Курс: *Архитектура компьютера и ОС*

Лекция 2: Файловая система и файловые дескрипторы

Лектор: Евгений Соколов

Дата: 08.09.2025

Содержание

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.

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. 1).

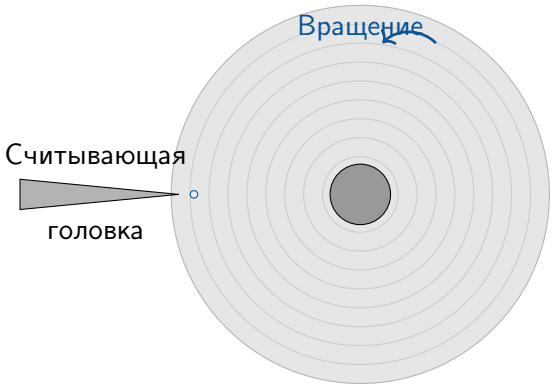

Рис. 1 – Упрощённая схема устройства жёсткого диска (HDD)

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 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.

2.1 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 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 (`-`).

2.3 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).

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*).

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.

3.1.1 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 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).

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.

3.3 Другие важные системные вызовы

3.3.1 `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.

3.3.2 `dup` и `dup2`: Копирование файловых дескрипторов

Эти вызовы создают копию файловый дескриптор.

```c
#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.

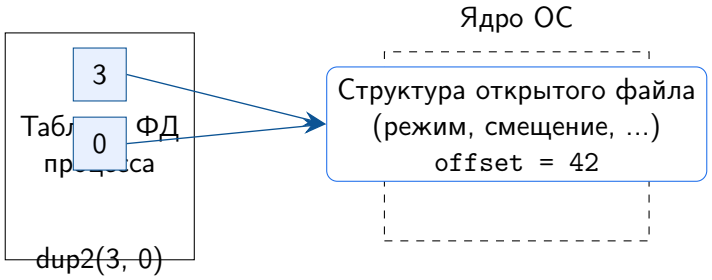

Рис. 2 – Схема работы `dup2`. Оба дескриптора (старый и новый) указывают на одну и ту же структуру открытого файла в ядре и разделяют общее смещение.

Ключевой особенностью является то, что новый и старый файловый дескриптор ссылаются на одну и ту же запись в таблице открытых файлов ядра (рис. 2). Это означает, что они **разделяют общее смещение**: изменение позиции через один файловый дескриптор немедленно отражается на другом.

3.3.3 `pipe`: Создание каналов

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

```c
#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`.

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 – 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 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.

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 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 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`.