3 Задача.

В файлике **могут** быть ошибки, может немного поехала нумерация :)
Так как файлики пошли по группам, давайте замьютимся(взаимная подписка) на гите.
https://www.youtube.com/watch?v=sllqqfz5OE - лекция.

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.

- **Регулярные файлы** (regular files) — это обычные файлы, которые содержат данные (текст, изображения, бинарные данные и т.д.). Они не являются директориями, символическими ссылками, устройствами или другими специальными типами файлов.
- В Unix-подобных системах тип файла можно определить с помощью функции `stat` и макросов, таких как `S_ISREG`.
- **Символическая ссылка** (или **симлинк**, от англ. *symbolic link*) — это специальный тип файла в Unix-подобных операционных системах, который служит указателем на другой файл или директорию. Символическая ссылка похожа на ярлык в Windows, но обладает большей гибкостью и мощью. (мы еще подробнее во втором пункте об этом поговорим :0)
  Немножко теории

Структура прав доступа:

Права доступа состоят из 10 символов. Первый символ указывает на тип файла, а остальные 9 символов разделены на три группы по три символа:

```
ls -l
```

1. **Тип файла** (1 символ):
   - `d` — каталог (directory).
   - `-` — обычный файл.
   - `l` — символическая ссылка (symlink).
   - `c` — символьное устройство (character device).
   - `b` — блочное устройство (block device).
   - `s` — сокет (socket).
   - `p` — именованный канал (pipe).
2. **Права для владельца** (3 символа):

- Первый символ: право на чтение (`r`).
- Второй символ: право на запись (`w`).
- Третий символ: право на выполнение (`x`).

3. **Права для группы** (3 символа):
   - Аналогично правам для владельца, но применяются к членам группы, которой принадлежит файл.
4. **Права для остальных** (3 символа):
   - Аналогично правам для владельца, но применяются ко всем остальным пользователям.

Точка монтирования — это каталог в файловой системе, через который становится доступным содержимое другого устройства или файловой системы. Когда вы подключаете (монтируете) устройство (например, USB-флешку, жесткий диск или сетевую файловую систему), его содержимое становится доступным через указанный каталог.
Общий каталог — это каталог, доступ к которому предоставлен нескольким пользователям или системам. Он может быть организован как:

Шаги реализации:

1. **Получение аргумента командной строки:**
   - Программа должна принимать путь к каталогу в качестве аргумента командной строки.
   - Используйте `argc` и `argv` для получения аргумента.
2. **Создание нового каталога:**
   - Прочитайте имя исходного каталога и переверните его.
   - Используйте функцию `mkdir` для создания нового каталога с перевернутым именем.
3. **Обход исходного каталога:**
   - Используйте функции `opendir` , `readdir` и `closedir` для обхода содержимого исходного каталога.
   - Проверяйте тип каждого элемента с помощью `d_type` в структуре `dirent` . Если это регулярный файл (`DT_REG`), то продолжайте обработку.
4. **Копирование файлов:**
   - Для каждого регулярного файла переверните его имя и создайте новый файл с перевернутым именем в целевом каталоге.
   - Откройте исходный файл для чтения и целевой файл для записи.
   - Прочитайте содержимое исходного файла с конца и запишите его в целевой файл.

```
#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 reverse_string(char *string){

int len= strlen(string);
```

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

char temp = string[i];

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

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

}//olleh

}

void reverse_file_content(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 copy_reverse_catalog(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
reverse_string(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);
reverse_string(entry->d_name);
strncat(dst_file_path, entry->d_name, sizeof(dst_file_path) - strlen(dst_file_path) - 1);
reverse_file_content(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);

}

copy_reverse_catalog(argv[1]);

return 0;

}
```

DIR - **структура**, определенная в стандартной библиотеке C (заголовочный файл `<dirent.h>`).

- Она используется для хранения информации об открытом каталоге.
  `struct dirent` — это структура, которая содержит информацию об элементе каталога. Её основные поля:
- **Дескриптор каталога**: Уникальный идентификатор, используемый операционной системой для доступа к каталогу.
- **Текущая позиция в каталоге**: Информация о том, какой элемент каталога будет прочитан следующим.
- **Буфер для чтения**: Временное хранилище для данных, считанных из каталога.
  Для работы с каталогами используются следующие функции:

1. **`opendir`**:
   - Открывает каталог и возвращает указатель на структуру `DIR`.
   - Пример:

```
DIR *dir = opendir("/path/to/directory");
       if (!dir) {
           perror("Ошибка при открытии каталога");
           return;
       }
```

2. **`readdir`**:
   - Читает очередной элемент каталога и возвращает указатель на структуру `struct dirent`.

- Пример:

```
  struct dirent *entry;
while ((entry = readdir(dir)) != NULL) {
    printf("Имя элемента: %s\n", entry->d_name);
}
```

3. `closedir`:
   - Закрывает каталог и освобождает связанные ресурсы.
   - Пример:

```
    closedir(dir);
```

```
- **`d_name`**:

    - Имя элемента (файла или подкаталога).

    - Это строка, завершающаяся нулевым символом (`\0`).

- **`d_type`**:

    - Тип элемента. Может быть одним из следующих:

        - `DT_REG` — регулярный файл.

        - `DT_DIR` — каталог.

        - `DT_LNK` — символьная ссылка.

        - `DT_UNKNOWN` — тип неизвестен.


---

### 1. **`struct dirent *entry` — что это за структура?**

`struct dirent` — это структура, которая используется для представления информации о элементах каталога (файлах и подкаталогах). Она определена в заголовочном файле `<dirent.h>`. Основные поля этой структуры:

- **`d_name`**: Имя файла или подкаталога (строка).
- **`d_type`**: Тип элемента каталога (например, обычный файл, каталог, символическая ссылка и т.д.).
```

Пример:
```c
struct dirent {
    ino_t d_ino;          // Номер inode (не всегда используется)
    char d_name[256];     // Имя файла или каталога
    unsigned char d_type; // Тип элемента (например, DT_REG для обычного файла)
};
```

---

2. `readdir` — что это за функция? Что она делает?

`readdir` — это функция, которая читает следующий элемент каталога. Она принимает указатель на структуру `DIR` (которая представляет открытый каталог) и возвращает указатель на структуру `struct dirent`, содержащую информацию о текущем элементе каталога.

- **Прототип функции:**

```c
struct dirent *readdir(DIR *dirp);
```

- **Возвращаемое значение:**
  - Указатель на структуру `struct dirent`, если элемент каталога успешно прочитан.
  - `NULL`, если достигнут конец каталога или произошла ошибка.

Пример использования:

```c
struct dirent *entry;
while ((entry = readdir(dir)) != NULL) {
    printf("Имя файла: %s\n", entry->d_name);
}
```

---

3. `DIR *dir = opendir(src_dir);` — что это?

`opendir` — это функция, которая открывает каталог для чтения его содержимого. Она принимает путь к каталогу и возвращает указатель на структуру `DIR`, которая представляет открытый каталог.

- **Прототип функции:**

```c
DIR *opendir(const char *name);
```

- **Возвращаемое значение:**
  - Указатель на структуру `DIR`, если каталог успешно открыт.

- `NULL`, если каталог не удалось открыть (например, если он не существует или нет прав доступа).

Пример:

```
DIR *dir = opendir("/path/to/directory");
if (dir == NULL) {
    perror("Ошибка при открытии каталога");
    return;
}
```

---

4. `DIR` — что это? Это тоже структура?

Да, `DIR` — это структура, которая представляет открытый каталог. Она используется для хранения состояния каталога при его чтении. Конкретное содержимое этой структуры зависит от реализации операционной системы, и обычно программисту не нужно знать её детали. Достаточно использовать функции, такие как `opendir`, `readdir` и `closedir`, для работы с каталогами.

---

5. В чем отличие `struct dirent *entry` от `DIR *dir`?

- **`DIR *dir`**:
  - Это указатель на структуру, которая представляет открытый каталог.
  - Используется для хранения состояния каталога при его чтении.
  - Пример: `DIR *dir = opendir("/path/to/directory");`
- **`struct dirent *entry`**:
  - Это указатель на структуру, которая содержит информацию о конкретном элементе каталога (например, имя файла и его тип).
  - Используется для получения информации о каждом элементе каталога при чтении.
  - Пример: `struct dirent *entry = readdir(dir);`

---

6. На что мы проверяем `NULL` в `while ((entry = readdir(dir)) != NULL)`?

Функция `readdir` возвращает `NULL`, когда достигнут конец каталога или произошла ошибка. В цикле `while` мы проверяем, что `readdir` вернула не `NULL`, то есть что ещё есть элементы каталога для чтения.

- Если `entry != NULL`, значит, есть ещё элементы каталога, и мы можем обработать текущий элемент.
- Если `entry == NULL`, значит, каталог закончился, и цикл завершается.

7. `snprintf` — почему тут используется именно он? Какие у него аргументы?

`snprintf` — это безопасная версия функции `sprintf`, которая форматирует строку и записывает её в буфер, но с ограничением на количество записываемых символов. Это помогает избежать переполнения буфера.

- **Прототип функции:**

```
int snprintf(char *str, size_t size, const char *format, ...);
```

- **Аргументы:**
  - `str`: Указатель на буфер, куда будет записана строка.
  - `size`: Максимальное количество символов, которые можно записать в буфер (включая завершающий нулевой символ `\0`).
  - `format`: Строка формата (как в `printf`).
  - `...`: Аргументы для подстановки в строку формата.

Пример:

```
char buffer[100];
snprintf(buffer, sizeof(buffer), "Hello, %s!", "world");
// В buffer будет записано: "Hello, world!"
```

В вашем коде:

```
snprintf(src_file_path, sizeof(src_file_path), "%s/%s", src_dir, entry->d_name);
```

- Формируется полный путь к исходному файлу, объединяя путь к каталогу (`src_dir`) и имя файла (`entry->d_name`).
- `sizeof(src_file_path)` гарантирует, что не произойдет переполнение буфера.

---

8. Что происходит в этом блоке кода?

```
snprintf(src_file_path, sizeof(src_file_path), "%s/%s", src_dir, entry->d_name); //
/home/user/file.txt
snprintf(dst_file_path, sizeof(dst_file_path), "%s/", dst_dir); // /home/user/reversed/
reverse_string(entry->d_name);
strncat(dst_file_path, entry->d_name, sizeof(dst_file_path) - strlen(dst_file_path) - 1);
// /home/user/reversed/file.txt
reverse_file_content(src_file_path, dst_file_path);
```

- **`snprintf(src_file_path, ...)`**:

    - Формирует полный путь к исходному файлу, объединяя путь к каталогу (`src_dir`) и имя файла (`entry->d_name`).
- **`snprintf(dst_file_path, ...)`** :
    - Формирует путь к новому каталогу (`dst_dir`) с добавлением `/` в конце.
- **`reverse_string(entry->d_name)`** :
    - Переворачивает имя файла (например, `file.txt` становится `txt.elif`).
- **`strncat(dst_file_path, ...)`** :
    - Добавляет перевернутое имя файла к пути нового каталога, формируя полный путь к новому файлу.
- **`reverse_file_content(src_file_path, dst_file_path)`** :
    - Копирует содержимое исходного файла в новый файл, предварительно перевернув его.

---

2. Программа для работы с файлами, каталогами и ссылками

Общее описание:

Написать программу, которая **создает, читает, изменяет права доступа** и **удаляет следующие объекты: файлы, каталоги, символьные и жесткие ссылки.** Для определения того какая именно функция должна быть исполнена предлагается иметь необходимое количество жестких ссылок на исполняемый файл с именами соответствующими выполняемому действию и в программе выполнять функцию соответствующую имени жесткой ссылки. Программа должна уметь:
a. создать каталог, указанный в аргументе;
b. вывести содержимое каталога, указанного в аргументе;
c. удалить каталог, указанный в аргументе;
d. создать файл, указанный в аргументе;
e. вывести содержимое файла, указанного в аргументе;
f. удалить файл, указанный в аргументе;
g. создать символьную ссылку на файл, указанный в аргументе;
h. вывести содержимое символьной ссылки, указанный в аргументе;
i. вывести содержимое файла, на который указывает символьная ссылка, указанная в аргументе;
j. удалить символьную ссылку на файл, указанный в аргументе;
k. создать жесткую ссылку на файл, указанный в аргументе;
l. удалить жесткую ссылку на файл, указанный в аргументе;
m. вывести права доступа к файлу, указанному в аргументе и количество жестких ссылок на него;
n. изменить права доступа к файлу, указанному в аргументе.

Теория: (очень хорошо на лекции рассказали об этом ***советую к просмотру***, мы повторим)

1. Inode (индексный дескриптор)

Что такое inode?

Inode — это структура данных в файловой системе, которая хранит метаданные о файле и указывает на его данные. Каждый файл в файловой системе Unix/Linux имеет уникальный inode, который содержит следующую информацию:

- Тип файла (обычный файл, каталог, символьная ссылка и т.д.).
- Права доступа (read, write, execute).
- Владелец и группа файла.
- Размер файла.
- Время создания, изменения и доступа.
- Указатели на блоки данных файла на диске.

Как это работает?

- Когда вы создаете файл, файловая система выделяет для него inode и записывает в него метаданные.
- Inode не содержит имени файла. Имя файла хранится в каталоге, который связывает имя файла с его inode.
- Например, если у вас есть файл `file.txt`, то в каталоге будет запись: `file.txt -> inode 12345`.

Пример:

```
$ ls -li
12345 -rw-r--r-- 1 user group 13 Oct 10 12:00 file.txt
```

- `12345` — это номер inode.
- `-rw-r--r--` — права доступа.
- `1` — количество жестких ссылок на этот inode.
- `user` и `group` — владелец и группа.
- `13` — размер файла в байтах.
- `Oct 10 12:00` — время последнего изменения.
- `file.txt` — имя файла.

---

2. Жесткие ссылки (Hard Links)

Что такое жесткая ссылка?

Жесткая ссылка — это дополнительное имя для существующего файла. Она указывает на тот же inode, что и оригинальный файл. Таким образом, жесткая ссылка и оригинальный файл равноправны: они ссылаются на одни и те же данные на диске.

Как это работает?

- Когда вы создаете жесткую ссылку, файловая система создает новую запись в каталоге, которая связывает новое имя с тем же inode.
- Количество жестких ссылок на inode увеличивается на 1.

- Удаление одного имени (оригинального файла или жесткой ссылки) не удаляет данные, пока есть хотя бы одна ссылка на inode.

Пример:

```
$ echo "Hello, World!" > file.txt
$ ln file.txt hardlink.txt  # Создаем жесткую ссылку
$ ls -li
12345 -rw-r--r-- 2 user group 13 Oct 10 12:00 file.txt
12345 -rw-r--r-- 2 user group 13 Oct 10 12:00 hardlink.txt
```

- Оба файла (`file.txt` и `hardlink.txt`) имеют одинаковый inode (`12345`) и количество ссылок (`2`).

Для чего нужны жесткие ссылки?

- **Экономия места**: Жесткие ссылки не занимают дополнительного места на диске, так как они ссылаются на те же данные.
- **Резервное копирование**: Если удалить оригинальный файл, данные останутся доступными через жесткую ссылку.
- **Организация файлов**: Можно создать несколько имен для одного файла в разных каталогах.

Ограничения:

- Жесткие ссылки нельзя создавать для каталогов (только для файлов).
- Жесткие ссылки не могут пересекать границы файловых систем (т.е. нельзя создать жесткую ссылку на файл в другой файловой системе).

---

3. Символьные ссылки (Symbolic Links, Symlinks)

Что такое символьная ссылка?

Символьная ссылка — это файл, который содержит путь к другому файлу или каталогу. *Это похоже на ярлык в Windows.*

Как это работает?

- Символьная ссылка — это отдельный файл, который содержит путь к целевому файлу или каталогу.
- Когда вы обращаетесь к символьной ссылке, операционная система перенаправляет вас к целевому файлу.
- Если целевой файл удален, символьная ссылка становится "битой" (недействительной).

Пример:

```
$ echo "Hello, World!" > file.txt
$ ln -s file.txt symlink.txt  # Создаем символьную ссылку
$ ls -l
```

```
-rw-r--r-- 1 user group 13 Oct 10 12:00 file.txt
lrwxrwxrwx 1 user group  8 Oct 10 12:00 symlink.txt -> file.txt
```

- `symlink.txt` — это символьная ссылка, которая указывает на `file.txt`.
- Права доступа `lrwxrwxrwx` указывают, что это символьная ссылка.

Для чего нужны символьные ссылки?

- **Гибкость**: Символьные ссылки могут указывать на файлы или каталоги в других файловых системах.
- **Организация файлов**: Можно создать ссылки на файлы в разных каталогах без дублирования данных.
- **Упрощение путей**: Символьные ссылки могут сокращать длинные пути.

Ограничения:

- Символьные ссылки занимают место на диске (хранят путь к файлу).
- Если целевой файл удален, ссылка становится недействительной.

---

4. Сравнение жестких и символьных ссылок

Характеристика	Жесткая ссылка	Символьная ссылка
Тип	Дополнительное имя для файла	Файл, содержащий путь к цели
Inode	Совпадает с оригинальным файлом	Имеет собственный inode
Место на диске	Не занимает места	Занимает место (хранит путь)
Пересечение ФС	Нет	Да
Удаление цели	Данные остаются	Ссылка становится "битой"
Создание для каталогов	Нет	Да

---

5. Для чего нужны ссылки? Какие проблемы они решают?

Жесткие ссылки:

- **Экономия места**: Не нужно дублировать данные.
- **Резервное копирование**: Удаление одного имени не удаляет данные.
- **Организация файлов**: Можно иметь несколько имен для одного файла.

Символьные ссылки:

- **Гибкость**: Можно ссылаться на файлы в других файловых системах.

- **Упрощение путей**: Сокращение длинных путей.
- **Организация файлов**: Создание ссылок на часто используемые файлы или каталоги.

Конечно! Понимание разницы между **относительным** и **абсолютным** путями очень важно при работе с файловой системой в программировании и в повседневном использовании компьютера. Давайте разберем, что это такое, как они работают и когда их использовать.

---

1. Абсолютный путь

Что это?

Абсолютный путь — это полный путь к файлу или директории, начиная с **корня файловой системы**. Корень файловой системы в Unix-подобных системах обозначается символом `/`, а в Windows — буквой диска (например, `C:\`).

Примеры:

- В Unix-подобных системах:
  - `/home/user/documents/file.txt`
  - `/usr/bin/bash`
- В Windows:
  - `C:\Users\user\Documents\file.txt`
  - `D:\Projects\README.md`

Особенности:

- Всегда начинается с корня файловой системы.
- Не зависит от текущей рабочей директории.
- Уникально идентифицирует файл или директорию в системе.

Когда использовать?

- Когда нужно точно указать расположение файла или директории, независимо от текущего местоположения.
- В скриптах или программах, где важно быть уверенным в правильности пути.

---

2. Относительный путь

Что это?

Относительный путь — это путь к файлу или директории, который указывается **относительно текущей рабочей директории**. Текущая рабочая директория — это директория, в которой вы находитесь в данный момент (например, в терминале или в программе).

Примеры:

- Если текущая рабочая директория — `/home/user`:
  - `documents/file.txt` (указывает на `/home/user/documents/file.txt`).
  - `../other_user/file.txt` (указывает на `/home/other_user/file.txt`).
- В Windows, если текущая директория — `C:\Users\user`:
  - `Documents\file.txt` (указывает на `C:\Users\user\Documents\file.txt`).
  - `..\Public\file.txt` (указывает на `C:\Users\Public\file.txt`).

Специальные символы:

- `.` — текущая директория.
- `..` — родительская директория (на уровень выше).

Особенности:

- Зависит от текущей рабочей директории.
- Короче и удобнее, если вы работаете в пределах одной директории или проекта.
- Может быть менее надежным, если текущая рабочая директория меняется.

Когда использовать?

- Когда вы работаете в пределах одного проекта или директории.
- В скриптах, где текущая рабочая директория известна и фиксирована.
- Для упрощения путей, если они становятся слишком длинными.

Примеры использования

Пример 1: Абсолютный путь

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

int main() {
    // Открываем файл по абсолютному пути
    FILE *file = fopen("/home/user/documents/file.txt", "r");
    if (file) {
        printf("Файл успешно открыт!\n");
        fclose(file);
    } else {
        perror("Ошибка при открытии файла");
    }
    return 0;
}
```

- В этом примере путь `/home/user/documents/file.txt` всегда указывает на один и тот же файл, независимо от текущей рабочей директории.

Пример 2: Относительный путь

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

int main() {
    // Открываем файл по относительному пути
    FILE *file = fopen("documents/file.txt", "r");
    if (file) {
        printf("Файл успешно открыт!\n");
        fclose(file);
    } else {
        perror("Ошибка при открытии файла");
    }
    return 0;
}
```

- В этом примере путь `documents/file.txt` зависит от текущей рабочей директории. Если текущая директория — `/home/user`, то файл будет открыт по пути `/home/user/documents/file.txt`.

Как узнать текущую рабочую директорию?

В Unix-подобных системах можно использовать функцию `getcwd`:

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

int main() {
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) {
        printf("Текущая рабочая директория: %s\n", cwd);
    } else {
        perror("Ошибка при получении текущей директории");
    }
    return 0;
}
```

Преобразование относительного пути в абсолютный

Иногда полезно преобразовать относительный путь в абсолютный. В Unix-подобных системах можно использовать функцию `realpath`:

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

int main() {
    char *abs_path = realpath("documents/file.txt", NULL);
    if (abs_path) {
        printf("Абсолютный путь: %s\n", abs_path);
```

```
        free(abs_path);
    } else {
        perror("Ошибка при преобразовании пути");
    }
    return 0;
}
```

Вот таблица, которая наглядно показывает различия между **абсолютным** и **относительным** путями:

Характеристика	Абсолютный путь	Относительный путь
Определение	Полный путь к файлу или директории, начиная с корня файловой системы.	Путь к файлу или директории, указанный относительно текущей рабочей директории.
Начинается с	Корня файловой системы (`/` в Unix, `C:\` в Windows).	Текущей директории (`.`), родительской директории (`..`) или имени файла/директории.
Пример (Unix)	`/home/user/documents/file.txt`	`documents/file.txt` или `../other_user/file.txt`
Пример (Windows)	`C:\Users\user\Documents\file.txt`	`Documents\file.txt` или `..\Public\file.txt`
Зависимость от текущей директории	Нет. Всегда указывает на одно и то же место, независимо от текущей директории.	Да. Зависит от текущей рабочей директории.
Удобство	Удобен для точного указания пути, особенно в скриптах и программах.	Удобен для работы в пределах одного проекта или директории.
Надежность	Высокая. Всегда указывает на правильное место.	Меньше, если текущая директория меняется.
Использование	Когда нужна точность и независимость от текущей директории.	Когда работа ведется в пределах одной директории или проекта.
Преобразование в абсолютный	Не требуется.	Можно преобразовать с помощью функций, например, `realpath` в Unix.
Специальные символы	Нет.	Используются `.` (текущая директория) и `..` (родительская директория).

Примеры использования:

Абсолютный путь:

- **Unix**: `/home/user/project/script.sh`
- **Windows**: `C:\Users\user\Project\script.bat`

Относительный путь:

- Если текущая директория — `/home/user` (Unix) или `C:\Users\user` (Windows):
  - `project/script.sh` (Unix) или `Project\script.bat` (Windows).
  - `../other_user/file.txt` (Unix) или `..\Public\file.txt` (Windows).

Когда что использовать?

- **Абсолютный путь**:
  - В скриптах, где важно точно указать расположение файла.
  - Когда текущая рабочая директория может меняться.
  - Для доступа к файлам вне текущего проекта.
- **Относительный путь**:
  - Внутри проекта, где все файлы находятся в одной иерархии.
  - Для упрощения путей и уменьшения их длины.
  - Когда текущая рабочая директория фиксирована.

К заданию!

Шаги реализации:

1. **Определение имени программы:**
   - Используйте `argv[0]` для определения имени программы (жесткой ссылки).
2. **Реализация функций:**
   - Для каждой операции (создание каталога, вывод содержимого каталога, удаление каталога и т.д.) реализуйте соответствующую функцию.
3. **Вызов соответствующей функции:**
   - В зависимости от имени программы вызывайте соответствующую функцию.

Пример кода:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

// Функция для создания каталога
void create_directory(const char *path) {
    if (mkdir(path, 0755) == -1) { // Создаем каталог с правами доступа 0755 (rwxr-xr-x)
        perror("Ошибка при создании каталога");
    } else {
        printf("Каталог '%s' успешно создан.\n", path);
    }
```

```c
}

// Функция для вывода содержимого каталога
void list_directory(const char *path) {
    DIR *dir = opendir(path); // Открываем каталог
    if (dir == NULL) {
        perror("Ошибка при открытии каталога");
        return;
    }

    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) { // Читаем каждый элемент каталога
        printf("%s\n", entry->d_name); // Выводим имя элемента
    }
    closedir(dir);
}

// Функция для удаления каталога
void remove_directory(const char *path) {
    if (rmdir(path) == -1) { // Удаляем каталог
        perror("Ошибка при удалении каталога");
    } else {
        printf("Каталог '%s' успешно удален.\n", path);
    }
}

// Функция для создания файла
void create_file(const char *path) {
    int fd = open(path, O_CREAT | O_WRONLY, 0644); // Создаем файл с правами доступа 0644
(rw-r--r--)
    if (fd == -1) {
        perror("Ошибка при создании файла");
    } else {
        close(fd); // Закрываем файл
        printf("Файл '%s' успешно создан.\n", path);
    }
}

// Функция для вывода содержимого файла
void read_file(const char *path) {
    char buffer[1024];
    int fd = open(path, O_RDONLY); // Открываем файл для чтения
    if (fd == -1) {
        perror("Ошибка при открытии файла"); // Выводим сообщение об ошибке, если файл не
удалось открыть
        return;
    }

    ssize_t bytes_read;
    while ((bytes_read = read(fd, buffer, sizeof(buffer))) { // Читаем файл по частям
        if (bytes_read == -1) {
            perror("Ошибка при чтении файла"); // Выводим сообщение об ошибке, если
чтение не удалось
            break;
        }
        write(STDOUT_FILENO, buffer, bytes_read); // Выводим содержимое файла на экран
    }
```

```
    close(fd); // Закрываем файл
}

// Функция для удаления файла
void remove_file(const char *path) {
    if (unlink(path) == -1) { // Удаляем файл
        perror("Ошибка при удалении файла"); // Выводим сообщение об ошибке, если что-то
пошло не так
    } else {
        printf("Файл '%s' успешно удален.\n", path); // Сообщаем об успешном удалении
файла
    }
}

// Функция для создания символьной ссылки
void create_symlink(const char *target, const char *link_path) {
    if (symlink(target, link_path) == -1) { // Создаем символьную ссылку
        perror("Ошибка при создании символьной ссылки"); // Выводим сообщение об ошибке,
если что-то пошло не так
    } else {
        printf("Символьная ссылка '%s' -> '%s' успешно создана.\n", link_path, target);
// Сообщаем об успешном создании ссылки
    }
}

// Функция для вывода содержимого символьной ссылки
void read_symlink(const char *path) {
    char buffer[1024];
    ssize_t len = readlink(path, buffer, sizeof(buffer) - 1); // Читаем содержимое
символьной ссылки
    if (len == -1) {
        perror("Ошибка при чтении символьной ссылки"); // Выводим сообщение об ошибке,
если чтение не удалось
    } else {
        buffer[len] = '\0'; // Добавляем завершающий нулевой символ
        printf("Символьная ссылка '%s' указывает на: %s\n", path, buffer); // Выводим
содержимое ссылки
    }
}

// Функция для вывода содержимого файла, на который указывает символьная ссылка
void read_symlink_target(const char *path) {
    char buffer[1024];
    ssize_t len = readlink(path, buffer, sizeof(buffer) - 1); // Читаем содержимое
символьной ссылки
    if (len == -1) {
        perror("Ошибка при чтении символьной ссылки"); // Выводим сообщение об ошибке,
если чтение не удалось
        return;
    }

    buffer[len] = '\0'; // Добавляем завершающий нулевой символ
    read_file(buffer); // Выводим содержимое файла, на который указывает ссылка
}

// Функция для удаления символьной ссылки
void remove_symlink(const char *path) {
    if (unlink(path) == -1) { // Удаляем символьную ссылку
```

```
        perror("Ошибка при удалении символьной ссылки"); // Выводим сообщение об ошибке, если что-то пошло не так
    } else {
        printf("Символьная ссылка '%s' успешно удалена.\n", path); // Сообщаем об успешном удалении ссылки
    }
}

// Функция для создания жесткой ссылки
void create_hardlink(const char *target, const char *link_path) {
    if (link(target, link_path) == -1) { // Создаем жесткую ссылку
        perror("Ошибка при создании жесткой ссылки");
    } else {
        printf("Жесткая ссылка '%s' -> '%s' успешно создана.\n", link_path, target);
    }
}

// Функция для удаления жесткой ссылки
void remove_hardlink(const char *path) {
    if (unlink(path) == -1) { // Удаляем жесткую ссылку
        perror("Ошибка при удалении жесткой ссылки"); // Выводим сообщение об ошибке, если что-то пошло не так
    } else {
        printf("Жесткая ссылка '%s' успешно удалена.\n", path); // Сообщаем об успешном удалении ссылки
    }
}

// Функция для вывода прав доступа и количества жестких ссылок
void print_file_info(const char *path) {
    struct stat st;
    if (stat(path, &st) == -1) { // Получаем информацию о файле
        perror("Ошибка при получении информации о файле");
        return;
    }

    printf("Права доступа: %o\n", st.st_mode & 0777); // Выводим права доступа
    printf("Количество жестких ссылок: %lu\n", st.st_nlink); // Выводим количество жестких ссылок
}

// Функция для изменения прав доступа
void change_permissions(const char *path, mode_t mode) {
    if (chmod(path, mode) == -1) { // Изменяем права доступа
        perror("Ошибка при изменении прав доступа");
    } else {
        printf("Права доступа к файлу '%s' успешно изменены.\n", path);
    }
}

// Основная функция
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Использование: %s <аргументы>\n", argv[0]); // Выводим сообщение об ошибке, если аргументов недостаточно
        return 1;
    }
```

```
    // Определяем действие на основе имени программы
    const char *action = strrchr(argv[0], '/'); // Ищем последний символ '/' в имени
программы
    action = action ? action + 1 : argv[0]; // Получаем имя программы без пути

    if (strcmp(action, "create_dir") == 0) {
        create_directory(argv[1]); // Создаем каталог
    } else if (strcmp(action, "list_dir") == 0) {
        list_directory(argv[1]); // Выводим содержимое каталога
    } else if (strcmp(action, "remove_dir") == 0) {
        remove_directory(argv[1]); // Удаляем каталог
    } else if (strcmp(action, "create_file") == 0) {
        create_file(argv[1]); // Создаем файл
    } else if (strcmp(action, "read_file") == 0) {
        read_file(argv[1]); // Читаем файл
    } else if (strcmp(action, "remove_file") == 0) {
        remove_file(argv[1]); // Удаляем файл
    } else if (strcmp(action, "create_symlink") == 0) {
        create_symlink(argv[1], argv[2]); // Создаем символьную ссылку
    } else if (strcmp(action, "read_symlink") == 0) {
        read_symlink(argv[1]); // Читаем символьную ссылку
    } else if (strcmp(action, "read_symlink_target") == 0) {
        read_symlink_target(argv[1]); // Читаем файл, на который указывает символьная
ссылка
    } else if (strcmp(action, "remove_symlink") == 0) {
        remove_symlink(argv[1]); // Удаляем символьную ссылку
    } else if (strcmp(action, "create_hardlink") == 0) {
        create_hardlink(argv[1], argv[2]); // Создаем жесткую ссылку
    } else if (strcmp(action, "remove_hardlink") == 0) {
        remove_hardlink(argv[1]); // Удаляем жесткую ссылку
    } else if (strcmp(action, "print_file_info") == 0) {
        print_file_info(argv[1]); // Выводим информацию о файле
    } else if (strcmp(action, "change_permissions") == 0) {
        change_permissions(argv[1], strtol(argv[2], NULL, 8)); // Изменяем права доступа
    } else {
        fprintf(stderr, "Неизвестное действие: %s\n", action); // Выводим сообщение об
ошибке, если действие неизвестно
        return 1;
    }

    return 0;
}
```

Как использовать:

1. Скомпилируйте программу:

```
    gcc -o program program.c
```

2. Создайте жесткие ссылки для каждого действия:

```
ln program create_dir
   ln program list_dir
   ln program remove_dir
   ln program create_file
   ln program read_file
```

```
    ln program remove_file
    ln program create_symlink
    ln program read_symlink
    ln program read_symlink_target
    ln program remove_symlink
    ln program create_hardlink
    ln program remove_hardlink
    ln program print_file_info
    ln program change_permissions
```

3. Выполняйте действия, используя соответствующие ссылки:

```
./create_dir my_directory
    ./list_dir my_directory
    ./remove_dir my_directory
    ./create_file my_file.txt
    ./read_file my_file.txt
    ./remove_file my_file.txt
    ./create_symlink my_file.txt my_symlink
    ./read_symlink my_symlink
    ./read_symlink_target my_symlink
    ./remove_symlink my_symlink
    ./create_hardlink my_file.txt my_hardlink
    ./remove_hardlink my_hardlink
    ./print_file_info my_file.txt
    ./change_permissions my_file.txt 0644
```

Пояснения:

- **`strrchr(argv[0], '/')`** : Находит последний символ `/` в имени программы, чтобы извлечь только имя файла (без пути).
- **`strcmp(action, "create_dir") == 0`** : Сравнивает имя программы с действием и выполняет соответствующую функцию.
- **`strtol(argv[2], NULL, 8)`** : Преобразует строку с правами доступа (например, `0644`) в число.

А теперь прости господи как это все под капотом выглядит. Все эти функции буквально - системные вызовы

1. `mkdir` — создание каталога

Что делает:

Создает новый каталог с указанным именем и правами доступа.

Как это работает:

1. **Пользовательский уровень**:

- Вызывается функция `mkdir(path, mode)`.
- `path` — путь к новому каталогу.
- `mode` — права доступа (например, `0755`).

2. **Системный вызов**:
   - Функция `mkdir` вызывает системный вызов `mkdir()` в ядре Linux.
3. **Ядро Linux**:
   - Ядро проверяет, есть ли у пользователя права на создание каталога в указанной директории.
   - Создается новый inode для каталога.
   - В родительском каталоге создается запись с именем нового каталога и ссылкой на его inode.
   - Устанавливаются права доступа (`mode`) для нового каталога.
4. **Файловая система**:
   - В зависимости от файловой системы (например, ext4, XFS), ядро обновляет метаданные (например, таблицу inode и блоки данных).

---

2. `closedir` — закрытие каталога

Что делает:

Закрывает открытый каталог, освобождая ресурсы.

Как это работает:

1. **Пользовательский уровень**:
   - Вызывается функция `closedir(DIR *dir)`.
   - `dir` — указатель на структуру `DIR`, представляющую открытый каталог.
2. **Ядро Linux**:
   - Освобождает ресурсы, связанные с открытым каталогом.
   - Закрывает файловый дескриптор, связанный с каталогом.

---

3. `rmdir` — удаление каталога

Что делает:

Удаляет пустой каталог.

Как это работает:

1. **Пользовательский уровень**:
   - Вызывается функция `rmdir(path)`.
   - `path` — путь к каталогу, который нужно удалить.
2. **Системный вызов**:

- Функция `rmdir` вызывает системный вызов `rmdir()` в ядре Linux.

3. **Ядро Linux**:
   - Проверяет, пуст ли каталог (в нем не должно быть файлов или подкаталогов).
   - Удаляет запись о каталоге из родительского каталога.
   - Освобождает inode и блоки данных, связанные с каталогом.

---

4. `unlink` — удаление файла или ссылки

Что делает:

Удаляет жесткую ссылку на файл. Если это последняя ссылка, файл удаляется.

Как это работает:

1. **Пользовательский уровень**:
   - Вызывается функция `unlink(path)`.
   - `path` — путь к файлу или ссылке.
2. **Системный вызов**:
   - Функция `unlink` вызывает системный вызов `unlink()` в ядре Linux.
3. **Ядро Linux**:
   - Уменьшает счетчик ссылок на inode файла.
   - Если счетчик ссылок становится равным нулю, файл удаляется:
     - Освобождаются блоки данных.
     - Освобождается inode.

Функция `unlink` — это системный вызов в Unix-подобных операционных системах (например, Linux, macOS), который используется для удаления файла из файловой системы. Давайте разберем, как она работает, почему она удаляет файл и можно ли её заменить.

Что делает `unlink`?

1. **Удаление файла**:
   - `unlink` удаляет файл, указанный по пути `path`.
   - Если файл успешно удален, функция возвращает `0`.
   - Если произошла ошибка (например, файл не существует или нет прав на удаление), функция возвращает `-1`, и переменная `errno` устанавливается в соответствующее значение ошибки.
2. **Как это работает**:
   - В Unix-подобных системах файлы хранятся на диске как структуры данных, называемые **inode**.
   - Каждый файл может иметь одно или несколько **имён** (жесткие ссылки), которые указывают на его inode.
   - Функция `unlink` удаляет одно из этих имён (ссылку) из файловой системы.

- Если это была последняя ссылка на файл (то есть других жёстких ссылок на этот inode нет), то файл физически удаляется с диска, и его данные освобождаются.

Почему она удаляет файл?

- `unlink` удаляет файл, потому что она убирает связь между именем файла (путь `path`) и его данными на диске (inode).
- Если на файл больше нет ссылок (например, других жёстких ссылок или открытых файловых дескрипторов), операционная система освобождает место, занимаемое файлом.

Давайте разберем, почему функция `unlink` используется для удаления **символьных ссылок** (symlink), хотя она также может удалять **жесткие ссылки** (hard link) и обычные файлы. Функция `unlink` — это низкоуровневый системный вызов в Unix-подобных системах, который выполняет следующие действия:

1. **Удаляет имя файла из файловой системы**:
   - Файловая система в Unix-подобных ОС использует **inode** для хранения метаданных файла (например, размер, права доступа и т.д.).
   - Каждый файл может иметь одно или несколько **имен** (жестких ссылок), которые указывают на его inode.
   - `unlink` удаляет одно из этих имен.
2. **Уменьшает счетчик ссылок на inode**:
   - Каждый inode содержит счетчик, который указывает, сколько имен (жестких ссылок) ссылаются на него.
   - Когда счетчик достигает нуля (то есть больше нет имен, ссылающихся на inode), файл физически удаляется с диска, и его данные освобождаются.

Почему `unlink` работает с символьными ссылками?

Символьная ссылка (symlink) — это **отдельный файл**, который содержит путь к другому файлу или директории. Когда вы создаете символьную ссылку, файловая система создает новый файл, который хранит путь к цели.

- **Символьная ссылка** — это файл, поэтому `unlink` может удалить его, как и любой другой файл.
- Удаление символьной ссылки **не влияет** на целевой файл или директорию, на которую она указывает.

Почему `unlink` работает с жесткими ссылками?

Жесткая ссылка (hard link) — это **дополнительное имя** для существующего файла (inode). Когда вы создаете жесткую ссылку, файловая система просто добавляет еще одно имя, ссылающееся

на тот же inode.

- `unlink` удаляет одно из имен, ссылающихся на inode.
- Если это была последняя ссылка на inode, файл физически удаляется.

---

5. `symlink` — создание символьной ссылки

Что делает:

Создает символьную ссылку (симлинк) на указанный файл или каталог.

Как это работает:

1. **Пользовательский уровень**:
   - Вызывается функция `symlink(target, linkpath)`.
   - `target` — путь к файлу или каталогу, на который ссылается симлинк.
   - `linkpath` — путь, по которому создается симлинк.
2. **Системный вызов**:
   - Функция `symlink` вызывает системный вызов `symlink()` в ядре Linux.
3. **Ядро Linux**:
   - Создает новый файл (симлинк) с указанным именем.
   - Записывает путь `target` в данные симлинка.
   - Создает запись в родительском каталоге для симлинка.
     Конечно! Функция `symlink` используется в Unix-подобных операционных системах (например, Linux, macOS) для создания **символьной ссылки** (symbolic link, или symlink). Символьная ссылка — это специальный тип файла, который ссылается на другой файл или директорию по пути. Давайте разберем, как работает `symlink`, какие аргументы она принимает и что делает.

Что такое символьная ссылка?

- Символьная ссылка (symlink) — это файл, который содержит путь к другому файлу или директории.
- Она похожа на ярлык в Windows: если вы открываете символьную ссылку, вы фактически открываете файл или директорию, на которую она указывает.
- Символьные ссылки могут ссылаться на файлы или директории, даже если они находятся на другом диске или в другой файловой системе.

Функция `symlink`

Функция `symlink` создает символьную ссылку. Её прототип выглядит так:

```c
#include <unistd.h>

int symlink(const char *target, const char *link_path);
```

Аргументы:

1. **`target`**:
   - Это путь к файлу или директории, на которую будет указывать символьная ссылка.
   - Может быть абсолютным (например, `/home/user/file.txt`) или относительным (например, `../file.txt`).
2. **`link_path`**:
   - Это путь, по которому будет создана символьная ссылка.
   - Например, если вы хотите создать ссылку с именем `my_link` в текущей директории, вы можете указать `./my_link`.

Возвращаемое значение:

- Если символьная ссылка успешно создана, функция возвращает `0`.
- Если произошла ошибка, функция возвращает `-1`, и переменная `errno` устанавливается в соответствующее значение ошибки.

Что делает **`symlink`**?

1. Создает новый файл (символьную ссылку) по пути `link_path`.
2. Записывает в этот файл путь `target`, на который должна указывать ссылка.
3. После создания символьной ссылки, при обращении к `link_path`, система автоматически перенаправляет вас к `target`.

Пример использования:

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

void create_symlink(const char *target, const char *link_path) {
    if (symlink(target, link_path) == -1) { // Создаем символьную ссылку
        perror("Ошибка при создании символьной ссылки"); // Выводим сообщение об ошибке
    } else {
        printf("Символьная ссылка '%s' -> '%s' успешно создана.\n", link_path, target);
    }
}

int main() {
    create_symlink("/path/to/target/file.txt", "/path/to/link/file_link.txt");
    return 0;
}
```

Что произойдет:

- Будет создана символьная ссылка `/path/to/link/file_link.txt`, которая указывает на `/path/to/target/file.txt`.
- Если вы откроете `/path/to/link/file_link.txt`, вы фактически откроете `/path/to/target/file.txt`.

Возможные ошибки:

- **`EEXIST`**: Файл или ссылка с именем `link_path` уже существует.
- **`ENOENT`**: Одна из директорий в пути `link_path` не существует.
- **`EPERM`**: Нет прав на создание ссылки в указанной директории.
- **`EACCES`**: Нет прав на доступ к целевой директории.

Чем символьная ссылка отличается от жесткой ссылки?

1. **Символьная ссылка (symlink)**:
   - Это отдельный файл, который содержит путь к другому файлу или директории.
   - Может ссылаться на файлы и директории, даже если они находятся на другом диске или в другой файловой системе.
   - Если целевой файл удален, символьная ссылка становится "битой" (dangling link).
2. **Жесткая ссылка (hard link)**:
   - Это дополнительное имя для существующего файла (inode).
   - Жесткие ссылки могут ссылаться только на файлы в той же файловой системе.
   - Если оригинальный файл удален, данные остаются доступны через жесткую ссылку, пока есть хотя бы одна ссылка на inode.

Можно ли заменить `symlink`?

Да, но только с помощью других инструментов или системных вызовов. Например:

1. **Использование команды `ln -s` через `system`**:

```
#include <stdlib.h>
#include <stdio.h>

void create_symlink(const char *target, const char *link_path) {
    char command[256];
    snprintf(command, sizeof(command), "ln -s %s %s", target, link_path); //
Формируем команду
    if (system(command) == -1) { // Выполняем команду
        perror("Ошибка при создании символьной ссылки");
    } else {
        printf("Символьная ссылка '%s' -> '%s' успешно создана.\n", link_path,
```

```
target);
    }
}
```

2. **Использование библиотек высокого уровня** (например, Python, Bash):
   - В Python: `os.symlink(target, link_path)`.
   - В Bash: `ln -s target link_path`.

---

6. `readlink` — чтение содержимого символьной ссылки

Что делает:

Читает путь, на который указывает символьная ссылка.

Как это работает:

1. **Пользовательский уровень**:
   - Вызывается функция `readlink(path, buffer, bufsize)`.
   - `path` — путь к симлинку.
   - `buffer` — буфер для хранения результата.
   - `bufsize` — размер буфера.
2. **Системный вызов**:
   - Функция `readlink` вызывает системный вызов `readlink()` в ядре Linux.
3. **Ядро Linux**:
   - Читает данные симлинка (путь, на который он указывает).
   - Копирует эти данные в буфер пользователя.

---

7. `link` — создание жесткой ссылки

Что делает:

Создает жесткую ссылку на существующий файл.

Как это работает:

1. **Пользовательский уровень**:
   - Вызывается функция `link(target, linkpath)`.
   - `target` — путь к существующему файлу.
   - `linkpath` — путь, по которому создается жесткая ссылка.
2. **Системный вызов**:
   - Функция `link` вызывает системный вызов `link()` в ядре Linux.
3. **Ядро Linux**:
   - Увеличивает счетчик ссылок на inode файла.
   - Создает новую запись в каталоге для жесткой ссылки, указывающую на тот же inode.

8. `stat` — получение информации о файле

Что делает:

Функция `stat` возвращает информацию о файле, такую как:

- Тип файла (обычный файл, директория, символьная ссылка и т.д.).
- Права доступа.
- Размер файла.
- Время последнего доступа, изменения и модификации.
- Количество жестких ссылок.
- Идентификатор владельца и группы.

Прототип функции:

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

int stat(const char *path, struct stat *st);
```

- **`path`**: Путь к файлу, информацию о котором нужно получить.
- **`st`**: Указатель на структуру `struct stat`, куда будет записана информация о файле.
- **Возвращаемое значение**:
  - `0` в случае успеха.
  - `-1` в случае ошибки, и переменная `errno` устанавливается в соответствующее значение.

---

2. Структура `struct stat`

Структура `struct stat` определена в заголовочном файле `<sys/stat.h>`. Она содержит поля, которые описывают различные атрибуты файла. Вот основные поля:

```
struct stat {
    dev_t     st_dev;         // ID устройства, содержащего файл
    ino_t     st_ino;         // Номер inode
    mode_t    st_mode;        // Тип файла и права доступа
    nlink_t   st_nlink;       // Количество жестких ссылок
    uid_t     st_uid;         // ID пользователя-владельца
    gid_t     st_gid;         // ID группы-владельца
    dev_t     st_rdev;        // ID устройства (если это специальный файл)
    off_t     st_size;        // Размер файла в байтах
    blksize_t st_blksize;     // Размер блока ввода-вывода
    blkcnt_t  st_blocks;      // Количество выделенных блоков
    time_t    st_atime;       // Время последнего доступа
    time_t    st_mtime;       // Время последней модификации
    time_t    st_ctime;       // Время последнего изменения статуса
};
```

Основные поля:

- `st_mode`:
  - Содержит информацию о типе файла и правах доступа.
  - Тип файла можно определить с помощью макросов:
    - `S_ISREG(mode)`: Обычный файл.
    - `S_ISDIR(mode)`: Директория.
    - `S_ISLNK(mode)`: Символьная ссылка.
    - `S_ISCHR(mode)`: Символьное устройство.
    - `S_ISBLK(mode)`: Блочное устройство.
    - `S_ISFIFO(mode)`: FIFO (именованный канал).
    - `S_ISSOCK(mode)`: Сокет.
  - Права доступа можно проверить с помощью макросов, например:
    - `S_IRUSR`: Права на чтение для владельца.
    - `S_IWUSR`: Права на запись для владельца.
    - `S_IXUSR`: Права на выполнение для владельца.
- `st_size`:
  - Размер файла в байтах.
- `st_nlink`:
  - Количество жестких ссылок на файл.
- `st_atime`, `st_mtime`, `st_ctime`:
  - Время последнего доступа, модификации и изменения статуса файла.

---

3. Как работает `stat`?

Пользовательский уровень:

1. Вы вызываете функцию `stat(path, &st)`.
2. Функция `stat` передает управление ядру через системный вызов.

Системный вызов:

- В Linux системный вызов `stat()` имеет номер **4** (на x86-64 архитектуре).
- Системный вызов `stat()` выполняет следующие действия:
  1. Находит inode файла по указанному пути.
  2. Заполняет структуру `struct stat` данными из inode.
  3. Возвращает результат в пользовательское пространство.

Ядро Linux:

1. Ядро ищет файл по пути `path`.
2. Если файл найден, ядро заполняет структуру `struct stat` данными из inode файла.
3. Если файл не найден или произошла ошибка, ядро возвращает ошибку.

---

4. Пример использования `stat`

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

void print_file_info(const char *path) {
    struct stat st;
    if (stat(path, &st) == -1) {
        perror("Ошибка при получении информации о файле");
        return;
    }

    printf("Информация о файле: %s\n", path);
    printf("  Размер: %ld байт\n", st.st_size);
    printf("  Количество ссылок: %ld\n", st.st_nlink);
    printf("  Владелец: %d\n", st.st_uid);
    printf("  Группа: %d\n", st.st_gid);
    printf("  Последний доступ: %s", ctime(&st.st_atime));
    printf("  Последняя модификация: %s", ctime(&st.st_mtime));
    printf("  Последнее изменение статуса: %s", ctime(&st.st_ctime));

    if (S_ISREG(st.st_mode)) {
        printf("  Тип: Обычный файл\n");
    } else if (S_ISDIR(st.st_mode)) {
        printf("  Тип: Директория\n");
    } else if (S_ISLNK(st.st_mode)) {
        printf("  Тип: Символьная ссылка\n");
    }
}

int main() {
    print_file_info("example.txt");
    return 0;
}
```

Вывод:

```
Информация о файле: example.txt
  Размер: 1024 байт
  Количество ссылок: 1
  Владелец: 1000
  Группа: 1000
  Последний доступ: Mon Mar 20 12:34:56 2023
  Последняя модификация: Mon Mar 20 12:34:56 2023
  Последнее изменение статуса: Mon Mar 20 12:34:56 2023
  Тип: Обычный файл
```

---

5. Системный вызов `stat`

- **Номер системного вызова**: На архитектуре x86-64 системный вызов `stat()` имеет номер **4**.
- **Альтернативные функции**:
  - `lstat()`: Аналогична `stat()`, но для символьных ссылок возвращает информацию о самой ссылке, а не о файле, на который она указывает.
  - `fstat()`: Получает информацию о файле по его файловому дескриптору.

---

9. `chmod` — изменение прав доступа к файлу

Конечно! Давайте подробно разберем, как работает функция `chmod`, как она изменяет права доступа к файлу и как это связано с системными вызовами в Linux.

---

1. Функция `chmod`

Что делает?

Функция `chmod` изменяет права доступа к файлу или директории. Права доступа определяют, кто может читать, записывать или выполнять файл.

Прототип функции:

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

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

- **`path`**: Путь к файлу или директории, права доступа к которым нужно изменить.
- **`mode`**: Новые права доступа, заданные в виде битовой маски. Обычно указываются в восьмеричном формате (например, `0644`).
- **Возвращаемое значение**:
  - `0` в случае успеха.
  - `-1` в случае ошибки, и переменная `errno` устанавливается в соответствующее значение.

---

2. Права доступа (`mode`)

Права доступа задаются в виде битовой маски. Они состоят из трех групп:

1. **Права для владельца (user)**:
   - `S_IRUSR` (0400): Право на чтение.
   - `S_IWUSR` (0200): Право на запись.
   - `S_IXUSR` (0100): Право на выполнение.
2. **Права для группы (group)**:
   - `S_IRGRP` (0040): Право на чтение.
   - `S_IWGRP` (0020): Право на запись.

- `S_IXGRP` (0010): Право на выполнение.

3. **Права для остальных (others)**:
   - `S_IROTH` (0004): Право на чтение.
   - `S_IWOTH` (0002): Право на запись.
   - `S_IXOTH` (0001): Право на выполнение.

Примеры:

- `0644`: Владелец может читать и записывать, группа и остальные — только читать.
- `0755`: Владелец может читать, записывать и выполнять, группа и остальные — читать и выполнять.

---

3. Как работает `chmod`?

Пользовательский уровень:

1. Вы вызываете функцию `chmod(path, mode)`.
2. Функция `chmod` передает управление ядру через системный вызов.

Системный вызов:

- В Linux системный вызов `chmod()` имеет номер **90** (на x86-64 архитектуре).
- Системный вызов `chmod()` выполняет следующие действия:
  1. Находит inode файла по указанному пути.
  2. Обновляет поле `st_mode` в inode, устанавливая новые права доступа.
  3. Возвращает результат в пользовательское пространство.

Ядро Linux:

1. Ядро ищет файл по пути `path`.
2. Если файл найден, ядро обновляет поле `st_mode` в inode файла.
3. Если файл не найден или произошла ошибка, ядро возвращает ошибку.

---

4. Пример использования `chmod`

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

int main() {
    const char *path = "example.txt";
    mode_t mode = 0644; // Новые права доступа: rw-r--r--

    if (chmod(path, mode) == -1) {
        perror("Ошибка при изменении прав доступа");
        return 1;
    }
```

```
    printf("Права доступа к файлу '%s' успешно изменены.\n", path);
    return 0;
}
```

Вывод:

```
Права доступа к файлу 'example.txt' успешно изменены.
```

---

5. Системный вызов `chmod`

- **Номер системного вызова**: На архитектуре x86-64 системный вызов `chmod()` имеет номер **90**.
- **Альтернативные функции**:
  - `fchmod()`: Аналогична `chmod()`, но принимает файловый дескриптор вместо пути.
  - `fchmodat()`: Расширенная версия `chmod()`, которая позволяет указать директорию для относительных путей.

---

6. Как права доступа хранятся в inode?

Права доступа хранятся в поле `st_mode` структуры `struct stat`. Это поле содержит:

- Тип файла (обычный файл, директория и т.д.).
- Права доступа для владельца, группы и остальных.

Пример:

Если `st_mode` равно `100644` (в восьмеричном формате `0100644`), это означает:

- Тип файла: Обычный файл (`0100000`).
- Права доступа: `rw-r--r--` (`0644`).

---

7. Проверка прав доступа

После изменения прав доступа вы можете проверить их с помощью функции `stat`:

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

void print_permissions(const char *path) {
    struct stat st;
    if (stat(path, &st) == -1) {
        perror("Ошибка при получении информации о файле");
```

```
        return;
    }

    printf("Права доступа к файлу '%s': %o\n", path, st.st_mode & 0777);
}

int main() {
    const char *path = "example.txt";
    print_permissions(path);
    return 0;
}
```

Вывод:

```
Права доступа к файлу 'example.txt': 644
```

- `STDOUT_FILENO` — это целочисленный файловый дескриптор, который соответствует стандартному потоку вывода (stdout). В Unix-подобных системах это значение обычно равно `1`.
- Стандартный вывод (stdout) — это поток, который по умолчанию связан с терминалом, куда выводятся данные программы (например, текст, который вы видите в консоли).

---

3. Программа для вывода содержимого `/proc/pid/pagemap`

Написать программу, которая выводит содержимое /proc/pid/pagemap

Общее описание:

Программа должна выводить содержимое файла `/proc/pid/pagemap`, который содержит информацию о страницах памяти процесса.

Шаги реализации:

1. **Получение PID:**
   - Программа должна принимать PID процесса в качестве аргумента командной строки.
2. **Открытие файла `/proc/pid/pagemap`:**
   - Используйте `open` для открытия файла `/proc/pid/pagemap`.
3. **Чтение и вывод содержимого:**
   - Используйте `read` для чтения содержимого файла и выводите его в шестнадцатеричном формате.

Пример кода:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
```

```c
#include <unistd.h>
#include <inttypes.h>

void print_pagemap(const char *pagemap_path) {
    int fd = open(pagemap_path, O_RDONLY);
    if (fd == -1) {
        perror("open");
        return;
    }

    uint64_t entry;
    ssize_t bytes_read;

    while ((bytes_read = read(fd, &entry, sizeof(entry))) > 0) {
        printf("%016" PRIx64 "\n", entry);
    }

    if (bytes_read == -1) {
        perror("read");
    }

    close(fd);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pid>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    char pagemap_path[256];
    snprintf(pagemap_path, sizeof(pagemap_path), "/proc/%s/pagemap", argv[1]);

    print_pagemap(pagemap_path);

    return 0;
}
```

Заключение

В этом ответе представлены три программы на языке C, которые выполняют различные операции с файлами, каталогами и ссылками, а также выводят содержимое `/proc/pid/pagemap`. Каждая программа подробно описана, и приведены примеры кода для их реализации.