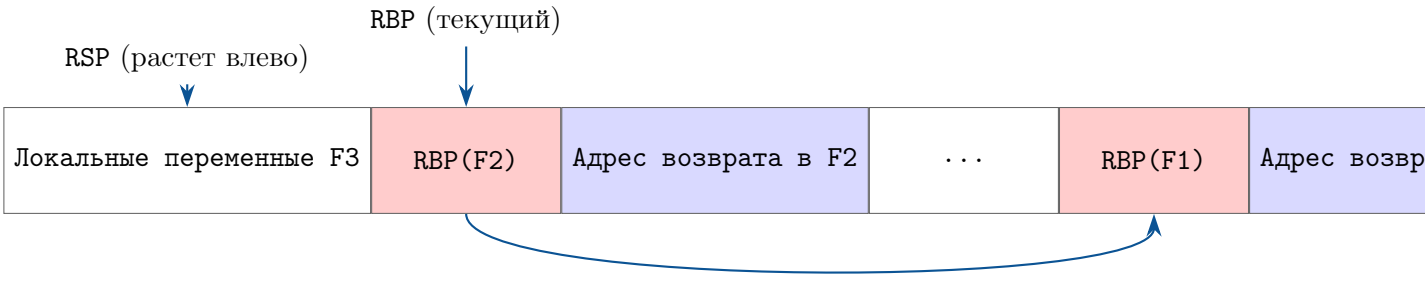
Для использования RBP применяется стандартный **пролог** (в начале функции) и **эпилог** (в конце).

```
f:
    ; --- PROLOG ---
    push rbp ; 1. Sokhranit' RBP predydushchey funktsii na stek
    mov rbp, rsp ; 2. Zapomnit' tekushchuyu vershinu steka kak bazu (nachalo)
                 ; nashego freyma. Teper' RBP stabilen.

    ; --- Telo funktsii ---
    ; Mesto dlya lokal'nykh peremennykh vydelyaetsya zdes'
    ; sub rsp, 32 ; (vydelit' 32 bayta)
    ; Dostup k peremennym idet otnositel'no RBP:
    ; mov [rbp - 8], rax

    ; --- EPILOG ---
    mov rsp, rbp ; 1. Osvobodit' vse lokal'nye peremennye, vernuv rsp k baze
    pop rbp ; 2. Vosstanovit' RBP predydushchey funktsii
    ret
```

Листинг 7.12 – Стандартный пролог и эпилог функции

7.4.1 Структура стека с RBP

Когда каждая функция использует этот пролог, значения RBP на стеке образуют **односвязный список**. Каждое сохраненное значение RBP указывает на RBP предыдущей (вызвавшей) функции.



Связный список фреймов

Рис. 7.1 – Структура стека при использовании фреймовых указателей (RBP)

Это позволяет отладчикам и другим инструментам легко "разматывать"стек (stack unwinding) и строить трассировку вызовов (stack trace).

Примечание

Использование RBP как фреймового указателя — это *соглашение*. Оно требует одного лишнего регистра и нескольких инструкций в прологе/эпилоге. Современные компиляторы (`g++ -O2`) по умолчанию часто отключают фреймовые указатели (`-fomit-frame-pointer`) для оптимизации. Вместо этого они генерируют специальную отладочную информацию (DWARF), которая позволяет разматывать стек, зная только RIP.

7.5 Секции данных в ассемблере

Ассемблерный код и данные не хранятся вперемешку. Они организованы в секции, которые сообщают операционной системе, как их следует загружать в память.

- **.text** — Код (инструкции). [cite: 216] Загружается с правами `Read-Only` и `Execute` (RX). [cite: 499]
- .rodata — Данные только для чтения. [cite: 217] (e.g., строковые литералы, константы). Загружаются с правами `Read-Only` (R). [cite: 501]
- .data — Инициализированные данные. [cite: 217] (e.g., глобальные переменные с начальным значением). Загружаются с правами `Read-Write` (RW). [cite: 505]
- .bss — Неинициализированные данные. [cite: 218] (e.g., `int x;`). Эти данные *не хранятся* в исполняемом файле, файл хранит только их размер. При загрузке ОС выделяет память и *обнуляет* ее. [cite: 507]

7.5.1 Директивы ассемблера для данных

Мы можем явно указать, в какую секцию помещать байты, с помощью директив.

```
; Ob"yavlyaem sektsiyu .rodata
.section .rodata

.global s1 ; Delaem metku s1 vidimoy dlya linkera
s1:
    .byte 0x48, 0x65 ; 'H', 'e'
    .ascii "ll" ; 'l', 'l'
    .asciz "o!" ; 'o', '!', i nulevoy bayt (terminator)
```

Листинг 7.13 – Пример секции .rodata (строка "Hello!")

В листинг 7.13 метка `s1` указывает на строку `"Hello!\0"`.

Другие директивы для инициализации данных в секциях .data или .rodata:

```
.data
    val1: .byte 0x10 ; 1 bayt
    val2: .short 0x1234 ; 2 bayta
    val3: .long 0x12345678 ; 4 bayta
    val4: .quad 0x1122334455667788 ; 8 bayt

    ; Zapolnit' 32 bayta znacheniem 0xFF
    buffer: .space 32, 0xFF

.bss
    ; Zarezervirovat' 1024 bayta (budut obnuleny)
```

```
12      big_buffer: .skip 1024
```

Листинг 7.14 – Директивы .data и .bss

Примечание

Все есть байты. В конечном счете, ассемблер просто транслирует мнемоники инструкций в байты в секции `.text`. Можно написать функцию, используя только директиву `.byte`, если знать машинные коды.

7.6 Флаги процессора и условные переходы

Большинство арифметических и логических инструкций (ALU) изменяют специальный регистр флагов (EFLAGS/RFLAGS). Условные переходы (`jcc`) анализируют эти флаги.

Основные флаги, интересующие нас:

- Carry Flag, флаг переноса (беззнаковое переполнение) (CF) (Carry Flag) — Установлен, если произошел перенос/заём из старшего бита (индикатор **беззнакового** переполнения). [cite: 313, 545]
- Zero Flag, флаг нуля (результат равен нулю) (ZF) (Zero Flag) — Установлен, если результат операции равен нулю. [cite: 314, 546]
- Sign Flag, флаг знака (установлен старший бит результата) (SF) (Sign Flag) — Установлен, если старший бит результата равен 1 (индикатор **отрицательного** числа в знаковой интерпретации). [cite: 315, 546]
- Overflow Flag, флаг переполнения (знаковое переполнение) (OF) (Overflow Flag) — Установлен, если произошло **знаковое** переполнение (e.g., 100+100 дало отрицательный результат в 8-битном знаковом представлении). [cite: 316, 547]

```
1  # 0b0001 + 0b0111 = 0b1000 (1 + 7 = 8)
2  # Rezul'tat (8) imeet bit znaka (SF=1).
3  # Proizoshlo znakovoe perepolnenie (1+7 != -8) (OF=1).
4  # Flagi: SF, OF
5
6  # 0b1001 + 0b0111 = 0b0000 (s perenosom) (9 + 7 = 16)
7  # Rezul'tat 0 (ZF=1).
8  # Proizoshlo bezznakovoe perepolnenie (CF=1).
9  # Flagi: ZF, CF
10
11 # 0b0010 - 0b0011 = 0b1111 (s zaemom) (2 - 3 = -1)
12 # Rezul'tat -1 (SF=1).
13 # Proizoshel bezznakovyy zaem (CF=1).
14 # Flagi: CF, SF
```

Листинг 7.15 – Примеры установки флагов (4-битная арифметика)

Инструкция `cmp` (compare) — это, по сути, `sub`, которая не сохраняет результат, а только устанавливает флаги.

После `cmp` (или `add`, `sub`, `and`...) используются инструкции условного перехода `jcc`:

- `je` / `jz` — Jump if Equal / Jump if Zero (проверяет ZF=1).
- `jne` / `jnz` — Jump if Not Equal / Jump if Not Zero (ZF=0).

- `js` — Jump if Sign (SF=1).
- `ja` — Jump if Above (беззнаковое "больше") (проверяет CF=0 и ZF=0).
- `jg` — Jump if Greater (знаковое "больше") (проверяет SF=OF и ZF=0).

7.7 Взаимодействие ассемблера и C/C++

Можно смешивать код на C/C++ и Ассемблер в одной программе, если соблюдать соглашения.

7.7.1 Позиционно-независимый код (PIC) и RIP-адресация

При попытке получить доступ к глобальной переменной (e.g., из C++ или секции .data) возникает проблема:

```
.data
my_var: .quad 123

.text
; OSHIBKA: Ne budet rabotat' v sovremennykh OS
mov rax, [my_var]
```

Листинг 7.16 – Наивный доступ к глобальной переменной

Проблема в том, что в современных ОС из соображений безопасности (ASLR — Address Space Layout Randomization) программа загружается в память по *случайному* адресу. [cite: 587-589] Мы не знаем абсолютный адрес `my_var` на этапе компиляции. [cite: 583]

Решение — Position-Independent Code, позиционно-независимый код (PIC) (PIC). Код не должен полагаться на абсолютные адреса, а только на *относительные*.

Определение: RIP-относительная адресация

В 64-битном режиме можно адресовать данные *относительно указателя инструкции* (RIP). Так как RIP всегда указывает на следующую исполняемую инструкцию, а `my_var` находится на *неизменном* расстоянии от этой инструкции (весь код и данные сдвигаются вместе), этот сдвиг остается константой. [cite: 593-594]

```
extern c ; Ob"yavlyaem metku 'c' vneshney (opredelena v C++)

.text
GetC:
    ; Korrektno: zagruzit' znachenie po adresu [rip + smeshchenie do 'c']
    mov rax, [c+rip]
    ret
```

Листинг 7.17 – Корректный доступ к глобальной переменной (PIC)

7.7.2 Оптимизация хвостового вызова (TCO)

Рассмотрим функцию-обертку, которая просто вызывает другую функцию и немедленно возвращает ее результат.

```
MyFuncWrapper:
    ; ... podgotovka argumentov ...
    call OtherFunc ; 1. Zapisat' adres vozvrata (A) na stek
    ret ; 2. Snyat' adres (A) so steka i pereyti na nego
```

Листинг 7.18 – Неоптимальный хвостовой вызов

Здесь `call` кладет на стек адрес возврата (в `MyFuncWrapper`), а `ret` немедленно его снимает. Это лишняя работа.

Tail Call Optimization, оптимизация хвостового вызова (TCO) (TCO) — это замена `call` + `ret` на один `jmp`.

```
MyFuncWrapper:
    ; ... podgotovka argumentov ...
    jmp OtherFunc ; Peredat' upravlenie OtherFunc
```

Листинг 7.19 – Оптимизированный хвостовой вызов (TCO)

Когда `OtherFunc` выполнит `ret`, она вернет управление не в `MyFuncWrapper`, а тому, кто вызвал `MyFuncWrapper` (т.к. его адрес возврата все еще лежит на вершине стека). [cite: 608] Компиляторы (`-O1` и выше) активно применяют эту оптимизацию.

7.7.3 Вызов функций C (scanf / printf)

Пользоваться вводом-выводом C++ (`iostream`) из Ассемблер почти невозможно из-за name mangling (искажения имен). [cite: 638-640] Гораздо проще использовать функции из C `<stdio.h>`, такие как `scanf` и `printf`. [cite: 643]

При этом нужно строго соблюдать два правила соглашения о вызовах (ABI): **1. Аргументы:** Первые 6 целочисленных аргументов/указателей передаются через регистры (именно в таком порядке): `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`.

2. Выравнивание стека: Перед инструкцией `call` RSP (указатель стека) **должен быть выровнен по 16-байтной границе**. [cite: 602]

Примечание

Ловушка выравнивания: Когда нашу функцию `main` вызывают, RSP уже выровнен по 16-байтной границе. *Но* инструкция `call` (которая вызвала `main`) помещает на стек 8-байтный адрес возврата. [cite: 603] Это означает, что *внутри* нашей функции `main` RSP **не выровнен** (он равен $16N + 8$). Перед тем, как мы сами сделаем `call` (например, `call scanf`), мы должны "скомпенсировать"эти 8 байт, например, `sub rsp, 8`. [cite: 604, 733]

```
.intel_syntax noprefix

.section .rodata
; Formatnaya stroka dlya chteniya ("%lld")
read_fmt: .asciz "%lld"
; Formatnaya stroka dlya zapisi ("%lld\n")
write_fmt: .asciz "%lld\n"

.text
.global main
main:
    ; --- PROLOG ---
    ; Vydelyaem 8 bayt dlya peremennoy 'n'
    ; I zaoctno VYRAVNIVAEM stek (rsp byl 16N+8, stal 16N)
    sub rsp, 8
```

```
16
17      ; --- Vyzov scanf ---
18      ; scanf("%lld", &n);
19      ; &n teper' = adres [rsp]
20
21      ; Arg 1 (RDI): Adres formatnoy stroki
22      lea rdi, [read_fmt+rip]
23      ; Arg 2 (RSI): Adres, kuda pisat' rezul'tat (vershina steka)
24      mov rsi, rsp
25
26      ; Dlya variadic funktsiy (kak scanf) nuzhno obnulit' rax
27      xor rax, rax
28      call scanf
29
30      ; --- Vyzov printf ---
31      ; printf("%lld\n", n + 1);
32      ; Zagruzhaem 'n' so steka
33      mov rsi, [rsp]
34      ; Uvelichivaem
35      add rsi, 1
36
37      ; Arg 1 (RDI): Adres formatnoy stroki
38      lea rdi, [write_fmt+rip]
39      ; Arg 2 (RSI): Znachenie (n + 1)
40      ; (uzhe v rsi)
41
42      xor rax, rax
43      call printf
44
45      ; --- EPILOG ---
46      ; "return 0;"
47      ; Po ABI, my vozvrashchaem znachenie iz main cherez RAX
48      xor rax, rax
49
50      ; Osvobozhdaem mesto na steke
51      add rsp, 8
52      ret
```

Листинг 7.20 – Пример: чтение числа (n) и вывод (n+1) на Assembler

Итоги раздела

- Для доступа к глобальным данным используйте **RIP-относительную адресацию** (`[my_var+rip]`).
- `call func` + `ret` можно заменить на `jmp func` (TCO).
- При вызове функций C (e.g., `printf`) стек **должен быть выровнен по 16 байт** *до* инструкции `call`.
- Аргументы передаются через `RDI`, `RSI`, `RDX`, `RCX`...
- `scanf` ожидает *указатель* (адрес) в `RSI`, `printf` — *значение*.
- Возвращаемое значение из `main` — это то, что лежит в `RAX` в момент `ret`.

7.8 Синтаксисы ассемблера: Intel vs. AT&T

Существует два доминирующих синтаксиса x86 Ассемблер.

- **Intel-синтаксис:** (Используется в этой лекции, в документации Intel, Microsoft).
- **AT&T-синтаксис (GNU):** (Используется по умолчанию в `objdump` и `gcc`). [cite: 688]

Ключевые отличия:

Таблица 7.1 – Сравнение синтаксисов Intel и AT&T (GNU)

Аспект	**Intel (мы)**	**AT&T (GNU)**
Порядок операндов	`mov rax, rbx` (Приемник, Источник)	`mov %rbx, %rax` (Источник, Приемник)
Регистры	`rax`, `rbx`	`%rax`, `%rbx` (с префиксом `%`)
Константы	`16`, `0x10`	`$16`, `$0x10` (с префиксом `$`)
Адресация	`[rax + rbx * 4 + 32]`	`32(%rax, %rbx, 4)`
Размер	`DWORD PTR [rax]`	`movl $0, (%rax)` (суффикс `l/q/w/b`)

Примечание

Полезно уметь читать оба синтаксиса. В `objdump` можно включить Intel-синтаксис с помощью флага `-M intel`.

Глава 8

8 Лекция

8.1 Оптимизация ассемблерного кода

Завершающая лекция по ассемблеру посвящена методам его эффективного использования, взаимодействию с ядром и компоновщиком, а также созданию программ, полностью независимых от стандартной библиотеки.

8.1.1 Проблема раздельной компиляции

Ранее мы рассматривали вызов ассемблерной функции из C++ с использованием раздельной компиляции. Этот подход имеет существенные недостатки производительности.

Рассмотрим пример с подсчётом суммы арифметической прогрессии. Если функция подсчёта реализована в C++ (и компилируется Clang), компилятор может распознать паттерн и заменить цикл на формулу $O(1)$. Если же функция вынесена в отдельный `.S` файл, компилятор видит только её объявление и вынужден генерировать цикл $O(n)$.

Более того, сам вызов функции между единицами трансляции (translation units) — это не бесплатная операция. Он включает:

- Инструкцию `call`, которая сохраняет адрес возврата в стек и выполняет переход (jump).
- Инструкцию `ret`, которая извлекает адрес из стека и выполняет косвенный переход (косвенный переход) по нему.

Эти операции вносят накладные расходы, которых можно избежать.

8.1.2 Встроенный ассемблер (GNU Inline Assembly)

Для устранения накладных расходов на вызов функции можно использовать встроенный ассемблер. Эта конструкция позволяет компилятору вставить ассемблерный код непосредственно в тело C++ функции, избегая `call`/`ret`.

Определение: Синтаксис GNU Inline Assembly

Конструкция `asm` в C/C++ (GCC, Clang) имеет следующий расширенный синтаксис:

```
asm [volatile] ("assembly template"
                : output operands    /* optional outputs */
                : input operands     /* optional inputs */
                : clobber list       /* optional clobbers */
);
```

- **assembly template:** Строковый литерал с ассемблерным кодом. Входы и выходы подставляются как `%0`, `%1` и т.д.
- **output operands:** Список переменных C/C++, в которые нужно записать результат.
- **input operands:** Список переменных/выражений C/C++, которые нужно передать в ассемблер.
- **clobber list:** Список регистров или состояний, которые изменяются (портятся) внутри вставки, о чём компилятор должен знать.

Рассмотрим пример сложения двух чисел (листинг 8.1).

```
1 long add_asm(long a, long b) {
2     long res;
```

```
    asm (
        "mov %[a_reg], %[res_reg]\n\t" // res = a
        "add %[b_reg], %[res_reg]\n\t" // res += b
        : [res_reg] "=&r" (res) // Output: res, in any register (r)
                                // & = early clobber
        : [a_reg] "r" (a), // Input: a, in any register (r)
          [b_reg] "r" (b) // Input: b, in any register (r)
        : "cc" // Clobbers: "cc" (condition codes / flags)
    );
    return res;
}
```

Листинг 8.1 – Использование inline asm для сложения

Операнды и ограничения (Constraints)

Компилятор не понимает семантику ассемблерного кода; для него это просто шаблон. Мы должны явно описать интерфейс между C++ и ассемблером с помощью ограничений:

- `"r"`: Поместить переменную в регистр общего назначения (например, `eax`, `rdi`).
- `-r"`: Выходной операнд (`=`), который будет в регистре.
- `"&r"`: Ограничение **Early Clobber** (`&`). Оно сообщает компилятору, что этот выходной регистр (`res_reg`) будет перезаписан *до* того, как все входные операнды (`a_reg`, `b_reg`) будут использованы. Это запрещает компилятору выделять один и тот же физический регистр для `res` и, например, `a`.

Список порчи (Clobbers)

Ассемблерная вставка может иметь побочные эффекты. Инструкция `add` изменяет регистр флагов (EFLAGS/RFLAGS). Если мы не сообщим об этом компилятору, он может ошибочно предположить, что флаги, установленные *до* `asm`-вставки, останутся неизменными *после* неё.

- `"cc"`: Сообщает компилятору, что регистр флагов (condition codes) был изменён.
- `"memory"`: Сообщает, что вставка читает или пишет в память по адресам, неизвестным компилятору. (См. раздел 8.2.3).
- `"rax"`, `"rcx"` и т.д.: Сообщает, что конкретный регистр был изменён.

8.1.3 Оптимизация на этапе компоновки (LTO)

Встроенный ассемблер решает проблему вызова, но не проблему оптимизации. Если функция `add` находится в другом `.cpp` файле, компилятор всё ещё не видит её реализацию при компиляции `main.cpp` и не может, например, заинлайнить её.

Определение: Link Time Optimization (LTO)

Link Time Optimization (оптимизация на этапе компоновки) (LTO) — это техника, при которой компилятор генерирует объектные файлы не в виде машинного кода, а в виде Intermediate Representation (промежуточное представление) (IR). На этапе компоновки (линковки) компилятор снова запускается, считывает IR из *всех* объектных файлов и выполняет оптимизации (включая инлайнинг, удаление мёртвого кода, константное сворачивание) так, как если бы весь код находился в одной единице трансляции.

Инфраструктура Low Level Virtual Machine (инфраструктура для построения компиляторов) (LLVM) (используемая Clang) идеально для этого подходит.

- **Без LTO:** `clang++ -O2` → `main.o` (x86-64), `add.o` (x86-64). Линкер просто склеивает их.
- **С LTO (`-flto`):** `clang++ -flto -O2` → `main.o` (LLVM IR), `add.o` (LLVM IR). На этапе линковки `clang` видит IR обеих функций, инлайнит `add` в `main` и может применить оптимизацию (например, свернуть сумму арифметической прогрессии в константу).

Примечание

Объектные файлы LTO, сгенерированные Clang, не являются стандартными ELF-файлами с машинным кодом. Это архивы LLVM IR (LLVM-AR). Для их просмотра вместо `objdump` используется `llvm-dis`. GCC также поддерживает LTO, но обычно встраивает своё IR (GIMPLE) в специальные секции ELF-файлов.

Итоги раздела

- Вызовы функций между `.cpp` и `.S` файлами несут накладные расходы (`call`/`ret`).
- встроенный ассемблер позволяет встроить ассемблерный код в C++, устраняя эти расходы, но требует аккуратного описания интерфейса (входы, выходы, clobbers (список порчи)).
- LTO позволяет компилятору оптимизировать код *между* единицами трансляции, генерируя промежуточное IR вместо машинного кода.

8.2 Взаимодействие с ядром и побочные эффекты

8.2.1 Прямой вызов `syscall`

Ассемблерные вставки позволяют нам выполнять Системный вызов напрямую, минуя обёртки стандартной библиотеки (libc).

Определение: Соглашение о `syscall` в Linux x86-64

- Инструкция: `syscall`.
- Номер системного вызова: передаётся в `RAX`.
- Аргументы (по порядку): `RDI`, `RSI`, `RDX`, `R10`, `R8`, `R9`.
- Возвращаемое значение: в `RAX`.
- **Порча:** Инструкция `syscall` *уничтожает* содержимое `RCX` и `R11`.

Примечание

Соглашение о `syscall` отличается от стандартного System V ABI в 4-м аргументе (`R10` вместо `RCX`). Это связано с тем, что `syscall` использует `RCX` для сохранения адреса возврата (RIP) и `R11` для сохранения RFLAGS, чтобы ядро могло вернуться в пользовательский процесс с помощью инструкции `sysret`.

В листинг 8.2 показан пример вызова `write` (номер 1) для печати "Hello".

```
#include <sys/syscall.h> // for SYS_write
#include <unistd.h> // for STDOUT_FILENO

long write_syscall(int fd, const char* buf, size_t count) {
    long ret;
    asm volatile (
        "syscall"
        : "=a" (ret) // Output: in RAX (a)
        : "a" (SYS_write), // Input: syscall number in RAX (a)
          "D" (fd), // Input: arg1 in RDI (D)
          "S" (buf), // Input: arg2 in RSI (S)
          "d" (count) // Input: arg3 in RDX (d)
        : "rcx", "r11", "memory" // Clobbers: syscall clobbers rcx, r11
                                 // "memory" because 'buf' is read
    );
    return ret;
}

int main() {
    const char* msg = "Hello from syscall!\n";
    write_syscall(STDOUT_FILENO, msg, 20);
    return 0;
}
```

Листинг 8.2 – Системный вызов `write` через inline asm

8.2.2 `volatile` и побочные эффекты

Что произойдёт, если мы вызовем `write_syscall`, но не будем использовать возвращаемое значение (`ret`)?

```
int main() {
    const char* msg = "Hello...\n";
    // The compiler (with -O2) might DELETE this line!
    write_syscall(STDOUT_FILENO, msg, 10);
    return 0;
}
```

Листинг 8.3 – Проблема оптимизации

С точки зрения компилятора, функция `write_syscall` (листинг 8.2 без `volatile`) — это "чёрный ящик который принимает 4 аргумента и возвращает `long`. Если этот `long` не используется, компилятор вправе удалить вызов целиком, следуя правилу "as-if"(программа должна вести себя *так, как если бы* она выполнялась).

Проблема в том, что у `syscall` есть **побочный эффект** (вывод на экран), о котором компилятор не знает.

Определение: asm `volatile`

Ключевое слово volatile перед `asm` (`asm volatile (...)`) запрещает компилятору:

1. **Удалять** эту ассемблерную вставку, даже если её выходные операнды не используются.
2. **Переупорядочивать** её относительно других `volatile` операций (например, доступа к `volatile` переменным).

Это необходимо для всех ассемблерных вставок, имеющих побочные эффекты (side effects), такие как системные вызовы.

8.2.3 Использование `asm` для барьеров компиляции

Конструкцию `asm volatile` можно использовать для управления оптимизациями компилятора.

Барьер оптимизации (`DoNotOptimize`)

Иногда в бенчмарках нужно C++ значение, чтобы компилятор не "выкинул"всё вычисление этого значения.

```
// Helper function, similar to Google Benchmark
template <class T>
void DoNotOptimize(T const& value) {
    // The asm block does nothing, but "reads" 'value'
    // 'm' = memory operand
    asm volatile("" :: "m" (value) : "memory");
}

// ...
long result = complex_calculation();
DoNotOptimize(result); // Now the compiler MUST
                       // compute 'result'
```

Листинг 8.4 – Запрет оптимизации переменной

Барьер памяти (Compiler Fence)

```
asm volatile("" ::: "memory");
```

Листинг 8.5 – Барьер памяти компилятора

Список clobbers (список порчи), содержащий `"memory"`, сообщает компилятору, что эта вставка может читать или писать в *любую* ячейку памяти. Это заставляет компилятор:

- **Сбросить** (spill) все значения из регистров, которые были изменены, обратно в память *до* этой вставки.
- **Загрузить** (reload) значения из памяти *после* этой вставки, если они понадобятся, не полагаясь на кэшированные в регистрах значения.

Это барьер *только для компилятора*, он не генерирует инструкций барьера CPU (типа `mfence`).

Итоги раздела

- Системные вызовы в Linux x86-64 выполняются инструкцией `syscall`, используя регистры `RAX`, `RDI`, `RSI`, `RDX`, `R10`...
- Инструкция `syscall` портит `RCX` и `R11`.
- `asm volatile` необходимо использовать, когда вставка имеет побочные эффекты (как `syscall`), чтобы компилятор её не удалил.
- `asm volatile` с `memory` в clobbers служит барьером памяти для компилятора.

8.3 Указатели, функции и полиморфизм

Знание ассемблера позволяет понять, как реализованы высокоуровневые конструкции C++, такие как указатели на функции и виртуальные методы.

8.3.1 Указатели на функции и косвенные переходы

Указатель на функцию в C++ — это переменная, хранящая адрес.

```
int f1(int x) { return x; }
int f2(int x) { return x * 2; }
int f3(int x) { return x * 3; }

// Syntax: ret_type (*var_name)(arg_types)
int (*fn_array[])(int) = { f1, f2, f3 };

int main() {
    int index = 1; // Assume this came from user input
    // ...
    int result = fn_array[index](5); // calls f2(5)
}
```

Листинг 8.6 – Массив указателей на функции

Во что транслируется `fn_array[index](5)`?

1. Загрузка адреса из `fn_array[index]` в регистр (например, `rax`).
2. Загрузка аргумента `5` в `rdi`.
3. Выполнение косвенный переход: `call rax`.

Определение: Указатель на функцию и косвенный переход

Численное значение указателя на функцию — это, как правило, адрес первой инструкции этой функции в секции `.text`. Вызов по такому указателю реализуется CPU через **косвенный вызов** (`call <reg>`), адрес которого неизвестен на этапе компиляции.

8.3.2 Защита от атак: `endbr64`

Косвенные переходы — основной вектор атак (ROP/JOP), когда злоумышленник получает контроль над регистром (`rax`) и заставляет программу прыгнуть не на начало функции, а в середину другой функции (на "гаджет").

Для борьбы с этим в современных CPU (Intel CET) введена инструкция endbr64.

- Компиляторы (GCC/Clang) теперь вставляют `endbr64` в начало каждой функции.
- Если ОС и CPU включают защиту, любой косвенный переход (`call rax`), который приземляется *не* на инструкцию `endbr64`, вызовет аппаратное исключение (fault).
- Это гарантирует, что косвенные вызовы могут приземляться только на легитимные начала функций.

Примечание

На данный момент (в лекции) Linux использует эту защиту в основном для кода ядра, но не для пользовательских (userspace) приложений. Однако компиляторы всё

равно генерируют `endbr64` для совместимости в будущем.

8.3.3 Реализация виртуальных функций C++

Динамический полиморфизм в C++ (ключевое слово `virtual`) также построен на косвенных вызовах.

Определение: vptr и vtable

- **vtable (Таблица виртуальных методов):** Статический массив указателей на функции, создаваемый компилятором для *каждого класса*, имеющего виртуальные методы.
- **vptr (Указатель на vtable):** Скрытый указатель, добавляемый компилятором в *каждый объект* такого класса. vptr указывает на vtable, соответствующую реальному типу объекта.

Рассмотрим вызов `a->foo()`:

```
struct A {
    virtual void foo() { /* A's foo */ }
};
struct B : A {
    virtual void foo() override { /* B's foo */ }
};

int main() {
    A* a = new B();
    a->foo(); // <-- How does this work?
}
```

Листинг 8.7 – Виртуальный вызов

Вызов `a->foo()` (где `a` в `rdi`) транслируется в:

1. `mov rax, [rdi]` ; Загрузить vptr из объекта (this) в `rax`
2. `call [rax]` ; Вызвать функцию по первому адресу в vtable

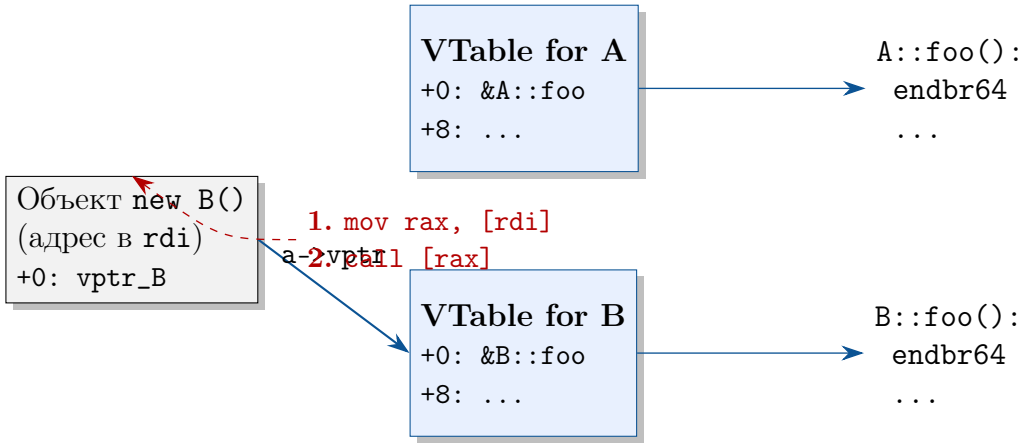


Рис. 8.1 – Схема виртуального вызова через vptr и vtable

Цена виртуализации

- **Цена по памяти:** +8 байт на *каждый* объект (vptr). Это нарушает принцип C++ "платишь только за то, что используешь т.к. вы платите за vptr, даже если никогда не делаете виртуальных вызовов.
- **Цена по времени:** Виртуальный вызов требует двух обращений к памяти (чтение vptr, чтение адреса из vtable) и косвенный переход, что медленнее прямого `call`.

8.3.4 JIT-компиляция (Just-in-Time)

Зная, что код — это просто байты в памяти, мы можем генерировать его во время выполнения.

Определение: JIT-компиляция

Just-in-Time (компиляция «на лету») (JIT) — это техника, при которой машинный код генерируется не на этапе компиляции, а во время выполнения программы. Это позволяет создавать код, оптимизированный под конкретные данные (например, SQL-запрос в ClickHouse) или под конкретное железо (например, используя AVX-инструкции, если они доступны на CPU пользователя).

Процесс JIT в Linux:

1. Выделить память с помощью `mmap` с правами `PROT_READ | PROT_WRITE`.
2. Записать в эту память байты машинного кода.
3. Изменить права памяти с помощью `mprotect` на `PROT_READ | PROT_EXEC` (W⊕X).
4. Преобразовать указатель на эту память в указатель на функцию.
5. Вызвать сгенерированную функцию.

```
#include <sys/mman.h> // mmap, mprotect
#include <string.h> // memcpy

// Bytes for the function:
// mov rax, rdi (48 89 f8)
// add rax, rsi (48 01 f0)
// ret (c3)
unsigned char code[] = { 0x48, 0x89, 0xf8, 0x48, 0x01, 0xf0, 0xc3 };

typedef long (*add_func_t)(long, long);

int main() {
    void* mem = mmap(NULL, sizeof(code),
                     PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    memcpy(mem, code, sizeof(code));

    // Important: make the memory executable
    mprotect(mem, sizeof(code), PROT_READ | PROT_EXEC);

    add_func_t fn = (add_func_t)mem;
    long result = fn(10, 20); // result == 30
```

```
24
25      munmap(mem, sizeof(code));
26      return 0;
27  }
```

Листинг 8.8 – Ручная JIT-компиляция функции `add(a, b)`

Примечание

Приложения вроде ClickHouse или JVM не пишут байты вручную. Они встраивают в себя бэкенд компилятора (например, LLVM) и используют его API для генерации оптимизированного кода "на лету".

Итоги раздела

- Указатели на функции реализуются через косвенный переход (`call rax`).
- Инструкция endbr64 защищает от атак, помечая легитимные цели для таких переходов.
- Виртуальные вызовы C++ используют vptr (в объекте) и vtable (на класс) для реализации косвенный переход, что несёт расходы памяти и времени.
- JIT-компиляция позволяет генерировать машинный код во время выполнения с помощью `mmap` и `mprotect`.

8.4 Динамическая компоновка

динамическая библиотека (.so) — это код, который компонуется с программой не при сборке, а при запуске.

8.4.1 Мотивация и основы (.so)

Две основные причины для использования динамическая библиотека (.so):

1. **Экономия памяти:** Множество программ (bash, ls, g++) используют одну и ту же стандартную библиотеку (libc, libstdc++). Вместо того чтобы каждый процесс загружал свою копию, ОС загружает `.so` в память один раз и отображает её в адресные пространства всех процессов.
2. **Оптимизация под платформу:** Можно иметь несколько реализаций `memcpy` (обычную, SSE, AVX) и при запуске загрузчик выберет ту `.so`, которая оптимизирована под текущий CPU.

Загрузчик (`ld.so` в Linux) отвечает за поиск и загрузку всех зависимостей (их можно посмотреть командой `ldd a.out`) перед запуском `_start`.

Позиционно-независимый код (PIC)

Динамическая библиотека не знает, по какому адресу она будет загружена в виртуальную память. Поэтому её код не может использовать абсолютную адресацию.

Определение: Position Independent Code (PIC)

PIC — это код, который использует **относительную адресацию** (в x86-64 — относительно регистра `RIP`) для всех переходов и доступа к данным. Это позволяет загружать `.so` в любое место в памяти без необходимости её модификации. Для сборки PIC используется флаг `-fPIC`.

8.4.2 Механизмы PLT и GOT

Как `main` (скомпилированный) может вызвать `printf` (адрес которой станет известен только при запуске)?

Определение: GOT и PLT

- Global Offset Table (глобальная таблица смещений) (GOT) **(Global Offset Table):** Глобальная таблица смещений. Это массив в секции данных, хранящий *реальные адреса* внешних функций и переменных.
- Procedure Linkage Table (таблица компоновки процедур) (PLT) **(Procedure Linkage Table):** Таблица компоновки процедур. Это секция *исполняемого* кода, содержащая "трамплины"(stubs) — по одному на каждую внешнюю функцию.

По умолчанию в Linux используется ленивое связывание (ленивое связывание).

Процесс первого вызова `printf`:

1. `main` вызывает не `printf`, а трамплин `printf@plt`.
2. Трамплин `printf@plt` прыгает на адрес, указанный в GOT для `printf`.
3. *Изначально* GOT указывает не на `printf`, а обратно на код в PLT.

4. Этот код в PLT кладёт ID функции (`printf`) в стек и прыгает на **динамический резолвер** (часть `ld.so`).
5. Резолвер находит реальный адрес `printf` в загруженной `libc.so`.
6. **(Патчинг)** Резолвер *перезаписывает* запись `printf` в GOT, указывая на реальный адрес.
7. Резолвер прыгает на реальный `printf`.

Второй и последующие вызовы `printf`:

1. `main` вызывает `printf@plt`.
2. Трамплин `printf@plt` прыгает на адрес, указанный в GOT.
3. GOT *уже* содержит реальный адрес `printf`. Происходит прямой переход к `printf`, минуя резолвер.

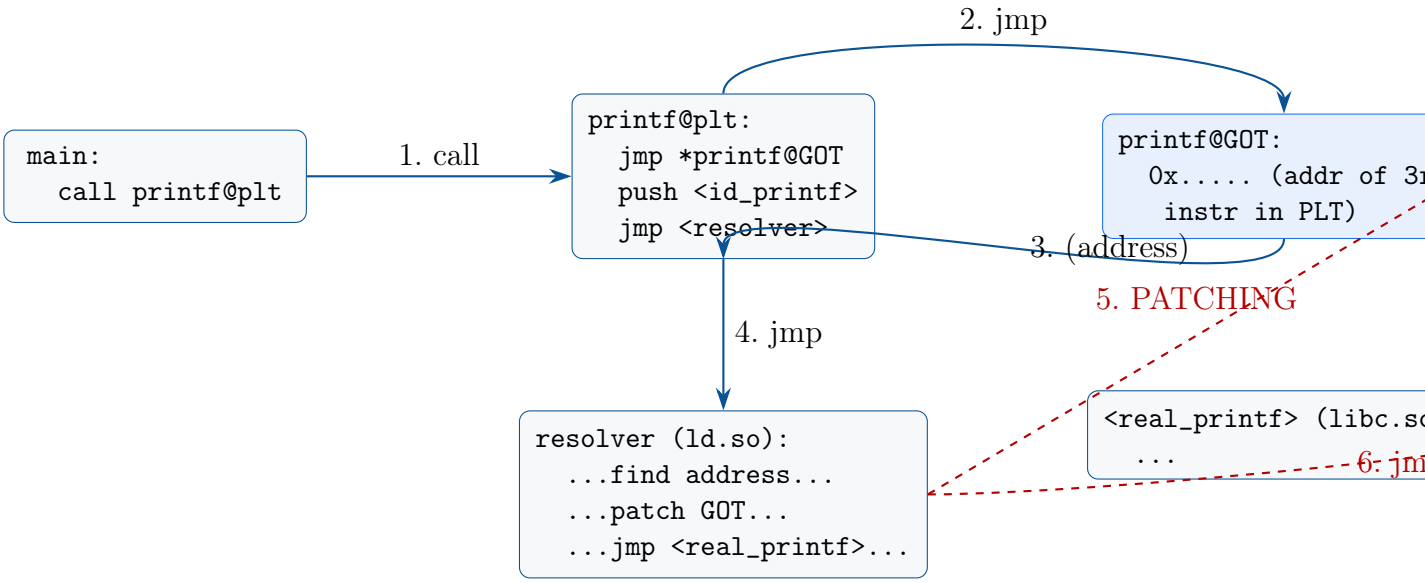


Схема ленивое связывание при *первом* вызове

Рис. 8.2 – Процесс ленивого связывания (Lazy Binding)

8.4.3 Перехват вызовов (LD_PRELOAD)

`LD_PRELOAD` — это переменная окружения Linux, которая указывает загрузчику `ld.so`, какую динамическая библиотека (.so) загрузить *в первую очередь*, до `libc` и всех остальных.

Если мы создадим свою `libmyhack.so`, в которой определим функцию `printf`, и запустим программу:

```
$ LD_PRELOAD=./libmyhack.so ./a.out
```

Когда резолвер `ld.so` будет искать `printf`, он сначала найдёт *нашу* реализацию в `libmyhack.so` и использует её.

Примечание

Компилятор может заменять небезопасные функции (как `printf`) на их "проверяющие"аналоги (например, `__printf_chk`) для защиты от переполнения буфера. Если вы хотите перехватить `printf`, вам, возможно, придётся перехватывать `__printf_chk`.

8.4.4 Ручная загрузка библиотек (dlopen)

Программа может сама загружать `.so` во время выполнения, используя API из `libdl`.

- `dlopen(const char* path, int mode)`: Загружает `.so`. Возвращает `void*` "handle".
- `dlsym(void* handle, const char* symbol)`: Ищет символ (функцию или переменную) по имени в загруженной библиотеке. Возвращает `void*`.
- `dlclose(void* handle)`: Выгружает библиотеку.
- `dlerror()`: Возвращает строку с описанием последней ошибки.

```c
#include <dlfcn.h>
#include <stdio.h>

// 1. Define the signature of the function we're looking for
typedef double (*sin_func_t)(double);

int main() {
    // 2. Load the library
    void* handle = dlopen("libm.so.6", RTLD_LAZY);
    if (!handle) { /* error handling */ }

    // 3. Find the symbol (function)
    void* sym = dlsym(handle, "sin");
    if (!sym) { /* error handling */ }

    // 4. Cast void* to the correct FUNCTION POINTER type
    sin_func_t my_sin = (sin_func_t)sym;

    // 5. Use it
    double result = my_sin(1.0); // ~0.841
    printf("sin(1.0) = %f\n", result);

    // 6. Close it
    dlclose(handle);
    return 0;
}
```

Листинг 8.9 – Ручная загрузка `libm.so` для вызова `sin`

Примечание

[title=Внимание!] Ошибка в сигнатуре функции при касте `dlsym` (листинг 8.9, шаг 4) — это тяжёлое **Undefined Behavior**. Если `sin` ожидает `double`, а вы вызовете его с `int`, это почти гарантированно приведёт к падению из-за нарушения соглашения о вызовах (аргументы будут лежать не в тех регистрах, `XMM0` vs `RDI`).

Итоги раздела

- Динамические библиотеки (`.so`) экономят память и позволяют подменять реализации.
- Они должны быть скомпилированы как PIC (`-fPIC`) для относительной адресации.
- PLT и GOT — механизмы, позволяющие вызывать функции, адреса которых неизвестны до запуска.
- ленивое связывание (по умолчанию) разрешает адрес функции при первом вызове через PLT.
- `LD_PRELOAD` позволяет перехватывать вызовы, подгружая свою `.so` первой.
- `dlopen` и `dlsym` позволяют программе вручную загружать плагины (`.so`) во время работы.

8.5 Freestanding: Программы без `stdlib`

Мы научились делать Системный вызов сами. Теперь мы можем полностью отказаться от стандартной библиотеки C/C++.

8.5.1 `hosted` vs `freestanding`

- **Hosted:** Стандартный режим. Компилятор предполагает наличие ОС и `stdlib`. Доступны `main`, `malloc`, `printf`, `std::vector` и т.д.
- **Freestanding:** Режим, в котором не предполагается наличие `stdlib`. Нельзя использовать `malloc`, I/O, исключения, RTTI (если они требуют поддержки `stdlib`). Это окружение для ядер ОС, драйверов, микроконтроллеров.

Чтобы собрать программу в `freestanding` режиме, используются флаги:

```
$ g++ -ffreestanding -nostdlib my_program.cpp -o my_program
```

8.5.2 Точка входа `_start`

`main` — это *не* точка входа в программу. Это просто функция, которую вызывает код *инициализации* из `stdlib` (например, `crt0.o`).

Настоящая точка входа, куда ядро Linux передаёт управление, — это метка *_start*. Мы должны определить её сами, обычно на ассемблере.

Состояние при запуске

Когда ядро запускает *_start*, CPU находится в следующем состоянии:

- `RSP` (указатель стека) 16-байтно выровнен.
- `RBP`, по соглашению, должен быть обнулён (`xor rbp, rbp`). Это используется дебаггерами и бэктрейсерами как маркер конца цепочки стековых кадров.
- Стек содержит аргументы и переменные окружения (рис. 8.3).

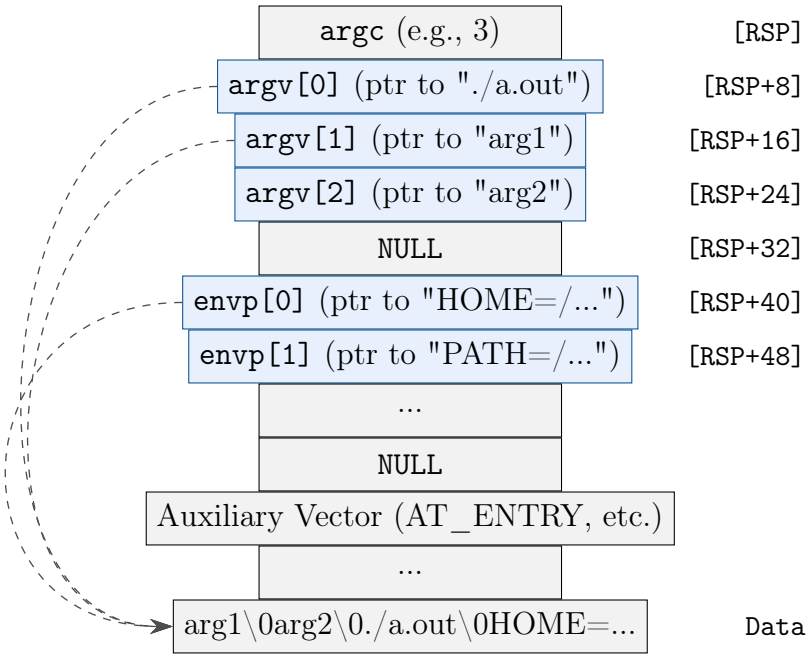


Рис. 8.3 – Содержимое стека при вызове `_start`

Пример _start

листинг 8.10 показывает минимальную `freestanding` программу.

```
.section .rodata
msg:
    .string "Hello, freestanding!\n"
msg_end:
    .equ msg_len, msg_end - msg

.section .text
.global _start

_start:
    # Convention: zero out RBP for backtracing
    xor %rbp, %rbp

    # syscall: write(1, msg, msg_len)
    mov $1, %rax # SYS_write
    mov $1, %rdi # fd (stdout)
    mov $msg, %rsi # buf
    mov $msg_len, %rdx # count
    syscall

    # syscall: exit(123)
    mov $60, %rax # SYS_exit
    mov $123, %rdi # exit_code
    syscall
```

Листинг 8.10 – Freestanding "Hello World"(AT&T синтаксис)

Примечание

Компилятор всё ещё может генерировать вызовы `memcpy`, `memset` и т.д. для оптимизации C++ кода (например, копирования структур). В настоящей `freestanding` среде вам пришлось бы предоставить реализации и этих функций.

8.5.3 Загрузка и расширение знака

При работе с ассемблером важно помнить о размерах данных. Ошибка из прошлой лекции: чтение 32-битного `int` в 64-битный регистр.

- `mov %eax, [%rsp]` (AT&T): Загружает 32 бита из памяти в `EAX`. *При этом старшие 32 бита* ***RAX*** *обнуляются.*
- `mov %rax, [%rsp]`: Загружает 64 бита.

Проблема: если мы читаем 32-битное отрицательное число (`0xFFFFFFFF`, т.е. -1) с помощью `mov %eax, ...`, `RAX` станет `0x00000000FFFFFFFF` (положительное число $\sim$ 4 млрд).

Определение: Инструкции расширения знака

- movsx / movzx (Move with Sign Extend) / `movslq` (Move Sign-extend Long to Quad): Копирует знаковый бит (старший бит) источника во все старшие биты приёмника.
- `movzx` (Move with Zero Extend) / `movzb/w...`: Заполняет старшие биты приёмника

нулями.

```
# Load a 32-bit dword from [rsp] into 64-bit rax
# with sign extension.
# Intel: movsx rax, dword ptr [rsp]
# AT&T:
movslq (%rsp), %rax
```

Итоги раздела

- `freestanding` режим позволяет писать код без `stdlib`.
- Точка входа в ELF — это *_start*, а не `main`.
- Ядро передаёт `argc`, `argv` и `envp` через стек.
- `_start` должна обнулить `RBP` (`xor %rbp, %rbp`).
- При загрузке 32-битных знаковых чисел в 64-битные регистры необходимо использовать movsx / movzx (`movslq`) для сохранения знака.

8.6 Введение в архитектуру процессора

Мы научились генерировать инструкции, но почему они выполняются так быстро? Современные CPU — это сложные системы, скрывающие огромную задержку (latency) доступа к памяти.

8.6.1 Проблема доступа к памяти и кэши

1. **Виртуальная память:** Разыменование указателя (виртуального адреса) требует 4-5 обращений к памяти для прохода по таблицам страниц (Page Tables).
2. **DRAM:** Обращение к основной памяти (DRAM) занимает ~100 наносекунд.

Итого ~500 нс (тысячи тактов CPU) на *каждый* доступ к памяти.

Решение 1: TLB

Translation Lookaside Buffer (буфер ассоциативной трансляции) (TLB) — это маленький, очень быстрый кэш внутри CPU, который хранит недавние отображения "виртуальная страница → физическая страница". Если в TLB есть попадание (hit), CPU избегает 4-5 обращений к памяти.

Решение 2: Иерархия кэшей L1/L2/L3

TLB решает проблему трансляции, но кэш решает проблему медленной DRAM.

- **L1 (32-64KB):** ~1 нс (несколько тактов). Обычно разделён на L1i (инструкции) и L1d (данные).
- **L2 (256KB-4MB):** ~4-12 нс.
- **L3 (8MB+):** ~30-50 нс. (Общий для всех ядер).
- **DRAM:** ~100+ нс.

Организация кэша

- **кэш-линия:** Данные передаются не побайтно, а блоками по 64 байта.
- **ассоциативный кэш:** Кэш организован как хэш-таблица.

Физический адрес разбивается на три части: `[ TAG (36 бит) | SET_INDEX (6-10 бит) | OFFSET (6 бит) ]`

1. **OFFSET (0-5):** Байт внутри 64-байтной кэш-линия.
2. **SET_INDEX (6-11):** Индекс "корзины"(set) в хэш-таблице.
3. **TAG (12-47):** Уникальный идентификатор кэш-линия.

Примечание

Простая хэш-функция (средние биты адреса) — это проблема. Если программа часто обращается к адресам с шагом, кратным большой степени двойки (например, `arr[i * 1024]`), все эти обращения могут попасть в *один и тот же set*, "выбивая"друг друга из кэша, даже если кэш L1 в целом пуст.

8.6.2 Конвейер инструкций (Pipeline)

Для сокрытия задержек (даже L1) CPU исполняет инструкции в конвейер (например: Fetch → Decode → Execute → Memory → Writeback). Исполнение нескольких инструкций перекрывается во времени.

Конфликты (Hazards)

- **конфликт по данным:** Инструкция N (напр. `add`) ждёт результат инструкции N-1 (напр. `mov`). Пример: итерация по связному списку (`node = node->next`) — это чистый конфликт по данным, т.к. следующий `mov` зависит от предыдущего `mov`.
- **конфликт по управлению:** Условный переход (`je`, `jne`). CPU не знает, какую инструкцию загружать (Fetch) следующей, пока не выполнится (Execute) `cmp`.

Продвинутые оптимизации CPU

1. **Out-of-Order Execution (внеочередное исполнение) (OoOE):** CPU может исполнять инструкции не по порядку, если они не зависят друг от друга, чтобы "заполнить"простои (например, во время конфликт по данным или промаха кэша).
2. **Переименование регистров:** CPU имеет сотни *физических* регистров, но только 16 *архитектурных* (`rax`...). CPU динамически переименовывает `rax` в `phys_reg_5` в одной инструкции и в `phys_reg_28` в другой, чтобы разорвать ложные зависимости по данным.
3. **предсказание ветвлений:** Для решения конфликт по управлению, CPU *угадывает* результат `je` и спекулятивно исполняет код.

Пример: Предсказатель ветвлений

Рассмотрим код (даже на Python) для подсчёта элементов в массиве:

```python
import numpy as np
data = np.random.randint(0, 256, size=100000)
# data.sort() # <--- The key line

count = 0
for x in data:
    if x < 128: # <--- The conditional branch
        count += 1
```

Листинг 8.11 – Тест предсказателя ветвлений

- **Несортированный массив:** `if x < 128` непредсказуем. Предсказатель ошибается в ~50% случаев.
- **Сортированный массив:** `if` всегда `True` для первой половины, всегда `False` для второй. Предсказатель ошибается *только один раз* (когда `True` меняется на `False`).

Результат: код на сортированном массиве работает *значительно* (в 5-10 раз) быстрее из-за почти 100% точности предсказание ветвлений.

Итоги раздела

- Доступ к DRAM очень медленный (~100 нс).
- TLB кэширует трансляцию виртуальных адресов в физические.
- Кэши L1/L2/L3 кэшируют сами данные из DRAM.
- Данные ходят кэш-линия по 64 байта.
- конвейер перекрывает исполнение инструкций.

- OoOE, переименование регистров и предсказание ветвлений — ключевые техники CPU для сокрытия задержек и решения конфликт по данным и конфликт по управлению.

Глава 9

9 Лекция

9.1 Оптимизации в современных процессорах

Современные 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.

9.1.1 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.

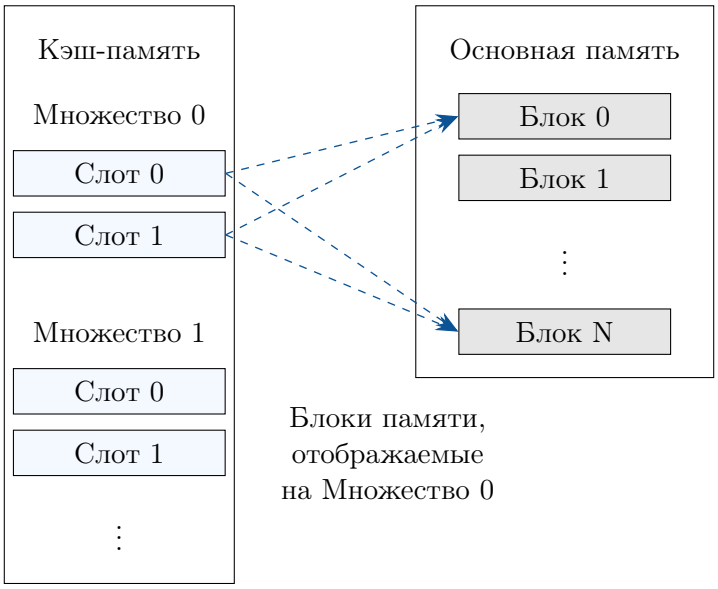


Рис. 9.1 – Схема 2-ассоциативного кэша: любой блок памяти, чей адрес отображается на Множество 0, может быть помещён в любой из двух слотов этого множества.

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

9.1.2 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.

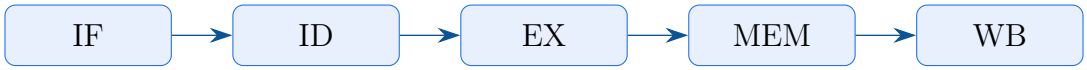


Рис. 9.2 – Классический 5-стадийный конвейер обработки инструкций

Современные процессоры идут дальше и реализуют 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.

9.1.3 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**.

9.1.4 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.

9.2 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.

9.2.1 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.

9.2.2 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:

$$\text{fp} = S \cdot M \cdot 2^E \tag{9.2.1}$$

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.

$$E_{\text{real}} = E_{\text{stored}} - \text{bias} \tag{9.2.2}$$

9.2.3 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.

9.2.4 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 9.1 – 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 9.1 – 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.

9.3 Основы многопоточности

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

9.3.1 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 9.2 – Создание и запуск потока в C++

9.3.2 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++.

9.3.3 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 9.3 – Безопасный инкремент с помощью std::atomic

9.3.4 Примитивы блокирующей синхронизации

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

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.

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.

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.

9.4 Классическая проблема: обедающие философы

Эта задача иллюстрирует проблему взаимоблокировка в системах с разделяемыми ресурсами.

9.4.1 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.

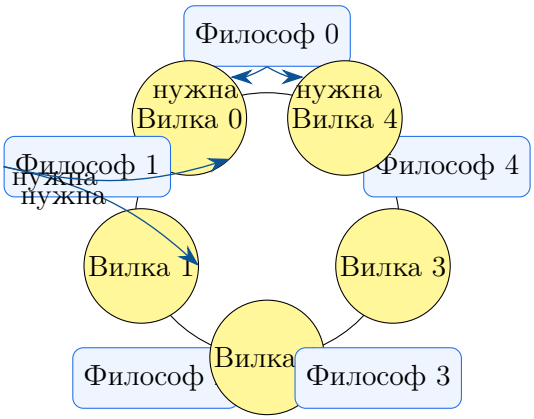


Рис. 9.3 – Схема расположения философов и вилок. Каждому философу для еды нужны две соседние вилки.

9.4.2 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).

Глава 10

10 Лекция

10.1 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.

10.1.1 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.

10.2 Аппаратная поддержка многопоточности

10.2.1 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.

10.2.2 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:

```
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.

10.2.3 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:

$$\text{Speedup}(T) = \frac{1}{S + \frac{P}{T}} \tag{10.2.1}$$

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

$$\text{Max Speedup} = \frac{1}{S} = 1 + \frac{P}{S} \tag{10.2.2}$$

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

10.3 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.

10.3.1 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).

10.3.2 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`.

10.4 Примитивы синхронизации

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

10.4.1 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`.

10.4.2 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.

10.4.3 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
}
```

10.5 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.

10.6 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‘.

10.7 Атомики и модель памяти

10.7.1 Compare-And-Swap (CAS)

Мощная атомарная операция для реализации lock-free алгоритмов. В C++: `compare_exchange_strong` 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.

10.8 Кэши и когерентность

10.8.1 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.

10.8.2 False Sharing (Ложное разделение)

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

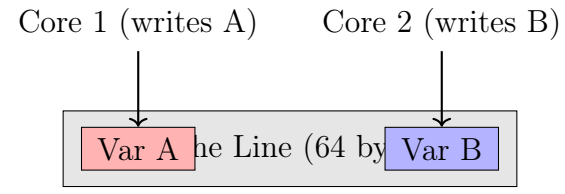


Рис. 10.1 – Иллюстрация False Sharing

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
};
```

10.9 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.

10.10 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.

Глава 11

11 Лекция

Оглавление

11.1 Введение

В рамках данной лекции рассматривается стек сетевых технологий: от физических принципов передачи сигнала до реализации высоконагруженных серверов с использованием механизмов мультиплексирования ввода-вывода. Основное внимание уделяется архитектурным ограничениям (trade-offs) различных протоколов и системных вызовов.

11.2 Физический и Канальный уровни

11.2.1 Принципы передачи сигнала

Фундаментальной задачей сетевого взаимодействия является передача информации между двумя физически соединёнными узлами. На физическом уровне это реализуется посредством модуляции напряжения в проводнике. Выделяют два типа сигналов:

1. **Аналоговый сигнал.** Обладает непрерывным спектром значений. Позволяет передавать больший объём информации, однако критически подвержен зашумлению, что делает его непригодным для точной передачи данных в современных компьютерных сетях.
2. **Цифровой сигнал.** Использует дискретный набор значений напряжения (например, высокий и низкий уровень). Значения, выходящие за пределы порогов, округляются, что обеспечивает устойчивость к помехам.

При передаче последовательности бит (например, длинной серии единиц) возникает проблема синхронизации: получатель (Receiver) может рассинхронизироваться с отправителем (Sender) относительно частоты тактов. Для решения данной проблемы применяется **Манчестерское кодирование**.

Определение: Манчестерское кодирование

Метод кодирования, при котором данные логически объединяются (операция XOR) с тактовым сигналом (Clock). Это гарантирует наличие перепада напряжения (фронта) в середине каждого битового интервала, что позволяет получателю синхронизировать частоту по самому сигналу.

11.2.2 Протокол Ethernet и MAC-адресация

Для структурирования потока бит используется протокол Ethernet, оперирующий единицами данных, называемыми фреймами (frames).

Адресация на канальном уровне осуществляется посредством MAC-адрес-адресов. Предполагается глобальная уникальность MAC-адресов, обеспечиваемая распределением диапазонов (OUI) между производителями оборудования (например, Intel, Cisco).

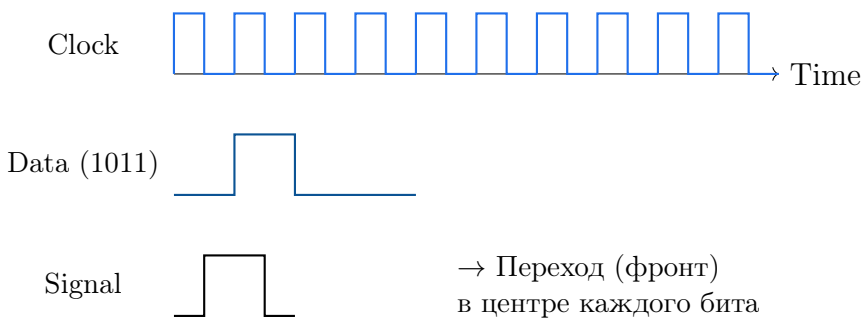


Рис. 11.1 – Принцип Манчестерского кодирования (схематично)

Таблица 11.1 – Структура Ethernet-фрейма

Поле	Описание	Размер
Preamble	Синхронизация	8 байт
Destination MAC	MAC-адрес получателя	6 байт
Source MAC	MAC-адрес отправителя	6 байт
EtherType	Тип инкапсулированного протокола (IPv4/IPv6)	2 байта
Payload	Полезная нагрузка (например, IP-пакет)	46-1500 байт
FCS	Контрольная сумма (CRC) для проверки целостности	4 байта

11.3 Сетевой уровень (Network Layer)

11.3.1 Протокол IP и маршрутизация

Протокол IP (Internet Protocol) обеспечивает глобальную адресацию и маршрутизацию пакетов между сетями. Наиболее распространённая версия IPv4 использует 32-битные адреса, записываемые в виде четырёх октетов (например, `192.168.0.1`).

Ввиду ограниченности адресного пространства IPv4 (всего $2^{32} \approx 4.3$ млрд адресов), широко применяется технология NAT.

Определение: NAT (Network Address Translation)

Механизм, позволяющий устройствам из локальной сети (с приватными адресами) выходить в глобальную сеть, используя один публичный IP-адрес маршрутизатора. Маршрутизатор подменяет адрес источника и порт, сохраняя состояние соединения в таблице трансляции.

11.3.2 Подсети и маски (CIDR)

Для логического разделения сетей используется бесклассовая адресация (CIDR). Адрес сети определяется префиксом, а маска подсети указывает количество бит, отведённых под адрес сети.

```
IP: 88.99.146.0
CIDR: /23 (23 bits network, 9 bits host)

Netmask: 11111111.11111111.11111110.00000000 (255.255.254.0)
Wildcard: 00000000.00000000.00000001.11111111 (0.0.1.255)

Network: 88.99.146.0
HostMin: 88.99.146.1
```

```
HostMax: 88.99.147.254
Broadcast: 88.99.147.255
Hosts/Net: 510 (2^9 - 2)
```

Листинг 11.1 – Пример расчета диапазона подсети /23

Примечание

Адреса, у которых хостовая часть состоит из всех нулей (адрес сети) или всех единиц (Broadcast), зарезервированы и не могут быть назначены конкретному интерфейсу.

Для предотвращения бесконечной циркуляции пакетов при ошибках маршрутизации используется поле **TTL** (Time To Live), значение которого уменьшается на единицу при прохождении каждого промежуточного узла (хопа).

11.4 Транспортный уровень (Transport Layer)

Сетевой уровень доставляет данные до хоста. Транспортный уровень (L4) отвечает за мультиплексирование потоков данных для конкретных приложений, используя абстракцию **порта** (16-битное число).

11.4.1 Сравнение TCP и UDP

Таблица 11.2 – Сравнение транспортных протоколов

Характеристика	TCP (Stream)	UDP (Datagram)
Гарантии	Гарантирует доставку и порядок байт.	Не гарантирует доставку и порядок.
Соединение	Требует установки соединения (handshake).	Отправка данных без установки соединения ("fire and forget").
Поток данных	Непрерывный поток байт.	Дискретные сообщения (датаграммы).
Overhead	Высокий (заголовки, подтверждения).	Низкий (минимальный заголовок).
Применение	Web (HTTP), Email, File transfer.	DNS, Streaming, VoIP, Gaming.

11.4.2 Порядок байт (Endianness)

Сетевые протоколы исторически используют порядок байт **Big-Endian** (старший байт по младшему адресу). Архитектура x86 использует **Little-Endian**. Для корректной интерпретации многобайтовых чисел (например, порта или IP-адреса) необходима конвертация:

```
#include <arpa/inet.h>

uint16_t host_port = 8080;
// Host TO Network Short (conversion for sending)
uint16_t net_port = htons(host_port);
// Network TO Host Short (conversion on receipt)
uint16_t back_port = ntohs(net_port);
```

Листинг 11.2 – Конвертация порядка байт

11.5 Прикладной уровень (Application Layer)

Прикладные протоколы (L7) определяют семантику передаваемых данных.

11.5.1 HTTP и DNS

Протокол HTTP — текстовый протокол, работающий поверх TCP. Сообщения состоят из стартовой строки, заголовков и опционального тела. В качестве разделителя строк используется последовательность `CRLF` (`\r\n`).

Система DNS выполняет трансляцию доменных имён (например, `google.com`) в IP-адреса. Для получения адреса в языке C используется функция `getaddrinfo`, возвращающая список структур `addrinfo`.

Примечание

Один домен может разрешаться (resolve) в несколько IP-адресов для балансировки нагрузки на уровне DNS (DNS Round Robin) и обеспечения географической близости к клиенту (CDN).

11.6 Socket API и Модели конкурентности

11.6.1 Базовый жизненный цикл TCP-сервера

Интерфейс сокетов (сокет) в POSIX-системах реализует файловую абстракцию для сетевого взаимодействия.

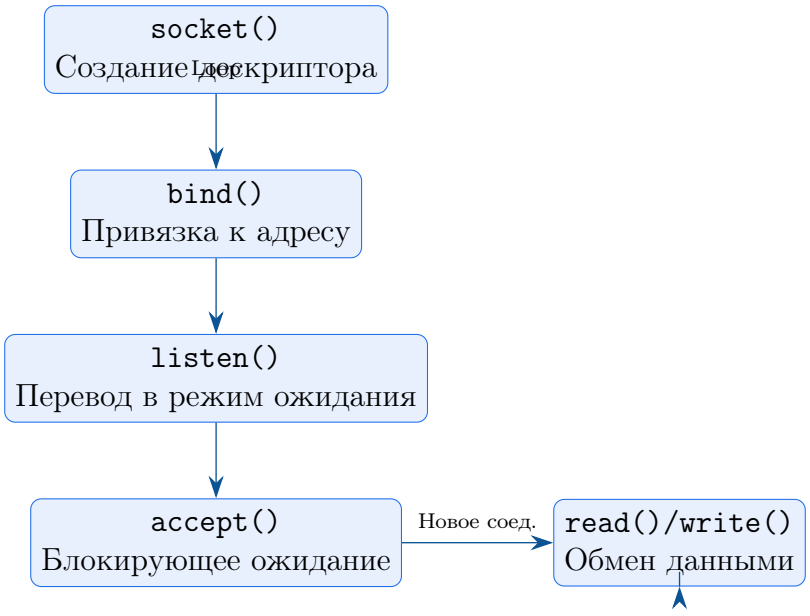


Рис. 11.2 – Системные вызовы TCP-сервера

Критическая проблема базовой модели: вызов `read()` является блокирующим. Если сервер обслуживает одного клиента, остальные клиенты ожидают в очереди (backlog), создаваемой ядром ОС.

11.6.2 Эволюция моделей ввода-вывода

Thread-per-connection (Поток на соединение)

Создание отдельного потока ОС (pthread) для каждого клиента.

- **Преимущество:** Простота реализации, линейный код.

- **Цена (Trade-off):** Высокие накладные расходы. Каждый поток требует аллокации стека (по умолчанию 2-8 МБ) и структур ядра. Переключение контекста (Context Switch) при тысячах потоков приводит к существенной деградации производительности (CPU тратится на планировщик, а не на полезную работу).

Non-blocking I/O (Неблокирующий ввод-вывод)

Файловый дескриптор переводится в неблокирующий режим (`O_NONBLOCK`). Системные вызовы `read`/`write` возвращают ошибку `EAGAIN`, если данные не готовы. Это позволяет одному потоку опрашивать множество сокетов, но приводит к проблеме **Busy Wait** (100% загрузка CPU при пустых циклах опроса).

I/O Multiplexing (epoll)

Механизм epoll (Linux) позволяет потоку «заснуть» до возникновения события на одном из множества наблюдаемых дескрипторов.

Определение: Event Loop

Архитектурный паттерн, в котором единый поток циклически ожидает событий от мультиплексора (epoll) и вызывает соответствующие обработчики (callbacks). Это позволяет обрабатывать десятки тысяч соединений (проблема C10k) с минимальными накладными расходами памяти и отсутствием лишних переключений контекста.

```
int epfd = epoll_create1(0);
struct epoll_event ev, events[MAX_EVENTS];

// Register server socket
ev.events = EPOLLIN;
ev.data.fd = server_socket;
epoll_ctl(epfd, EPOLL_CTL_ADD, server_socket, &ev);

while (1) {
    // Wait for events (blocking)
    int nfds = epoll_wait(epfd, events, MAX_EVENTS, -1);

    for (int i = 0; i < nfds; ++i) {
        if (events[i].data.fd == server_socket) {
            // handle new connection (accept)
        } else {
            // handle data (read/write)
        }
    }
}
```

Листинг 11.3 – Скелет Event Loop на epoll

11.7 Итоги раздела

Итоги раздела

- Сетевое взаимодействие представляет собой иерархию абстракций: от модуляции напряжения до прикладных протоколов (OSI Model).

- Протоколы TCP и UDP предоставляют различные гарантии (надёжность против скорости), что диктует их области применения.
- Прямое использование потоков (Thread-per-connection) неэффективно для I/O-bound задач с большим числом соединений из-за накладных расходов на память и планирование.
- Современные высоконагруженные серверы используют неблокирующий ввод-вывод и механизм epoll для мультиплексирования событий в едином цикле (Event Loop).

Глава 12

13 Лекция

12.1 Асинхронная модель и Системные события: Signals, Timers, inotify

12.1.1 Введение в асинхронность и доставку сигналов

Работа с сигналами в UNIX-подобных системах представляет собой один из старейших механизмов межпроцессного взаимодействия (IPC) и обработки исключительных ситуаций. Ключевая особенность сигналов — их асинхронная природа.

Определение: Сигнал

Программное прерывание, доставляемое процессу операционной системой. Обработчик сигнала (signal handler) может быть вызван в произвольный момент времени, прерывая нормальное исполнение инструкций пользовательского кода (user-space).

В момент доставки сигнала ядро приостанавливает выполнение основного потока инструкций, сохраняет контекст процессора, модифицирует стек пользователя (или переключается на альтернативный стек сигналов) и передает управление функции-обработчику. Это накладывает строгие ограничения на код обработчика.

Из-за асинхронности вызова, компилятор не может предсказать момент изменения переменных внутри обработчика. Если переменная используется в основном цикле и модифицируется в обработчике, она должна быть объявлена как `volatile sig_atomic_t`. Без спецификатора `volatile` оптимизатор может закэшировать значение в регистре и никогда не увидеть изменений («бесконечный цикл ожидания»).

Процесс Init и иммунитет к сигналам

В иерархии процессов Linux особую роль играет процесс с PID 1, известный как `init`. Он является корнем дерева процессов и имеет специфическую защиту на уровне ядра.

Процессу `init` нельзя отправить сигнал, на который у него явно не установлен обработчик. Это «костыль» ядра (kernel workaround), предназначенный для предотвращения случайного завершения системы. Из этого следует важный вывод: процессу `init` невозможно отправить сигналы `SIGKILL` (9) и `SIGSTOP`, так как эти сигналы технически невозможно перехватить (установить на них обработчик). Следовательно, ядро просто игнорирует их доставку для PID 1, если только сам `init` не был написан с учетом их обработки (что невозможно для данных сигналов).

12.1.2 Классификация сигналов: Синхронные и Асинхронные

Источники сигналов можно разделить на две фундаментальные категории в зависимости от их происхождения относительно исполняемого потока.

Асинхронные сигналы

Генерируются внешними событиями или другими процессами. Примером служит системный вызов `kill`, который отправляет сигнал от одного процесса другому (с проверкой прав доступа). Пользовательское действие `Ctrl+Z` в терминале отправляет `SIGSTOP`, который переводит процесс в состояние «stopped» (исключает из планировщика ОС), а `SIGCONT` возобновляет его исполнение.

Синхронные сигналы (Traps)

Являются прямым следствием выполнения конкретной инструкции процессора.

- **SIGSEGV (Segmentation Fault):** Доступ к невалидной области памяти (отсутствует маппинг в таблице страниц или нарушение прав доступа).

- **SIGBUS:** Ошибка шины, часто возникающая при невыровненном доступе к памяти на архитектурах, не поддерживающих его, или при доступе к файлу через `mmap`, если файл был усечен.
- **SIGFPE:** Ошибка арифметической операции (деление на ноль, переполнение).

Самый простой способ вызвать синхронный сигнал программно — функция `raise()`, которая, по сути, реализует `kill(getpid(), sig)`. Стандарт POSIX гарантирует, что сигнал от `raise` будет доставлен синхронно: обработчик вызовется до возврата из функции.

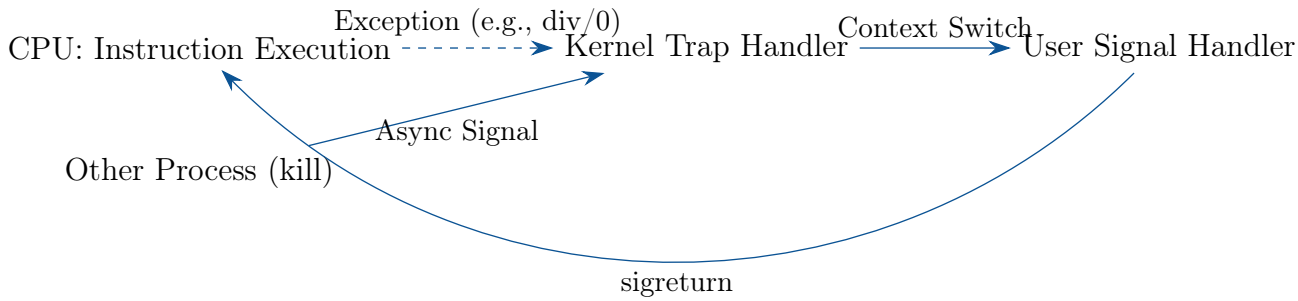


Рис. 12.1 – Пути доставки синхронных и асинхронных сигналов

12.1.3 Обработка синхронных ошибок памяти

Обычно обработка синхронных сигналов вроде `SIGSEGV` бессмысленна — нельзя просто вернуться к инструкции, вызвавшей сбой (она снова вызовет сбой). Однако существуют архитектурные паттерны, использующие это поведение.

Lazy Process Migration

Один из продвинутых сценариев — «ленивая» миграция процессов между машинами.

1. На целевой машине создается процесс-пустышка.
2. При попытке исполнения кода или доступа к данным происходит `SIGSEGV` (память не выделена).
3. Установленный обработчик `SIGSEGV` перехватывает управление.
4. Обработчик определяет адрес сбоя, подгружает нужную страницу памяти по сети с исходной машины, выполняет `mmap`.
5. Обработчик завершается, и инструкция перезапускается — теперь уже успешно.

Блокировка синхронных сигналов

Возникает вопрос: что произойдет, если заблокировать `SIGSEGV` с помощью `sigprocmask`, а затем вызвать ошибку сегментации?

Примечание

Операционная система не может отложить доставку синхронного сигнала (в отличие от асинхронного), так как продолжение исполнения невозможно. Ядро Linux в такой ситуации принудительно завершает процесс («убивает»), даже если сигнал заблокирован. Это компромисс реализации: программа считается некорректной.

Если же попытаться разблокировать сигнал внутри его собственного обработчика и вызвать ошибку снова, возникнет бесконечная рекурсия. Каждый новый вызов обработчика создает новый стековый фрейм, что неизбежно приведет к исчерпанию стека (Stack Overflow) и аварийному завершению.

12.1.4 Эволюция API: signalfd и Event Loop

Асинхронные обработчики сигналов сложны в отладке и ограничены в возможностях (можно использовать только *async-signal-safe* функции). Современный подход в Linux — интеграция сигналов в общий цикл событий (Event Loop) через файловые дескрипторы.

Механизм `signalfd` позволяет принимать сигналы синхронно, вычитывая их как данные из специального Файловый дескриптор.

```c
#include <sys/signalfd.h>
#include <signal.h>
#include <unistd.h>

int main() {
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);

    // 1. Block signals so they are not delivered via standard async handlers
    if (sigprocmask(SIG_BLOCK, &mask, NULL) == -1)
        handle_error("sigprocmask");

    // 2. Create a file descriptor for reading signals
    int sfd = signalfd(-1, &mask, 0);
    if (sfd == -1)
        handle_error("signalfd");

    // The sfd can now be added to epoll or read via blocking read()
    struct signalfd_siginfo fdsi;
    ssize_t s = read(sfd, &fdsi, sizeof(struct signalfd_siginfo));
    // ...
}
```

Листинг 12.1 – Использование signalfd с блокировкой

При чтении из `signalfd` возвращается структура `signalfd_siginfo`. В ней содержится детальная информация о сигнале: PID отправителя, `uid`, код завершения (для `SIGCHLD`) и т.д.

Любопытная деталь реализации структуры — наличие в конце поля-заполнителя (padding):

```c
uint8_t __pad[28]; // Pad to 128 bytes
```

Размер 28 байт (вместе с остальными полями доводит размер структуры до 128 байт) оставлен для прямой совместимости (forward compatibility). Если в будущем ядру потребуется передавать дополнительные данные с сигналом, они займут место этого паддинга, и старые программы (скомпилированные с текущим размером структуры) продолжат корректно работать, просто не читая новые поля.

12.1.5 Управление процессами: pidfd

Традиционный системный вызов `waitpid` имеет архитектурный недостаток, связанный с переиспользованием PID. Поскольку идентификаторы процессов — это ограниченный ресурс (обычно 16-битное число), после завершения процесса и вызова `wait` его PID освобождается. Операционная система может сразу же назначить этот PID новому процессу.

Если родительский процесс "промедлит"с ожиданием, он может случайно вызвать `waitpid` или отправить сигнал (`kill`) уже новому, ни в чем не повинному процессу, который занял тот же номер. Это состояние гонки (Race Condition).

Для решения этой проблемы в Linux 5.3+ введен механизм `pidfd`.

Определение: pidfd

Файловый дескриптор, ссылающийся на конкретный экземпляр процесса, а не на его числовой идентификатор. Даже если процесс завершится и его PID будет переиспользован, `pidfd` будет гарантированно указывать на "старый"(уже мертвый) процесс, предотвращая ошибочную отправку сигналов.

`pidfd` также интегрируется с `epoll`: дескриптор становится доступным для чтения (readable), когда процесс завершается. Это позволяет унифицировать ожидание сетевых событий, сигналов и завершения дочерних процессов в одном цикле.

12.1.6 Мониторинг файловой системы: inotify

Для отслеживания изменений в файловой системе (создание, удаление, запись) вместо неэффективного поллинга (периодического вызова `stat`) используется механизм `inotify`.

При работе с `inotify` возникает сложность: события имеют переменный размер. Структура `inotify_event` содержит поле имени файла, длина которого заранее неизвестна.

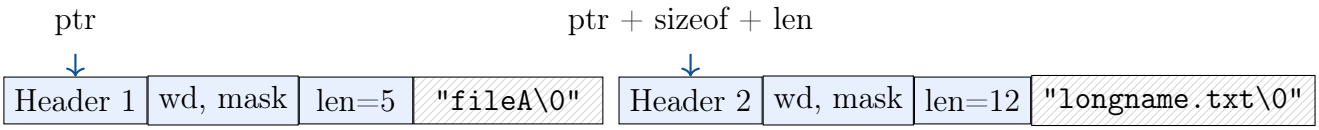


Рис. 12.2 – Буфер чтения inotify с записями переменной длины

Поле `name` в конце структуры является массивом переменной длины (Variable Length Array concept). При чтении из дескриптора `inotify` программа получает буфер с последовательностью таких структур. Итерирование по ним требует ручного смещения указателя:

$$\text{next\_ptr} = \text{current\_ptr} + \text{sizeof(struct inotify\_event)} + \text{event->len}$$

Еще один нюанс `inotify` — атомарность переименования. Операция `mv A B` генерирует два события: `MOVED_FROM` и `MOVED_TO`. Чтобы связать их между собой (понять, что файл не просто исчез и появился, а был переименован), ядро заполняет поле `cookie` одинаковым уникальным значением для обоих событий.

Итоги раздела

- Сигналы обрабатываются асинхронно, что требует защиты общих данных (`volatile sig_atomic_t`).
- Синхронные сигналы (Traps) нельзя отложить; их блокировка может привести к аварийному завершению ядра.
- Современные Linux API (`signalfd`, `pidfd`, `inotify`) уходят от концепции callbacks/interrupts к концепции дескрипторов, интегрируемых в единый Event Loop (`epoll`).
- `pidfd` решает критическую проблему безопасности (гонки PID) при управлении процессами.

12.2 Семантика памяти в C++: Pointer Provenance и Абстрактная машина

В низкоуровневом системном программировании существует фундаментальный разрыв между тем, как память видит процессор (CPU), и тем, как её представляет компилятор языка C/C++. Для процессора память — это плоский линейный массив байтов, а адрес — простое целое число, с которым можно производить любые арифметические операции. Однако для оптимизирующего компилятора C++ (опирающегося на стандарт Abstract Machine) указатель — это гораздо более сложная сущность.

Игнорирование этой разницы приводит к тому, что код, выглядящий абсолютно корректным с точки зрения ассемблера, становится некорректным (Undefined Behavior) с точки зрения стандарта языка, позволяя компилятору удалять проверки безопасности или генерировать нерабочий код.

12.2.1 Концепция Pointer Provenance

Ключевым понятием, объясняющим поведение современных компиляторов при работе с памятью, является *происхождение указателя* (Provenance).

Определение: Pointer Provenance (Происхождение указателя)

Абстрактное свойство указателя, связывающее его с конкретным объектом аллокации (allocation site). Указатель в семантике C++ можно представить не как число `uint64_t address`, а как кортеж:

$$\text{ptr} = (\text{Allocation\_ID}, \text{Offset})$$

Любая арифметика над указателем изменяет только `Offset`, но не `Allocation_ID`. Доступ к памяти валиден тогда и только тогда, когда адрес физически попадает в диапазон аллокации **и** `Allocation_ID` совпадает с ID объекта по этому адресу.

Рассмотрим классический пример, демонстрирующий эту концепцию. Пусть у нас есть два массива, расположенных в памяти друг за другом (что часто случается на стеке).

```
int* f() {
    int a[3] = {1, 1, 1};
    int b[3] = {2, 2, 2};

    // Assume stack grows downwards and 'b' is placed right after 'a'
```

```
    // Address-wise: &a[3] == &b[0]
    int* p = &a[0];
    p += 3; // Formally this is &a[3], a past-the-end iterator

    // Attempting dereference
    // Asm: reads b[0] (value 2)
    // C++: Undefined Behavior
    return *p;
}
```

Листинг 12.2 – Выход за границы массива с точки зрения ассемблера и C++

Компилятор, анализируя этот код, рассуждает следующим образом: указатель `p` происходит от объекта `a` (Provenance: `a`). Арифметика `p += 3` создает указатель со смещением 3, но с тем же происхождением. Поскольку доступ `*p` выходит за границы объекта `a`, компилятор вправе считать этот код «мертвым» или невозможным (unreachable), даже если физически по этому адресу расположены данные массива `b`.

Примечание

Стандарт разрешает вычислять указатель на элемент, следующий за последним (*past-the-end pointer*), например `&arr[size]`, для использования в итераторах и сравнениях. Однако **разыменовывать** такой указатель запрещено, даже если за массивом есть валидная память.

12.2.2 Оптимизация аллокаций: Dead Allocation Elimination

Понимание provenance позволяет объяснить агрессивные оптимизации. Рассмотрим функцию, которая выделяет память, но использует её специфическим образом.

```
int example() {
    int* p = new int(42); // Allocation A
    int* q = new int(42); // Allocation B

    // Comparison of pointers from different allocations
    bool equal = (p == q);

    delete p;
    delete q;

    return equal; // Always false
}
```

Листинг 12.3 – Удаление "неиспользуемой"аллокации

В этом примере компилятор видит, что `p` и `q` имеют разный provenance (разные вызовы `new`). Стандарт гласит, что указатели на разные живые объекты не могут быть равны. Следовательно, выражение `p == q` всегда ложно.

Далее вступает в силу *Dead Allocation Elimination*: раз содержимое памяти по адресам `p` и `q` никак не влияет на наблюдаемое поведение программы (кроме самого факта их существования, который мы только что оптимизировали до `false`), вызовы `new` и `delete` можно полностью удалить.

В результирующем ассемблерном коде не будет вызовов аллокатора `malloc/new`, функция просто вернет 0. Это контринтуитивно, так как `new` имеет побочный эффект

(выделение памяти), но компилятор имеет право его удалить, если этот эффект не наблюдаем в рамках абстрактной машины.

12.2.3 Проблема Roundtrip Casting и XOR Linked List

Преобразование указателя в целое число (`reinterpret_cast<uintptr_t>`) и обратно — это операция, поддерживаемая компиляторами, но с нюансами.

Стандарт гарантирует, что `ptr -> int -> ptr` вернет исходный указатель с исходным provenance. Однако, если над числом была произведена арифметика, provenance теряется.

$$\text{ptr} \xrightarrow{\text{cast}} \text{int} \xrightarrow{\text{math}} \text{new\_int} \xrightarrow{\text{cast}} \text{new\_ptr (Provenance: ???)}$$

Это ставит под угрозу такие классические структуры данных, как **XOR Linked List**.

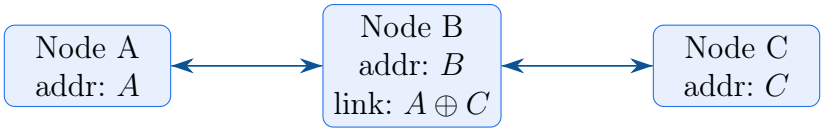


Рис. 12.3 – Концепция XOR Linked List

В XOR-списке (рис. 12.3) вместо хранения двух указателей `prev` и `next`, хранится их побитовое исключающее ИЛИ: `link = prev ^ next`. Чтобы перейти к следующему элементу, зная предыдущий, выполняется операция:

$$\text{next} = \text{link} \oplus \text{prev}$$

С точки зрения C++, мы берем целое число, выполняем над ним операцию XOR и кастуем результат в указатель. Полученный указатель не имеет provenance (он «создан из воздуха» с точки зрения компилятора). Доступ по такому указателю формально является Undefined Behavior, хотя на большинстве платформ это работает. Тем не менее, теоретически компилятор может сломать этот код, решив, что раз указатель не получен от аллокатора, то он не может указывать ни на один валидный объект.

12.2.4 Конфликт оптимизаций LLVM: Case Study (2018)

В 2018 году исследователи обнаружили, что в компиляторе LLVM (используется в Clang, Rust, Swift) комбинация трех корректных по отдельности оптимизаций приводила к некорректной генерации кода.

Рассмотрим следующий код (упрощенная модель):

```
void bug_demo(int *p, int *q) {
    // p and q point to different objects (different provenance)
    // But physically they might be equal after arithmetic logic

    uintptr_t ip = (uintptr_t)p;
    uintptr_t iq = (uintptr_t)q;

    if (ip == iq) {
        // If addresses match numerically, write to p
        *p = 10;
    }
}
```

Листинг 12.4 – Код, ломающий оптимизатор LLVM

Цепочка преобразований, которую выполнял компилятор:

1. **Gvn (Global Value Numbering) / Constant Propagation:** Компилятор видит условие `if (ip == iq)`. Внутри ветки `then` он знает, что `ip` равно `iq`. Следовательно, он может заменить использование `p` на `q` (или наоборот), так как их числовые представления равны. *Результат:* замена `*p = 10` на `*q = 10` (или создание эквивалентного указателя из `iq`).
2. **IntToPtr Cast Folding:** Если мы привели `q` к `int`, а потом обратно, это эквивалентно исходному `q`.
3. **Dead Store Elimination (на основе Provenance):** Это критический шаг. Компилятор анализирует запись `*q = 10`. Он знает, что в этой функции (по условиям вызова) мы работаем с объектом `p`. Указатель `q` имеет другой provenance. Стандарт говорит, что доступ к объекту `p` через указатель с provenance `q` невозможен (они не алиасятся). *Вывод компилятора:* запись `*q = 10` недостижима или не влияет на `p`. Инструкция удаления записи удаляется.

Итог: В исходном коде, если адреса совпадали, запись должна была произойти. В скомпилированном коде запись исчезла. Корректная программа сломалась. Это привело к пересмотру модели памяти в LLVM и ограничению оптимизаций при работе с `inttoptr` кастами.

12.2.5 Практический пример: Hazard Pointers и Placement New

В высокопроизводительных базах данных (например, YDB) часто используются кастомные аллокаторы. Рассмотрим структуру, где заголовок и данные аллоцируются одним куском памяти:

```
struct Chunk {
    int header;
    // Data starts right here, after the header
};

void* mem = malloc(sizeof(Chunk) + sizeof(Data));
Chunk* chunk = new (mem) Chunk(); // Placement new

// Attempting to access data via pointer arithmetic from the header
char* data_ptr = reinterpret_cast<char*>(chunk) + sizeof(Chunk);
Data* data = reinterpret_cast<Data*>(data_ptr);

// Constructing data in place
new (data) Data();
```

Листинг 12.5 – Опасная арифметика с placement new

Проблема здесь заключается в том, как компилятор видит объект `chunk`. Он был создан как объект типа `Chunk`. Получение указателя на `Data`, который лежит за пределами `sizeof(Chunk)`, формально является выходом за границы объекта `Chunk`. Хотя физически память выделена одним блоком `malloc`, с точки зрения C++ `chunk` — это указатель на объект конкретного размера. Попытка "шагнуть"за его пределы и обратиться к памяти может быть расценена как UB, если компилятор не увидит связь с исходным `malloc`.

Итоги раздела

- Указатель в C++ — это **адрес** + **provenance** (информация о происхождении).
- Арифметическое равенство адресов (`0x1000 == 0x1000`) не гарантирует возможность взаимозаменяемости указателей, если они имеют разный provenance.
- Компилятор имеет право удалять «мертвые» аллокации (`new` без использования), даже если это меняет наблюдаемые побочные эффекты.
- Преобразование указателей в целые числа и обратно стирает provenance, что может мешать оптимизатору отслеживать зависимости (alias analysis) и приводить к удалению «лишних» записей в память.
- Для написания корректных аллокаторов и низкоуровневых структур данных необходимо использовать `std::launder` (C++17) или специальные атрибуты компилятора, чтобы «отмыть» указатель и начать новый цикл жизни объекта.

12.3 Каталог Undefined Behavior и Агрессивные Оптимизации

В основе философии языков C и C++ лежит принцип «Trust the Programmer» (Доверяй программисту). Компилятор исходит из предположения, что программист никогда не пишет некорректный код, не допускает переполнений и не выходит за границы массивов. Это позволяет отказаться от дорогостоящих проверок в рантайме (bounds checking, overflow checking) и применять агрессивные оптимизации.

Однако обратной стороной этой медали является *неопределенное поведение* (Undefined Behavior, UB). Если программа нарушает правила абстрактной машины, стандарт перестает гарантировать что-либо, и компилятор получает право трансформировать код любым образом, полагая, что данная ситуация недостижима.

12.3.1 Нарушение потока управления (Control Flow UB)

Одним из самых опасных видов UB является нарушение контрактов функций, возвращающих значение.

Отсутствие return в не-void функции

Согласно стандарту, если поток исполнения достигает конца функции, возвращающей значение (не `void`), и не встречает инструкции `return`, возникает Undefined Behavior.

```
int process_data(bool cond) {
    if (cond) {
        return 42;
    }
    // Missing return for the case cond == false
}

int main() {
    process_data(false);
}
```

Листинг 12.6 – Отсутствие return и генерация кода

При компиляции с оптимизациями (например, `-O2`) компилятор может рассуждать так:

1. Вызов `process_data(false)` приведет к исполнению пути без `return`.

2. Этот путь является UB.
3. Программа, содержащая UB, некорректна.
4. Следовательно, случай `cond == false` невозможен.
5. Можно удалить проверку `if (cond)` и считать, что `cond` всегда истинно, или просто сгенерировать пустую функцию.

На уровне ассемблера это часто приводит к тому, что функция не содержит инструкции возврата (`ret`) или восстановления стека для "невозможной"ветки. Процессор продолжает исполнение инструкций, следующих сразу за телом функции в памяти (fallthrough). Это может быть код следующей функции, данные (интерпретируемые как инструкции) или невыровненный мусор. Часто это заканчивается сигналом `SIGILL` (Illegal Instruction) или `SIGSEGV`.

Примечание

Единственным исключением является функция `main`. Стандарт C++ разрешает не писать `return 0;` в конце `main` — в этом случае компилятор подставляет его автоматически. Для всех остальных функций это строгое UB.

Бесконечные циклы без побочных эффектов

Еще один контринтуитивный аспект оптимизации связан с завершаемостью программ. Стандарт C++ (до C++11 и в определенных контекстах после) гласит, что бесконечный цикл без побочных эффектов (side effects) является UB. Под побочными эффектами понимаются операции ввода-вывода, доступ к `volatile` переменным или атомикам.

Если компилятор видит цикл, который просто вычисляет что-то в регистрах и никогда не завершается, он имеет право удалить этот цикл целиком, считая, что программа не должна застревать навечно.

```
bool check_fermat() {
    // Brute-force a, b, c to infinity or overflow
    for (int a = 1; ; ++a) {
        for (int b = 1; b <= a; ++b) {
            for (int c = 1; c <= a + b; ++c) {
                if (a*a*a + b*b*b == c*c*c) {
                    return true; // Counter-example found!
                }
            }
        }
    }
    // Execution never reaches here
    return false;
}
```

Листинг 12.7 – «Доказательство» Великой теоремы Ферма через UB

Компилятор анализирует этот код:

1. В цикле нет побочных эффектов (IO, volatile).
2. Если цикл бесконечен, это UB. Значит, цикл **должен** завершиться.
3. Единственный штатный выход из цикла — `return true`.

4. Следовательно, функция всегда возвращает `true`.

Оптимизатор заменяет все тело функции на инструкцию `mov eax, 1; ret`, тем самым "опровергая"теорему Ферма.

Примечание

В языке Rust конструкция `loop {}` является легитимным способом остановить программу или организовать вечный цикл. Однако, так как Rust использует бэкенд LLVM (общий с Clang), в прошлом возникали баги, когда LLVM удалял такие циклы, считая их UB по правилам C++. Это приводило к крашам в абсолютно безопасном (memory safe) коде на Rust.

12.3.2 Арифметика и Типы данных

Переполнение знаковых целых (Signed Integer Overflow)

В C++ переполнение знаковых типов (`int`, `long`) является UB. Переполнение беззнаковых (`unsigned`) определено как арифметика по модулю 2^N.

Компилятор полагается на то, что переполнения не происходит. Это позволяет делать следующие упрощения:

- `if (x + 1 > x)` → всегда `true`.
- `for (int i = 0; i < N; ++i)` → можно использовать 64-битный счетчик или векторизовать цикл, не беспокоясь о том, что `i` станет отрицательным после `INT_MAX`.

Это может приводить к неожиданным бесконечным циклам:

```
void check(int n) {
    // If n = INT_MAX, theoretically loop should be infinite upon overflow
    for (int i = 0; i < n + 100; ++i) {
        printf("%d\n", i);
    }
}
```

Листинг 12.8 – Оптимизация цикла с переполнением

Компилятор может удалить проверку `i < n + 100`, если решит, что `n + 100` не может переполниться, или наоборот, превратить цикл в бесконечный, игнорируя условие остановки.

Неинициализированные переменные

Чтение неинициализированной памяти — это не просто получение случайного "мусора". Это получение значения, которое может вести себя нестабильно (quantum state). Переменная `bool b`, которая не была инициализирована, может одновременно оцениваться как `true` в одной ветке оптимизации и как `false` в другой, приводя к выполнению взаимоисключающих блоков кода.

12.3.3 Strict Aliasing и Type-Based Alias Analysis (TBAA)

Одним из важнейших механизмов оптимизации работы с памятью является анализ алиасинга (Alias Analysis). Компилятору необходимо знать, могут ли два указателя адресовать одну и ту же ячейку памяти.

Определение: Strict Aliasing Rule

Правило строгого алиасинга гласит, что доступ к объекту в памяти может осуществляться только через указатель (или ссылку) совместимого типа.

- Нельзя читать `float` через `int*`.
- Нельзя читать `struct A` через `struct B*`.
- **Исключение:** Тип `char*` (и `unsigned char*`, `std::byte*`) может алиасить любые данные. Это необходимо для реализации `memcpy` и побайтового копирования.

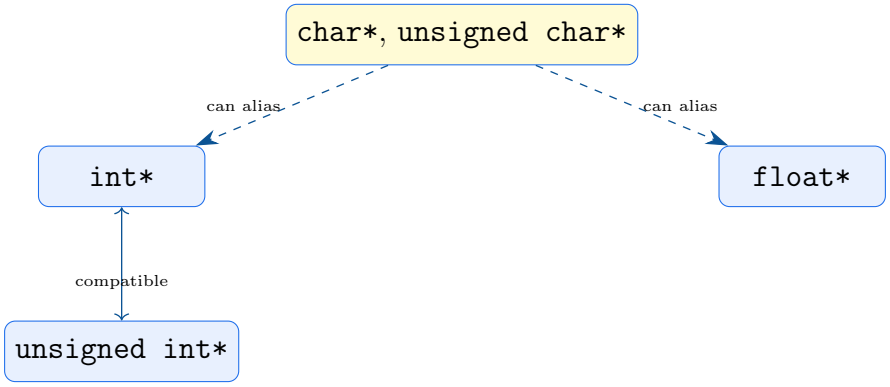


Рис. 12.4 – Иерархия совместимости указателей (упрощенная)

Нарушение этого правила позволяет компилятору предполагать, что записи в память через указатели разных типов **не влияют** друг на друга. Это критически важно для векторизации и регистрового кэширования.

Пример: std::vector<int> против std::vector<size_t>

Рассмотрим реализацию заполнения вектора значениями.

```
template <typename T>
void fill_vector(std::vector<T>& v, const T& val) {
    for (size_t i = 0; i < v.size(); ++i) {
        v[i] = val;
    }
}
```

Листинг 12.9 – Влияние типов на оптимизацию цикла

Компилятор генерирует принципиально разный код для `T=int` и `T=size_t`.

Случай 1: `std::vector<int>` Тип элементов — `int`, тип размера вектора — `size_t` (обычно `unsigned long`). Правила Strict Aliasing гарантируют, что запись в `v[i]` (типа `int`) не может изменить поле `v.size()` (типа `size_t`) внутри структуры вектора. *Оптимизация:* Компилятор загружает `v.size()` в регистр один раз перед циклом.

Случай 2: `std::vector<size_t>` Тип элементов — `size_t`, тип размера — `size_t`. Указатель на данные (`v.data()`) имеет тип `size_t*`. Компилятор не может доказать, что массив данных вектора не перекрывается с памятью, где лежит само поле `size` структуры вектора (теоретически, вы могли создать вектор поверх его же заголовка через placement new и злые касты). *Результат:* Компилятор вынужден перезагружать значение `v.size()` из памяти на **каждой** итерации цикла, так как запись `v[i] = val` потенциально могла изменить размер. Это блокирует авто-векторизацию и снижает производительность.

Для решения таких проблем в языке C (и как расширение в C++) существует ключевое слово `restrict`, которое обещает компилятору, что указатель не алиасится ни с чем другим. В стандартном C++ часто приходится копировать размер в локальную переменную перед циклом, чтобы "подсказать"оптимизатору:

```
size_t n = v.size(); // Local copy to avoid reloading
for (size_t i = 0; i < n; ++i) v[i] = val;
```

Итоги раздела

- Undefined Behavior — это не ошибка, а контракт: компилятор обещает быстрый код, если программист обещает соблюдать правила.
- Отсутствие `return` в функции (кроме main) ломает поток управления и может привести к выполнению произвольного кода.
- Бесконечные циклы без побочных эффектов могут быть полностью удалены.
- Strict Aliasing Rule позволяет компилятору эффективно работать с памятью, но требует строгого соблюдения типов. Использование `reinterpret_cast` между несовместимыми типами (например, `int*` в `float*`) ведет к UB.
- При работе с однотипными данными (как в примере с `vector<size_t>`) возможна пессимизация кода из-за страха компилятора перед алиасингом.

12.4 Многопоточность, Линковка и ODR: Где ломаются абстракции

В предыдущих главах мы рассматривали код преимущественно в контексте одного потока исполнения и одного модуля трансляции. Однако реальные системные приложения работают в многопоточной среде и собираются из сотен объектных файлов. В этих условиях оптимизации, корректные локально, могут приводить к катастрофическим последствиям глобально.

Компиляторы C++ традиционно оптимизируют код, исходя из модели «as-if single-threaded». Это означает, что любое преобразование кода допустимо, если оно сохраняет наблюдаемое поведение *текущего* потока. Однако, когда память становится разделяемым ресурсом, это предположение вступает в конфликт с моделями согласованности памяти.

12.4.1 Проблема спекулятивных записей (Write Invention)

Одной из самых коварных проблем, возникающих на стыке оптимизации и многопоточности, является «изобретение записи» (Write Invention или Speculative Store).

Рассмотрим функцию, которая инкрементирует глобальную переменную только при выполнении определенного условия.

```
int global_counter = 0;

void process(bool condition) {
    if (condition) {
        global_counter++;
    } else {
        // Heavy computation that doesn't touch global_counter
        heavy_computation();
    }
```

```
}
```

Листинг 12.10 – Исходный код с условной записью

С точки зрения однопоточной логики, компилятор может попытаться избавиться от условного перехода (который может портить предсказание ветвлений) и заменить его на безусловную арифметику с последующей компенсацией.

```
void process_optimized(bool condition) {
    // Speculative write: we always increment the counter
    int temp = global_counter;
    global_counter = temp + 1;

    if (!condition) {
        // If condition was false, revert the change
        global_counter = temp;
        heavy_computation();
    }
}
```

Листинг 12.11 – Некорректная оптимизация (Псевдокод)

Для однопоточной программы эти два варианта эквивалентны. Но в многопоточной среде вариант с оптимизацией вносит *состояние гонки* (Data Race).

Представьте, что параллельно работает второй поток, который читает `global_counter`, когда `condition` ложно.

1. **Исходный код:** Второй поток всегда видит старое значение (записи нет).
2. **Оптимизированный код:** Второй поток может увидеть промежуточное значение (инкрементированное), которое через мгновение будет «откачено» первым потоком.

Примечание

Современные стандарты C++ и модели памяти (например, LLVM Memory Model) явно запрещают компиляторам создавать записи в память (store) на тех путях исполнения, где их не было в исходном коде. Это правило известно как «Do not invent stores». Тем не менее, баги такого рода периодически всплывают в старых версиях компиляторов или на экзотических архитектурах.

12.4.2 One Definition Rule (ODR) и процесс линковки

Еще одна зона риска находится на этапе сборки программы. C++ использует модель раздельной компиляции: каждый `.cpp` файл компилируется в отдельный модуль трансляции (Translation Unit, TU), ничего не зная о других. Связывание этих модулей в единый исполняемый файл выполняет линкер.

Определение: One Definition Rule (ODR)

Правило одного определения гласит:

1. В пределах одного модуля трансляции сущность (переменная, функция, класс) может быть определена только один раз.
2. В всей программе глобальная сущность может быть определена только один раз.
3. **Исключение:** Inline-функции, шаблоны и классы могут быть определены в

нескольких модулях трансляции, но эти определения должны быть **побитово идентичны**.

Механизм Weak Symbols

Когда вы объявляете функцию `inline` (или определяете метод прямо в теле класса в хедере), компилятор помечает этот символ как «слабый» (weak symbol) в объектном файле. Это инструкция для линкера: «Если встретишь несколько определений этого символа, выбери любое одно и отбрось остальные». Линкер не проверяет, что тела функций идентичны — он просто верит программисту на слово.

Это открывает дорогу к нарушению ODR, которое не ловится ни компилятором, ни линкером, но приводит к Runtime-ошибкам.

```
// File: module_a.cpp
// Definition 1: sum
inline int logic(int a, int b) { return a + b; }

void check_a() {
    if (logic(1, 2) != 3) abort(); // Expect 3
}

// File: module_b.cpp
// Definition 2: sum + 1 (ERROR: same name, different body)
inline int logic(int a, int b) { return a + b + 1; }

void check_b() {
    if (logic(1, 2) != 4) abort(); // Expect 4
}
```

Листинг 12.12 – ODR Violation: Разные определения одной inline-функции

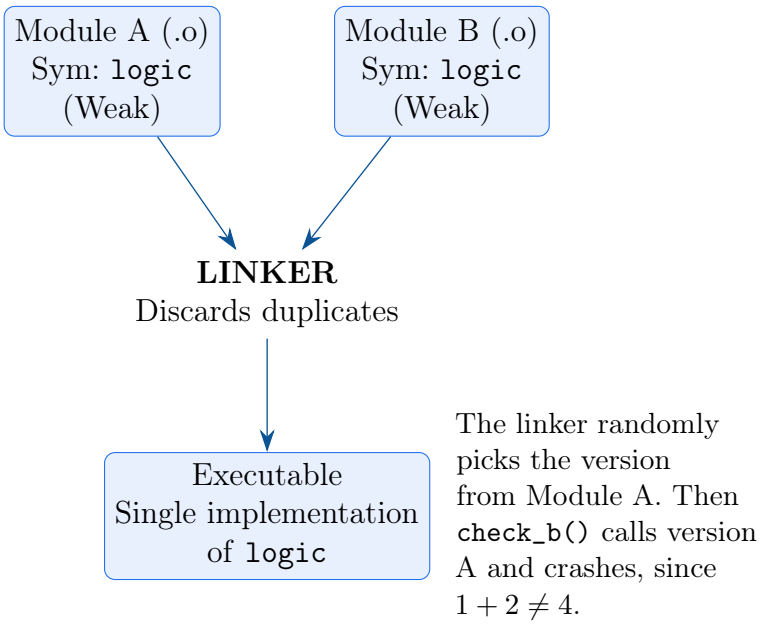


Рис. 12.5 – Процесс выбора реализации inline-функции линкером

Если собрать этот код с оптимизацией `-O2`, компилятор может подставить (заинлайнить) тела функций прямо в места вызова `check_a` и `check_b` *до* этапа линковки. В

этом случае программа может «случайно» заработать корректно. Однако при сборке `-O0` вызовы останутся, линкер выберет одну реализацию, и один из модулей сломается.

Static vs Inline

Чтобы избежать таких проблем для вспомогательных функций, следует использовать ключевое слово `static` (в C) или безымянные пространства имен (в C++). Это сообщает линкеру, что символ имеет *внутреннее связывание* (internal linkage) и не должен быть виден за пределами текущего объектного файла. В таком случае в каждом модуле будет своя независимая копия функции.

12.4.3 Нарушение ABI: Кейс библиотеки Abseil

Еще более сложный случай нарушения ODR происходит не из-за кода, а из-за несовпадения настроек препроцессора и компилятора в разных модулях. Это классическая проблема нарушения Application Binary Interface (ABI).

В библиотеке Google Abseil (и многих других, включая STL) часто используется условная компиляция для добавления отладочной информации.

```
// header.h
struct Container {
    int size;
    void* data;

#ifdef DEBUG_BUILD
    // Additional debug fields
    // This changes sizeof(Container) and offsets below!
    int debug_magic;
    const char* creation_stacktrace;
#endif

    int capacity; // Offset depends on DEBUG_BUILD
};
```

Листинг 12.13 – Условное изменение структуры данных

Представьте, что вы используете предварительно скомпилированную библиотеку, собранную в режиме `Release` (без поля `debug_magic`). Ваше приложение вы собираете в режиме `Debug` (или с включенным Address Sanitizer), где этот макрос определен.

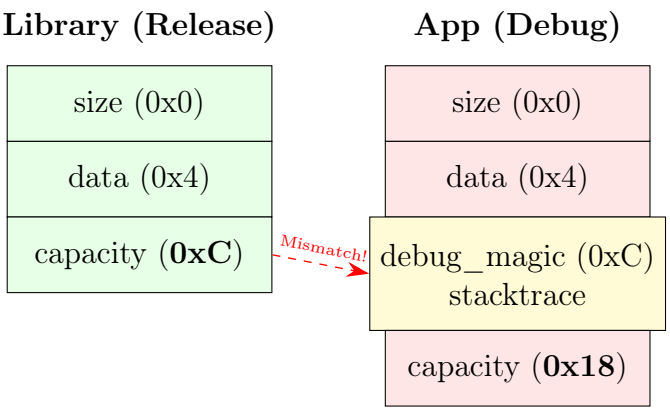


Рис. 12.6 – Несовпадение раскладки памяти (Layout Mismatch)

Когда ваше приложение передает указатель на `Container` в библиотечную функцию,

библиотека ожидает найти поле `capacity` по смещению `0xC`. Однако в вашей версии структуры по этому смещению находится поле `debug_magic`. Библиотека запишет данные в `debug_magic`, думая, что пишет в `capacity`, или наоборот. Это приведет к порче памяти (Memory Corruption), которую крайне сложно отладить, так как ошибка возникает не в момент записи, а спустя долгое время при использовании испорченных данных.

Итоги раздела

- Однопоточные оптимизации могут вносить гонки данных (Data Races) в многопоточный код, если компилятор "изобретает"записи в память.
- One Definition Rule — фундаментальное требование C++. Нарушение ODR (разные тела inline-функций) приводит к неопределенному поведению, так как линкер произвольно выбирает одну из реализаций.
- Слабые символы (Weak Symbols) — механизм, позволяющий множественные определения, но возлагающий ответственность за их идентичность на программиста.
- Нарушение ABI через несовпадение флагов препроцессора (`-DDEBUG`, `-fsanitize`) в разных модулях приводит к несовпадению раскладки структур в памяти. Всегда следите за тем, чтобы все статические библиотеки и основное приложение собирались с идентичными фундаментальными флагами.

Глава 13

Глоссарий