3 Задача.

В файлике **могут** быть ошибки, может немного поехала нумерация :)
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.

```c
#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:
  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` — символическая 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:**

```
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:

```
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:**

```
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.

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);
    }
}

// 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.

3. Программа для вывода содержимого `/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:

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