**ОБЩАЯ ТЕОРИЯ**

**Файл** — это набор данных, хранящийся на диске или другом носителе, который ОС рассматривает как единицу хранения информации.

**Файл** — это абстракция, предоставляемая операционной системой для хранения и управления данными. На более низком уровне это просто последовательность байтов, которые могут быть записаны, прочитаны или изменены. Тем не менее, концепция "файла" в операционных системах охватывает больше, чем просто набор байтов.

Когда речь идет о таких файлах как /dev/null или /dev/random, важно понимать, что они тоже являются файлами, но специальными.

**/dev/null** — это файл, который принимает все данные, отправленные в него, и ничего не возвращает. Это "черная дыра", куда можно отправить ненужные данные. Это не физический файл на диске, а специальный файл, обрабатываемый ядром.

**/dev/random** — это файл, который генерирует случайные данные (или почти случайные, в зависимости от состояния системы). Это тоже специальный файл, но его цель — предоставить случайные данные для программ.

Они называются специальными файлами устройства. Это просто способ взаимодействия с различными аппаратными ресурсами или виртуальными ресурсами, такими как генераторы случайных чисел или устройства ввода/вывода.

Файлы, такие как /dev/null или /dev/random, не содержат обычных данных, как текстовые или бинарные файлы. Тем не менее, они также управляются системой как файлы, и для них тоже можно использовать стандартные операции с файлами, такие как чтение и запись.

**Файл** — это абстракция, через которую операционная система предоставляет доступ к данным, устройствам или системным ресурсам.

Все файлы имеют несколько общих характеристик, независимо от их содержания и назначения:

**1. Идентификатор**: Каждый файл в операционной системе имеет уникальный идентификатор — имя, по которому его можно найти в файловой системе. Также существует концепция дескрипторов файлов, которые представляют файлы в процессе.

**2. Содержимое**: Это данные, которые могут быть текстом, изображениями, программами, конфигурациями или любым другим типом данных. Это основная цель файла — хранить информацию.

**3. Метаданные**: Включают в себя информацию о файле, такую как права доступа, время создания, последней модификации и другие свойства. Метаданные помогают операционной системе управлять файлами.

**4. Реальные или виртуальные файлы**: Операционная система воспринимает все ресурсы как файлы, будь то обычные файлы, устройства или каналы для ввода-вывода.

**Что объединяет все файлы?**

Все файлы, независимо от того, являются ли они обычными файлами, специальными файлами устройства или виртуальными файлами, имеют несколько общих признаков:

**1. Абстракция для доступа**: Операционная система предоставляет единый интерфейс для работы с файлами. Пользователи и программы могут читать из файлов и записывать в них, не заботясь о том, что скрывается под этой абстракцией (физическое устройство хранения, виртуальный файл и т. д.).

**2. Использование дескрипторов файлов**: Для работы с файлом операционная система использует дескрипторы (или указатели), которые представляют открытые файлы. Это позволяет процессам работать с файлами без необходимости напрямую взаимодействовать с их физическим расположением.

**3. Системные вызовы**: Операционные системы используют системные вызовы для взаимодействия с файлами — такие как open(), read(), write(), close(), stat() и т.д. Это стандартный способ взаимодействия с файлом, будь то обычный файл или устройство.

**Основные свойства файла**:

- Имя файла (например, document.txt)

- Тип файла (обычный, каталог, устройство и т. д.)

- Размер (в байтах)

- Владелец и права доступа (chmod, chown)

- Метаданные (время создания, изменения)

- Содержимое (данные в виде байтов)

**Типы файлов**:

- Обычные файлы (regular files)

Могут содержать текст, код, изображения (.txt, .jpg, .c, a.out)

- Каталоги (директории) (directories)

Хранят список файлов и других директорий. Например: /home/user/, /etc/

- Символические ссылки (symbolic links)

Указывают на другой файл или каталог. Работают как ярлыки.

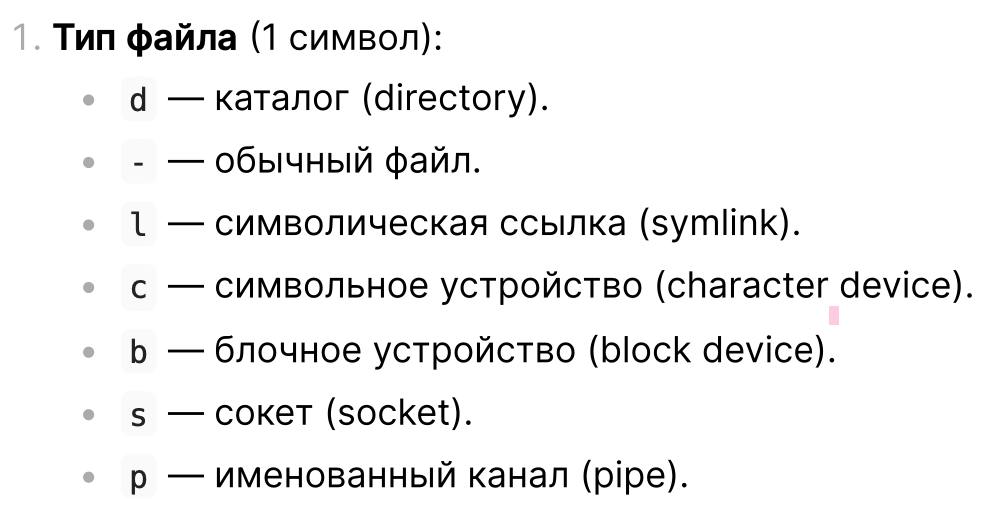
- Специальные файлы (устройства)

Файлы устройств (/dev/sda, /dev/null)

Два типа: *блочные* (диски) и *символьные* (терминалы).

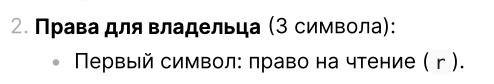
- FIFO (именованные каналы) и сокеты

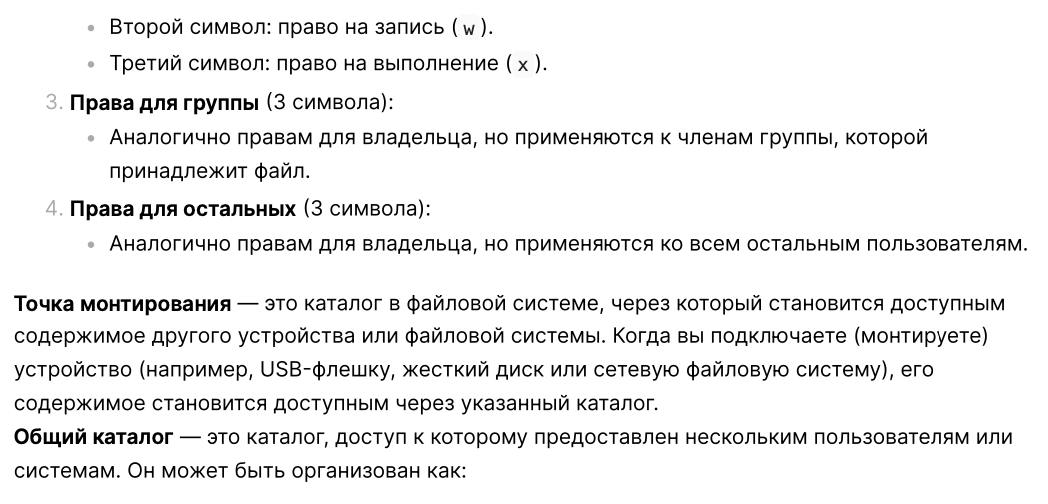
Используются для межпроцессного взаимодействия.



Права доступа состоят из 10 символов.

Первый указывает на тип файла, а остальные 9 разделены на 3 группы по 3 символа.





**Файловая система (ФС)** — это метод организации и хранения данных на диске.

Она управляет файлами, их расположением, доступом и метаданными.

Что делает файловая система?

1. Определяет структуру хранения:

ФС решает, как файлы записываются, читаются, изменяются.

2. Работает с каталогами:

ФС позволяет организовывать файлы в иерархические структуры (/home/user/docs/).

3. Управляет доступом:

Кто может читать/писать файл (chmod, chown).

4. Следит за использованием места:

ФС следит, какие блоки на диске заняты, а какие свободны.

5. Обеспечивает защиту от сбоев:

Журналирование (ext3/ext4) помогает восстанавливать файлы после ошибок.



**Физическая файловая система (FS)** – это то, что на диске

Каждая ФС использует структуры данных для работы с файлами:

- **Superblock** — информация о ФС (размер, состояние, свободные блоки).

- **Inode Table** — метаданные файлов (размер, права, ссылки).

- **Data Blocks** — где хранятся сами данные файлов.

- **Dentry (Directory Entry)** — связывает имена файлов с inode.

**Виды файловых систем**

1. Файловые системы в Linux

ext4 (стандартная в Linux) — журналируемая, быстрая, устойчивая.

XFS (Red Hat) — хороша для больших файлов.

Btrfs (новая) — поддержка снапшотов и самовосстановления.

2. Файловые системы в Windows

NTFS (основная) — поддерживает права доступа, шифрование.

FAT32 (старый стандарт) — работает в Windows, Linux, macOS, но ограничен 4 ГБ на файл.

3. Файловые системы для флешек

exFAT — лучше FAT32, поддерживает большие файлы.

FAT32 — совместимость, но ограничение на размер файлов (4 ГБ).

**Виртуальная файловая система (VFS)** – прослойка в ядре

**VFS (Virtual File System)** — виртуальная файловая система

VFS — это абстрактный интерфейс ядра, позволяющий работать с файлами независимо от конкретной файловой системы (ext4, NTFS, FAT32). Он позволяет программе работать с файлами единым способом, даже если файлы хранятся на разных носителях.

**Как взаимодействуют VFS и реальная файловая система?**

Когда программа открывает файл, например, open("/home/user/file.txt"), происходит следующее:

1 Программа вызывает open("/home/user/file.txt").

2 VFS принимает вызов и проверяет кеш открытых файлов.

3 VFS парсит путь /home/user/file.txt и ищет его в кешах dentry и inode.

4 Если в кеше нет, VFS спрашивает файловую систему (например, ext4).

5 Файловая система ext4 ищет файл на диске, проверяя каталог /home/user/.

6 Каталог /home/user/ – это просто таблица (dentry), в которой записано, что file.txt связан с inode 1234.

7 Файловая система возвращает inode 1234, и VFS кэширует его.

8 Теперь программа может read() данные, а VFS через драйвер передаёт запрос в ext4, которая читает блоки данных с диска.

9 Когда программа закрывает файл (close()), VFS освобождает кеши и уведомляет файловую систему.

Переключение между файловыми системами происходит в момент, когда операционная система получает запрос на работу с файлом, расположенным на разных файловых системах. Это переключение осуществляется с помощью **монтирования (mounting)** файловых систем в определенные точки, т.е. каталог или устройство привязывается к директории в дереве каталогов операционной системы.

**Монтирование файловых систем:** Когда вы монтируете файловую систему (например, на флешке или другом разделе), ядро системы подключает эту файловую систему к дереву каталогов, и она становится доступной для работы.

Например, команда mount /dev/sdb1 /mnt/usb монтирует файловую систему с устройства /dev/sdb1 в каталог /mnt/usb.

После монтирования файловая система становится доступной как обычная часть файловой структуры, и все операции с файлами будут автоматически перенаправляться к соответствующей файловой системе.

**Работа с файлами после монтирования**: После монтирования файловой системы операции на файлах (например, чтение или запись) будут обрабатываться соответствующим драйвером файловой системы. VFS будет направлять запросы на файлы в нужные части, взаимодействуя с конкретной файловой системой через специфичные для нее драйверы.

Когда ты подключаешь флешку (или другое устройство), происходит несколько шагов, которые обеспечивают автоматическое монтирование:

1. Подключение устройства:

Ты вставляешь флешку в USB-порт.

Операционная система (например, Linux) обнаруживает новое устройство через систему udev (или эквивалентную систему на других ОС). Это называется "горячее подключение" (hotplug).

2. udev:

Когда операционная система обнаруживает новое устройство, udev (в Linux) запускает процесс обнаружения устройства и идентифицирует его тип. Например, флешку с файловой системой FAT32.

udev решает, что с этим устройством нужно делать, и обычно вызывает скрипт или процесс для автоматического монтирования устройства.

3. Автоматическое монтирование:

В Linux автоматически срабатывают демоны, такие как udevadm, которые могут также взаимодействовать с сервисами монтирования, такими как udisks или gnome-disks в графических оболочках.

Эти службы монтируют устройство в заранее определённую точку монтирования, например, в каталог /media/ или /run/media/. Это происходит без участия пользователя, то есть ты даже не видишь команду mount, но всё равно можешь сразу начать работать с флешкой.

4. Работа с устройством:

После того, как устройство смонтировано, ты можешь сразу начать работать с файлами на флешке, например, через файловый менеджер или командную строку.

Например, система может автоматически создать каталог в /media/, например, /media/usb0, и это будет точка монтирования для флешки.

Ты видишь это как обычный каталог, и все операции с файлами, такие как копирование или чтение, происходят через этот каталог.

Удаление устройства:

Когда ты "безопасно извлекаешь" флешку, система автоматически размонтирует её, и все данные, которые могли быть в буфере, записываются на устройство.

Обычно это происходит через команду umount, но ты снова не видишь этой команды. Это всё происходит автоматически.

**VDFS (Virtual Distributed File System)** — это виртуальная распределённая файловая система.

Она позволяет объединить несколько различных файловых систем или устройств хранения в единое логическое файловое пространство. Это особенно полезно в облачных системах, кластерах и сетевых хранилищах.

Простыми словами: Представь, что у тебя есть файлы, разбросанные по разным дискам и серверам. VDFS делает так, будто они находятся в одном месте.

**Пример использования VDFS:**

Допустим, у тебя есть: Локальный диск (/mnt/local/), Сетевое хранилище (/mnt/nas/), Облачное хранилище (Google Drive, S3)

VDFS создаёт единый каталог /mnt/vdfs/, в котором файлы из всех этих мест выглядят как единая структура.

Программа запрашивает файл через VDFS → VDFS определяет, где он реально находится → передаёт запрос нужной файловой системе → возвращает данные программе. Принцип работы похож на VFS (Virtual File System), но VDFS работает в распределённых системах и объединяет разные хранилища.

**procfs (процессная файловая система, /proc)** — это виртуальная файловая система, которая предоставляет доступ к информации о процессах и состоянии системы в виде файлов.

- Это НЕ обычная файловая система, потому что файлы в ней не занимают место на диске – они создаются динамически в памяти.

- Используется для взаимодействия с ядром Linux. Находится в /proc и позволяет читать данные ядра как файлы.

- Позволяет получать информацию о процессах, загрузке CPU, памяти, файловых системах, сетевых интерфейсах и многом другом. Ты получаешь информацию о процессоре, но на самом деле этот файл не существует на диске – данные генерируются динамически из ядра.

**Как они взаимодействуют?**

1. Ты запускаешь команду ls /home/user.

2. VFS принимает команду, обрабатывает её и передает дальше.

3. VFS решает, какая файловая система отвечает за /home/user.

4. Если это локальный диск → VFS передает запрос в ext4 (или другую ФС).

5. Если это сетевой ресурс → VFS использует VDFS (например, NFS).

6. Если это информация о процессах → запрос идет в procfs.

Файловая система Linux использует две важные структуры для работы с файлами:

- **inode (индексный дескриптор)** – хранит информацию о файле.

- **dentry (структура каталога)** – связывает имя файла с его inode.

1. **inode** – индексный дескриптор файла

Каждый файл в Linux описывается уникальным inode (индексным узлом).

Это структура данных, которая хранит метаданные файла, но не его имя.

Что хранит inode?

- Размер файла (в байтах).

- Владелец и группа (UID, GID).

- Права доступа (rwx для владельца, группы, других).

- Временные метки (ctime, mtime, atime).

- Список блоков на диске, где хранятся данные файла.

- Количество жёстких ссылок (hard links).

inode **не хранит** имя файла.

Один файл может иметь несколько имён, благодаря hard links. Это работает потому, что имя файла и его данные хранятся отдельно:

- Имена файлов хранятся в dentry (записях каталога).

- Метаданные и указатели на данные хранятся в inode.

Если несколько имен файлов (разные dentry) указывают на один и тот же inode, это значит, что у нас есть жёсткие ссылки.

Как создать жёсткую ссылку?

Используется команда ln без флага -s:

ln file1.txt file2.txtКак это выглядит в файловой системе?

Допустим, у нас есть файл file1.txt:

file1.txt → [dentry] → [inode #1234] → [данные на диске]После создания жёсткой ссылки:

file1.txt → [dentry] ─┐

├──> [inode #1234] → [данные на диске]

file2.txt → [dentry] ─┘Теперь у нас два имени (file1.txt и file2.txt), но они оба указывают на один и тот же inode.

Каждый inode хранит счетчик ссылок (сколько имён на него указывают).

Можно проверить это с помощью команды ls -l:

Вывод:

-rw-r--r-- 2 user user 100 мар 27 12:00 file1.txt

-rw-r--r-- 2 user user 100 мар 27 12:00 file2.txtЗдесь "2" в первой колонке означает, что этот inode имеет две ссылки.

Что происходит при удалении одной ссылки?

Если удалить file1.txt:

rm file1.txtФайл не удалится сразу, потому что у inode всё ещё есть вторая ссылка (file2.txt).

После удаления последней ссылки (file2.txt), inode будет освобождён, и данные файла будут окончательно удалены.

Вопрос: у меня есть файл и один поток начинает его читать, а второй удаляет. что будет? сможем ли мы продолжить чтение?

Ответ: Да, поток, читающий файл, сможет продолжить чтение даже после его удаления другим потоком.

Почему так происходит?

В Linux (и Unix-подобных системах) удаление файла с помощью rm или unlink() не сразу освобождает его данные на диске.

Файл удаляется из каталога (то есть удаляется его имя в dentry), но его содержимое остаётся в системе, пока хотя бы один процесс держит его открытым.

2. **dentry** – запись в каталоге (directory entry)

dentry связывает имя файла с его inode.

Каталог в Linux – это просто список dentry, указывающих на inode.

Что хранит dentry?

- Имя файла (например, file.txt).

- Ссылка на inode, который описывает файл.

- Флаги состояния (активен ли, в кеше ли).

- Ссылку на родительский каталог (чтобы понимать структуру).

**Как inode и dentry работают вместе?**

1. Открытие файла: ФС ищет имя файла (dentry) в каталоге.

2. По dentry получает inode.

3. inode указывает, где на диске лежат данные файла.

4. ФС читает данные из этих блоков.

Пример

echo "Hello" > /home/user/file.txt

cat /home/user/file.txtКаталог /home/user/ содержит dentry для file.txt.

Этот dentry указывает на inode с метаданными файла.

По inode ФС находит блоки с данными на диске.

Читаем Hello из файла.

Функции open() и close() в Linux (и UNIX-подобных системах) используются для работы с файлами на низком уровне, то есть через системные вызовы ядра.

Нельзя сразу использовать read() и write(), потому что они требуют файловый дескриптор, который создается open()!

**Файловый дескриптор (FD)** — это указатель на открытый файл в памяти ядра.

Когда мы передаем имя файла ("file.txt") в open(), оно используется только для поиска и открытия файла. После этого нам уже не нужно его имя – файл ассоциируется с числовым дескриптором.

Если бы read() и write() принимали имя файла, это привело бы к лишним затратам и неэффективности:

1.Открытие файла – это дорого!

Каждый раз, когда мы читаем или пишем в файл, система должна:

- Проверять путь файла (/home/user/file.txt).

- Искать его в dentry cache или на диске.

- Загружать inode (метаданные файла).

- Разрешать права доступа.

Вместо этого open() делает всё один раз, а потом read() просто читает данные через fd.

2. Файлы можно удалять и переименовывать

Допустим, один процесс читает файл:

read("file.txt", buffer, 128);Но что, если другой процесс удалил file.txt или переименовал его?

В случае файлового дескриптора (fd) всё будет работать, потому что файл остается открытым до close(), даже если его удалили!

Но если read() использовал имя файла, он каждый раз искал бы файл заново – и мог бы просто не найти его после удаления.

3. Файловый дескриптор – это абстракция

Один раз открыли файл (open()), получили fd, и дальше не важно, какая это файловая система: локальная, виртуальная или сетевая

**open()** – это системный вызов, который сообщает ядру, что процесс хочет работать с файлом.

**read()** читает данные из файла, используя файловый дескриптор

**write()** записывает данные в файл.

**close()** закрывает файловый дескриптор и освобождает ресурсы ядра.

**Что происходит на уровне ядра?**

Когда вызываем **open("file.txt", O\_RDONLY);**

1. Проверяется путь файла (file.txt).

2. VFS (Virtual File System) ищет file.txt в соответствующей файловой системе (ext4, NTFS и т. д.).

3. Создается запись в dentry cache (кэш имен файлов).

4. Считывается inode файла (метаданные файла).

5. Ядро выделяет файловый дескриптор (например, 3).

Когда вызываем **read(fd, buffer, 128);**

1. Проверяется, есть ли fd в таблице открытых файлов процесса.

2. Находится inode файла через dentry.

3. Читаются данные из кеша страницы или с диска.

4. Записываются в buffer программы.

Когда вызываем **close(fd);**

1. Удаляется запись о файле из таблицы открытых файлов процесса.

2. Ядро проверяет, есть ли другие процессы, которые используют этот файл.

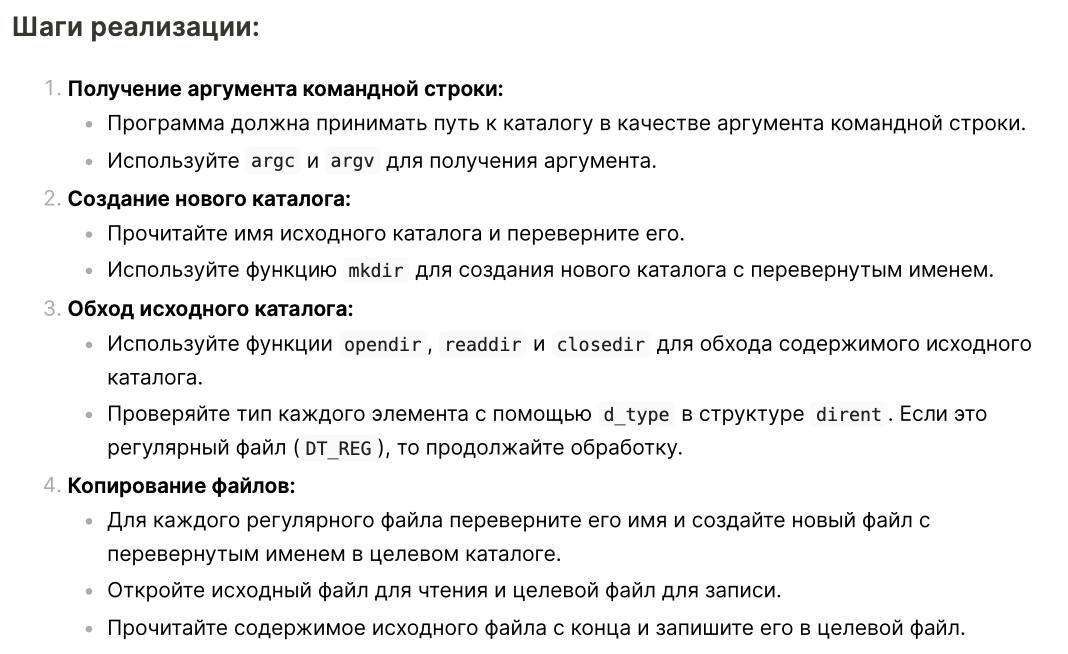
3. Если нет – inode освобождается, дескриптор удаляется.

**ЗАДАНИЕ**

1. Написать программу, которая копирует каталог “задом наперед”. Программа получает в качестве аргумента путь к каталогу. Далее:

a. Программа создает каталог с именем заданного каталога, прочитанного наоборот. Если задан каталог “qwerty”, то должен быть создан каталог “ytrewq”.

b. Программа копирует все регулярные файлы из исходного каталога в целевой (пропуская файлы другого типа), переворачивая их имена исодержимое. То есть с именами файлов поступаем также как и с именем каталога, а содержимое копируется начиная с последнего байта и до нулевого.



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <errno.h>

void ReverseString(char \*string) {

int length = strlen(string);

for (int i = 0; i < length / 2; i++) {

char temp = string[i];

string[i] = string[length - i - 1]; //hello

string[length - i - 1] = temp; //oellh

}

}

void ReverseFileContent(const char \*src\_path, const char \*dst\_path) {

int src\_fd = open(src\_path, O\_RDONLY); //только читаю

int dst\_fd = open(dst\_path, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644); // rw- r-- r-- 4+2 4 4

off\_t size\_file = lseek(src\_fd, 0, SEEK\_END);

char buffer;

for (off\_t i = size\_file - 1; i >= 0; i--) {

lseek(src\_fd, i, SEEK\_SET);

read(src\_fd, &buffer, 1);

write(dst\_fd, &buffer, 1);

}

close(src\_fd);

close(dst\_fd);

}

void CopyReverseCatalog(const char\* src\_dir) {

char dst\_dir[256];

if (strlen(src\_dir) >= sizeof(dst\_dir)) {

fprintf(stderr, "Ошибка: путь к каталогу слишком длинный\n");

return;

}

strcpy(dst\_dir, src\_dir); // Копируем путь к каталогу

ReverseString(dst\_dir); // Переворачиваем путь

if (mkdir(dst\_dir, 0755) == -1 && errno != EEXIST) {

perror("Ошибка при создании каталога");

return;

}

DIR \*dir = opendir(src\_dir);

if (!dir) {

perror("Ошибка при открытии каталога");

return;

}

struct dirent \*entry;

while ((entry = readdir(dir)) != NULL) {

if (entry->d\_type == DT\_REG) {

char src\_file\_path[512];

char dst\_file\_path[512];

snprintf(src\_file\_path, sizeof(src\_file\_path), "%s/%s", src\_dir, entry->d\_name);

snprintf(dst\_file\_path, sizeof(dst\_file\_path), "%s/", dst\_dir);

ReverseString(entry->d\_name);

strncat(dst\_file\_path, entry->d\_name, sizeof(dst\_file\_path) - strlen(dst\_file\_path) - 1);

ReverseFileContent(src\_file\_path, dst\_file\_path);

}

}

closedir(dir);

}

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <directory>\n", argv[0]);

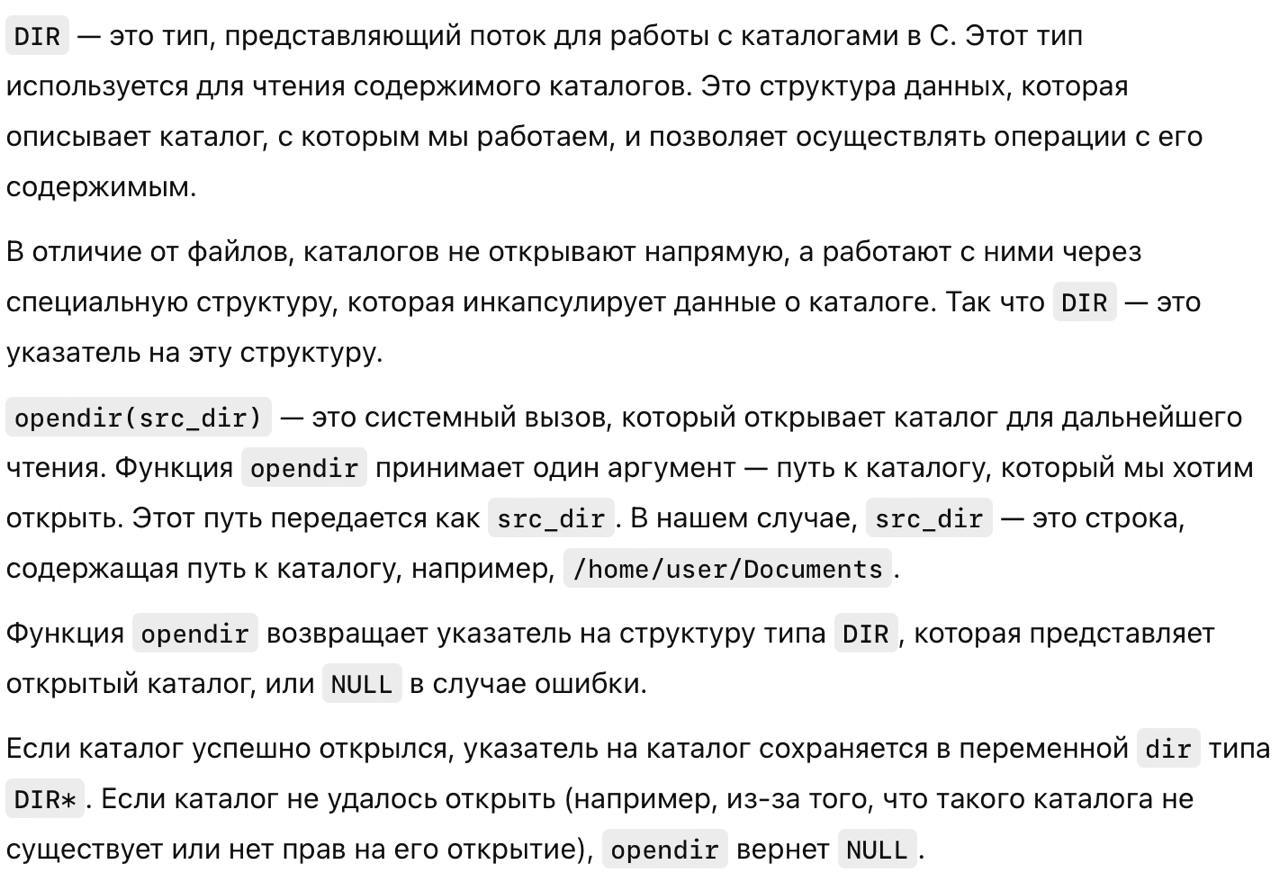
exit(EXIT\_FAILURE);

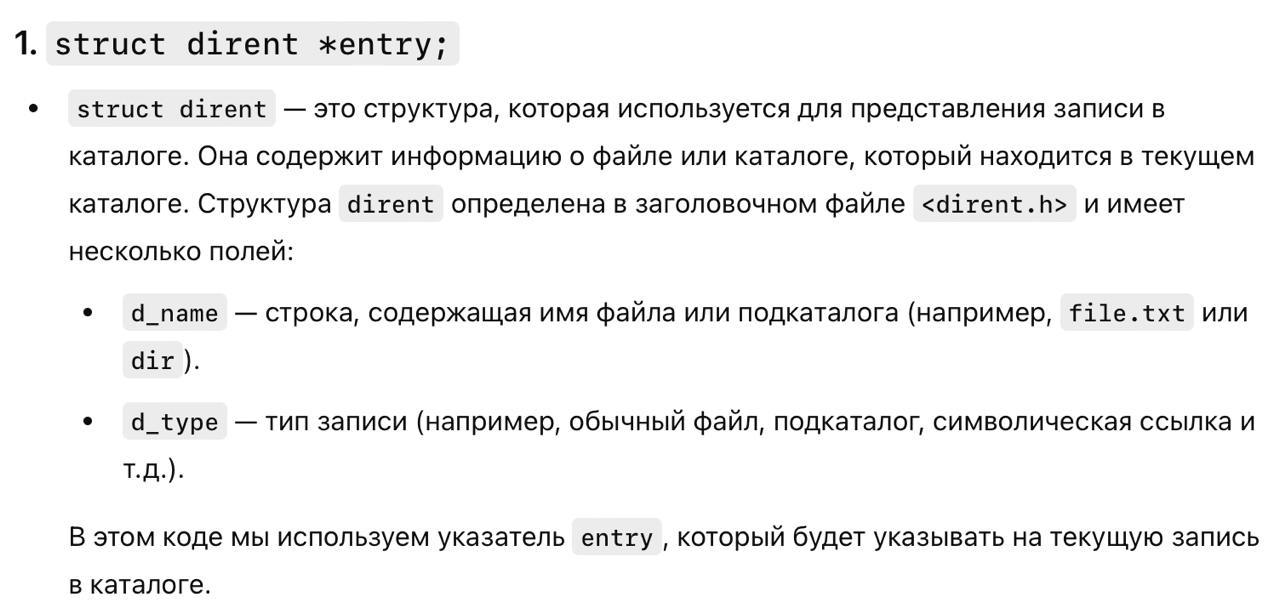
}

CopyReverseCatalog(argv[1]);

return 0;

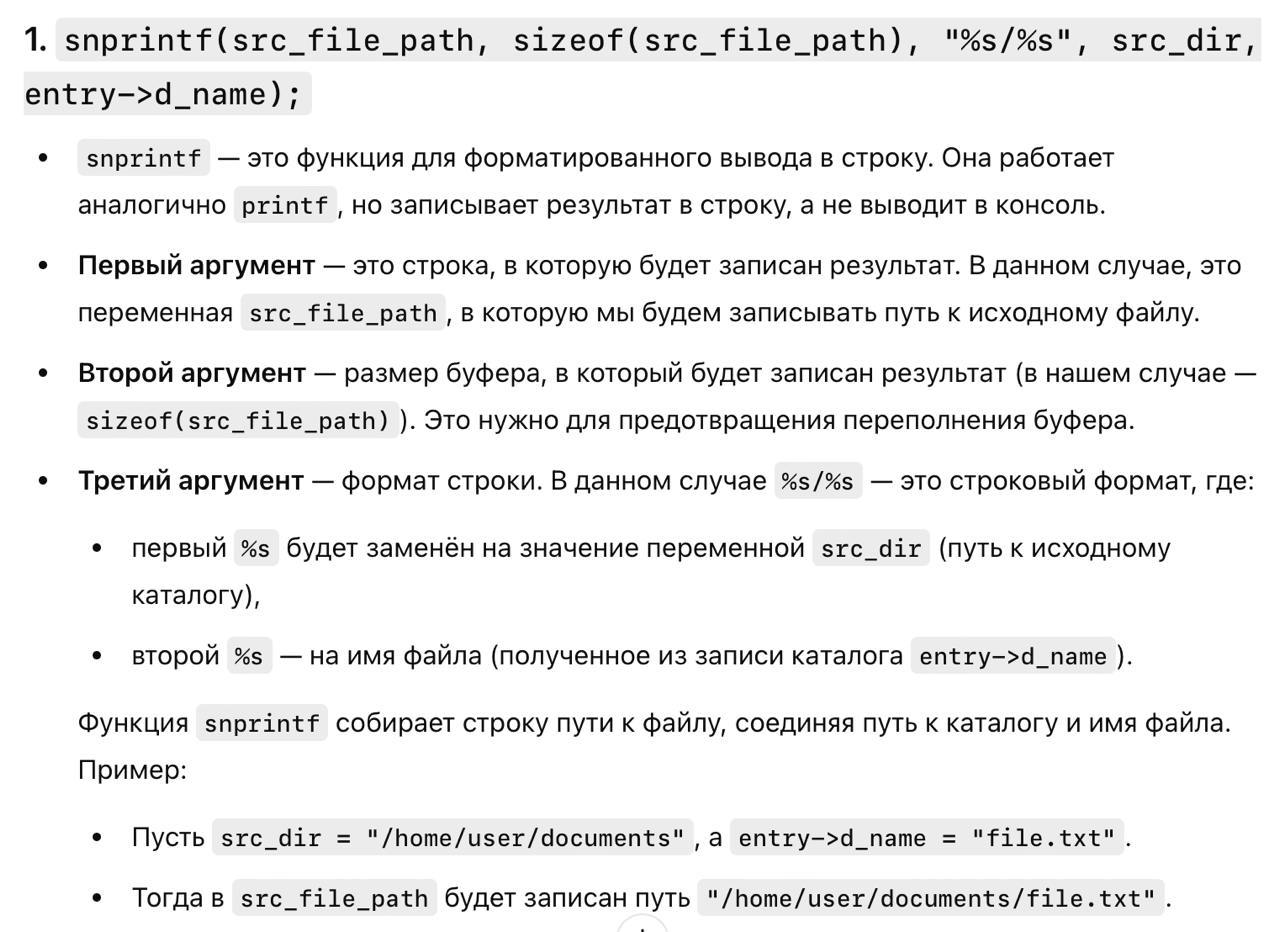
}



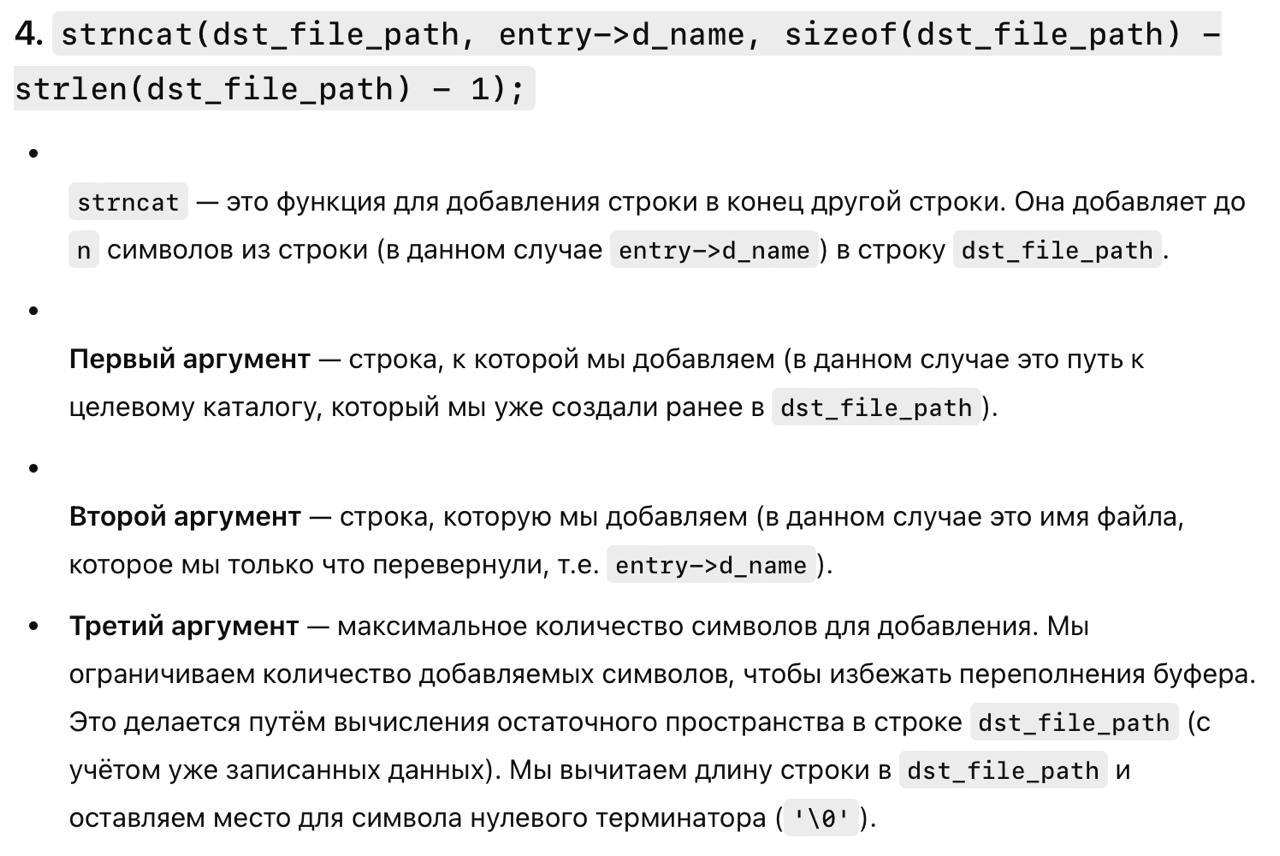


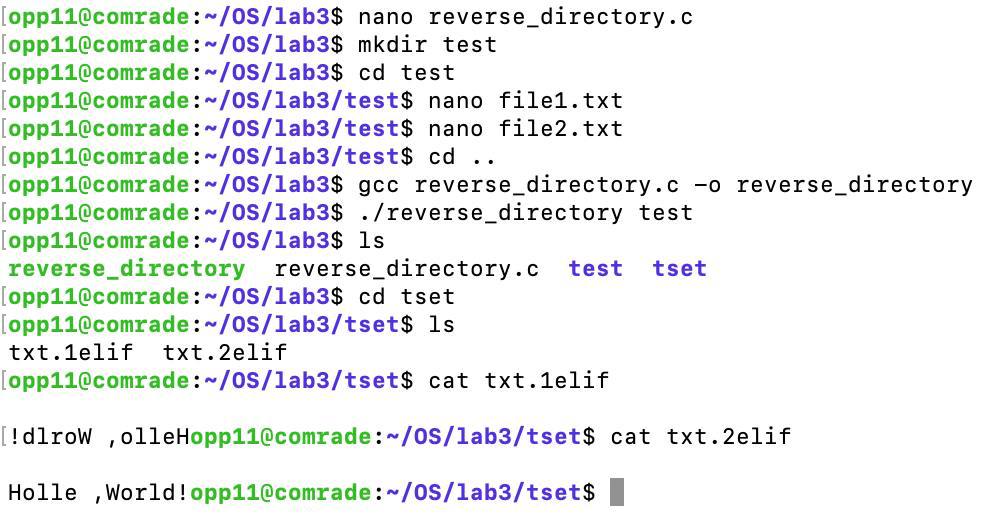
DIR — это структура, которая используется для управления состоянием открытого каталога.

dirent — это структура, которая описывает отдельную запись (файл или каталог) в каталоге.







Работа программы:  


Немного измененная программа (читает не по 1 байту, а блоком 4096 байт)  
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <errno.h>

**void** ReverseString(**char** \*string) {

**int** length = strlen(string);

**for** (**int** i = 0; i < length / 2; i++) {

**char** temp = string[i];

string[i] = string[length - i - 1]; //oello

string[length - i - 1] = temp;//oellh

} //olleh

}

**void** ReverseFileContent(**const** **char** \*src\_path, **const** **char** \*dst\_path) {

**int** src\_fd = open(src\_path, O\_RDONLY); //только читаю

**int** dst\_fd = open(dst\_path, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644); // rw- r-- r--

off\_t size\_file = lseek(src\_fd, 0, SEEK\_END); // узнаю размер файла

off\_t remaining = size\_file;

**char** buffer[4096];

**while** (remaining > 0) {

off\_t chunk\_size = (remaining >= **sizeof**(buffer)) ? **sizeof**(buffer) : remaining;

remaining -= chunk\_size;

lseek(src\_fd, remaining, SEEK\_SET);

read(src\_fd, buffer, chunk\_size);

write(dst\_fd, buffer, chunk\_size);

}

close(src\_fd);

close(dst\_fd);

}

**void** CopyReverseCatalog(**const** **char**\* src\_dir) {

**char** dst\_dir[256];

**if** (strlen(src\_dir) >= **sizeof**(dst\_dir)) {

fprintf(stderr, "Ошибка: путь к каталогу слишком длинный\n");

**return**;

}

strcpy(dst\_dir, src\_dir); // Копируем путь к каталогу

ReverseString(dst\_dir); // Переворачиваем путь

**if** (mkdir(dst\_dir, 0755) == -1 && errno != EEXIST) { // создаем каталог

perror("Ошибка при создании каталога");

**return**;

}

DIR \*dir = opendir(src\_dir); // открываем каталог

**if** (!dir) {

perror("Ошибка при открытии каталога");

**return**;

}

**struct** dirent \*entry; // чтение записей из каталога. d\_name — строка, содержащая имя файла или подкаталога (например, file.txt или dir). d\_type — тип записи (например, обычный файл, подкаталог, символическая ссылка и т.д.).

**while** ((entry = readdir(dir)) != **NULL**) {

**if** (entry->d\_type == DT\_REG) { // проверка на регулярный файл

**char** src\_file\_path[512];

**char** dst\_file\_path[512];

snprintf(src\_file\_path, **sizeof**(src\_file\_path), "%s/%s", src\_dir, entry->d\_name); // src\_dir = "/home/user/documents", entry->d\_name = "file.txt", src\_file\_path = "/home/user/documents/file.txt"

snprintf(dst\_file\_path, **sizeof**(dst\_file\_path), "%s/", dst\_dir); // dst\_dir = "/home/user/documents/"

ReverseString(entry->d\_name); // entry->d\_name = "file.txt", после вызова ReverseString имя файла станет "txet.elif"

strncat(dst\_file\_path, entry->d\_name, **sizeof**(dst\_file\_path) - strlen(dst\_file\_path) - 1); // dst\_file\_path = "/home/user/backup/txet.elif"

ReverseFileContent(src\_file\_path, dst\_file\_path);

}

}

closedir(dir);

}

**int** main(**int** argc, **char** \*argv[]) {

**if** (argc != 2) {

fprintf(stderr, "Usage: %s <directory>\n", argv[0]);

exit(EXIT\_FAILURE);

}

CopyReverseCatalog(argv[1]);

**return** 0;

}

Вследствие чего работает она значительно быстрее:  
