Задание 4
Сетевая файловая система

Ядро Linux — монолитное (https://en.wikipedia.org/wiki/Monolithic_kernel). Это означает, что все его части работают в общем адресном пространстве. Однако, это не означает, что для добавления какой-то возможности необходимо полностью перекомпилировать ядро. Новую функциональность можно добавить в виде *модуля ядра*.
Такие модули можно легко загружать и выгружать по необходимости прямо во время работы системы.

С помощью модулей можно реализовать свои файловые системы, причём со стороны пользователя такая файловая система ничем не будет отличаться от ext4 (https://en.wikipedia.org/wiki/Ext4) или NTFS (https://en.wikipedia.org/wiki/NTFS). В этом задании мы с Вами реализуем упрощённый аналог NFS (https://en.wikipedia.org/wiki/Network_File_System): все файлы будут храниться на удалённом сервере, однако пользователь сможет пользоваться ими точно так же, как и файлами на собственном жёстком диске.

Мы рекомендуем при выполнении этого домашнего задания использовать отдельную виртуальную машину: любая ошибка может вывести всю систему из строя, и вы можете потерять ваши данные.

Для выполнения лабораторной работы понадобится 2 виртуальных машины, объединенных в сеть. На одну из виртуальных машин нужно будет установить nfs-server.

Мы проверили работоспособность всех инструкций для дистрибутива Ubuntu 22.04 x64 (https://releases.ubuntu.com/22.04/) и ядра версии 5.15.0-53. Возможно, при использовании других дистрибутивов, вы столкнётесь с различными ошибками и особенностями, с которыми вам придётся разобраться самостоятельно.

Часть 1. Сервер файловой системы

Для получения токенов и тестирования вы можете воспользоваться консольной утилитой curl (https://curl.se).

Сервер поддерживает два типа ответов:

- Бинарные данные: набор байт (`char*`), который можно скастить в структуру, указанную в описании ответа. Учтите, что первое поле ответа (первые 8 байт) — код ошибки.
- JSON-объект (https://www.json.org/json-en.html): человекочитаемый ответ. Для его получения необходимо передавать GET-параметр `json`.

Формат JSON предлагается использовать только для отладки, поскольку текущая реализация функции `networkfs_http_call` работает только с бинарным форматом. Однако, вы можете её доработать и реализовать собственный JSON-парсер.

Поскольку в ядре используются не совсем привычные функции для работы с сетью, мы реализовали для вас собственный HTTP-клиент в виде функции `networkfs_http_call` (`http.c:120` (http.c#L120)):

```
int64_t networkfs_http_call(
const char *token,
const char *method,
char *response_buffer,
size_t buffer_size,
size_t arg_size,
...
);
```

- `const char *token` — ваш токен
- `const char *method` — название метода без неймспейса fs (list, create, …)
- `char *response_buffer` — буфер для сохранения ответа от сервера
- `size_t buffer_size` — размер буфера
- `size_t arg_size` — количество аргументов
- далее должны следовать 2 × `arg_size` аргументов типа `const char*` — пары `param1, value1, param2, value2`, … — параметры запроса

Функция возвращает 0, если запрос завершён успешно; положительное число — код ошибки из документации API, если сервер вернул ошибку; отрицательное число — код ошибки из `http.h` (http.h#L6) или `errno-base.h` (`ENOMEM, ENOSPC`) в случае ошибки при выполнении запроса (отсутствие подключения, сбой в сети, некорректный ответ сервера, …).

Часть 2. Знакомство с простым модулем

Давайте научимся компилировать и подключать тривиальный модуль. Для компиляции модулей ядра нам понадобятся утилиты для сборки и заголовочные файлы.
Установить их можно так:

```
$ sudo apt-get install build-essential linux-headers-`uname -r`
```

Мы уже подготовили основу для вашего будущего модуля в файлах `entrypoint.c` (entrypoint.c) и `Makefile` (Makefile). Познакомьтесь с ней.

Ядру для работы с модулем достаточно двух функций — одна должна инициализировать модуль, а вторая — очищать результаты его работы. Они указываются с помощью `module_init` и `module_exit`.

Важное отличие кода для ядра Linux от user-space-кода — в отсутствии в нём стандартной библиотеки `libc`. Например, в ней же находится функция printf. Мы можем печатать данные в системный лог с помощью функции `printk` (https://www.kernel.org/doc/html/latest/core-api/printk-basics.html).

В `Makefile` указано, что наш модуль `networkfs` состоит из двух единиц трансляции — `entrypoint` и `http`. Вы можете самостоятельно добавлять новые единицы, чтобы декомпозировать ваш код удобным образом.
Соберём модуль:
Установить их можно так:

```
$ sudo make
```

Если наш код скомпилировался успешно, в текущей директории появится файл `networkfs.ko` — это и есть наш модуль.
Осталось загрузить его в ядро:

```
$ sudo insmod networkfs.ko
```

Однако, мы не увидели нашего сообщения. Оно печатается не в терминал, а в системный лог — его можно увидеть командой `dmesg`:

```
$ dmesg
<...>
[ 123.456789] Hello, World!
```

Для выгрузки модуля нам понадобится команда `rmmod`:

```
$ sudo rmmod networkfs
$ dmesg
<...>
[ 123.987654] Goodbye!
```

Часть 3. Подготовка файловой системы

Операционная система предоставляет две функции для управления файловыми системами:

- `register_filesystem`(https://www.kernel.org/doc/htmldocs/filesystems/API-register-filesystem.html) — сообщает о появлении нового драйвера файловой системы
- `unregister_filesystem`(https://www.kernel.org/doc/htmldocs/filesystems/API-unregister-filesystem.html) — удаляет драйвер файловой системы

В этой части мы начнём работать с несколькими структурами ядра:

- `inode` (https://elixir.bootlin.com/linux/v5.15.53/source/include/linux/fs.h#L624) — описание метаданных файла: имя файла, расположение, тип файла (в нашем случае — регулярный файл или директория)
- `dentry` (https://elixir.bootlin.com/linux/v5.15.53/source/include/linux/dcache.h#L91) — описание директории: список inode внутри неё, информация о родительской директории, …
- `super_block`(https://elixir.bootlin.com/linux/v5.15.53/source/include/linux/fs.h#L146) — описание всей файловой системы: информация о корневой директории, …

Функции `register_filesystem` и `unregister_filesystem` принимают структуру с описанием файловой системы. Начнём с такой:

```
struct file_system_type networkfs_fs_type =
{
.name = "networkfs",
.mount = networkfs_mount,
.kill_sb = networkfs_kill_sb
};
```

Для монтирования файловой системы в этой структуре мы добавили два поля. Первое — `mount` — указатель на функцию, которая вызывается при монтировании.
Например, она может выглядеть так:

```
struct dentry* networkfs_mount(struct file_system_type
*fs_type, int flags, const char *token, void *data)
{
struct dentry *ret;
ret = mount_nodev(fs_type, flags, data, networkfs_fill_super);
if (ret == NULL)
{
printk(KERN_ERR "Can't mount file system");
}
else
{
printk(KERN_INFO "Mounted successfuly");
}
return ret;
}
```

Эта функция будет вызываться всякий раз, когда пользователь будет монтировать нашу файловую систему. Например, он может это сделать следующей командой (документация (https://linux.die.net/man/8/mount)):

```
$ sudo mount -t networkfs <token> <path>
```

Опция `-t` нужна для указания имени файловой системы — именно оно указывается в поле `name`. Также мы передаём токен, полученный в прошлой части, и локальную директорию, в которую ФС будет примонтирована. Обратите внимание, что эта директория должна быть пуста.

Мы используем функцию `mount_nodev` (https://elixir.bootlin.com/linux/v5.15.53/source/include/linux/fs.h#L2476), поскольку наша файловая система не хранится на каком-либо физическом устройстве:

```
struct dentry* mount_nodev(struct file_system_type *fs_type,
int flags, void *data, int (*fill_super)(struct super_block *,
void *, int));
```

Последний её аргумент — указатель на функцию `fill_super`. Эта функция должна заполнять структуру `super_block` информацией о файловой системе. Давайте начнём с такой функции:

```
int networkfs_fill_super(struct super_block *sb, void *data,
int silent)
{
struct inode *inode;
inode = networkfs_get_inode(sb, NULL, S_IFDIR, 1000);
sb->s_root = d_make_root(inode);
if (sb->s_root == NULL)
{
return -ENOMEM;
}
printk(KERN_INFO "return 0\n");
return 0;
}
```

Аргументы `data` и `silent` нам не понадобятся. В этой функции мы используем ещё одну (пока) неизвестную функцию — `networkfs_get_inode`. Она будет создавать новую структуру `inode`, в нашем случае — для корня файловой системы:

```
struct inode *networkfs_get_inode(struct super_block *sb, const
struct inode *dir, umode_t mode, int i_ino)
{
struct inode *inode;
inode = new_inode(sb);
inode->i_ino = i_ino;
if (inode != NULL)
{
inode_init_owner(inode, dir, mode);
}
return inode;
}
```

Давайте поймём, что эта функция делает. Файловой системе нужно знать, где находится корень файловой системы. Для этого в поле `s_root` мы записываем результат функции `d_make_root` (https://elixir.bootlin.com/linux/v5.15.53/source/fs/dcache.c#L2038), передавая ему корневую `inode`. На сервере корневая директория всегда имеет номер 1000.

Для создания новой `inode` используем функцию `new_inode` (https://elixir.bootlin.com/linux/v5.15.53/source/fs/inode.c#L961). Кроме этого, с помощью функции `inode_init_owner` (https://elixir.bootlin.com/linux/v5.15.53/source/fs/inode.c#L2159) зададим тип ноды — укажем, что это директория.

На самом деле, `umode_t` содержит битовую маску, все значения которой доступны в заголовочном файле `linux/stat.h` (https://elixir.bootlin.com/linux/v5.15.53/source/include/uapi/linux/stat.h#L9) — она задаёт тип объекта и права доступа.

Второе поле, которое мы определили в `file_system_type` — поле `kill_sb` — указатель на функцию, которая вызывается при отмонтировании файловой системы. В нашем случае ничего делать не нужно:

```
void networkfs_kill_sb(struct super_block *sb)
{
printk(KERN_INFO "networkfs super block is destroyed. Unmount
successfully.\n");
}
```

Не забудьте зарегистрировать файловую систему в функции инициализации модуля, и удалять её при очистке модуля. Наконец, соберём и примонтируем нашу файловую систему:

```
$ sudo make
$ sudo insmod networkfs.ko
$ sudo mount -t networkfs 8c6a65c8-5ca6-49d7-a33d-daec00267011
/mnt/ct
```

Если вы всё правильно сделали, ошибок возникнуть не должно. Тем не менее, перейти в директорию `/mnt/ct` не выйдет — ведь мы ещё не реализовали никаких функций для навигации по ФС.

Теперь отмонтируем файловую систему:

```
$ sudo umount /mnt/ct
```

Часть 4. Вывод файлов и директорий

В прошлой части мы закончили на том, что не смогли перейти в директорию:

```
$ sudo mount -t networkfs 8c6a65c8-5ca6-49d7-a33d-daec00267011
/mnt/ct
$ cd /mnt/ct
-bash: cd: /mnt/ct: Not a directory
```

Чтобы это исправить, необходимо реализовать некоторые методы для работы с `inode`.

Чтобы эти методы вызывались, в поле `i_op` нужной нам ноды необходимо записать структуру `inode_operations` (https://elixir.bootlin.com/linux/v5.15.53/source/include/linux/fs.h#L2037). Например, такую:

```
struct inode_operations networkfs_inode_ops =
{
.lookup = networkfs_lookup,
};
```

Первая функция, которую мы реализуем — `lookup`. Именно она позволяет операционной системе определять, что за сущность описывается данной нодой.

Сигнатура функции должна быть такой:

```
struct dentry*
networkfs_lookup(struct inode *parent_inode, struct dentry
*child_dentry, unsigned int flag);
```

- `parent_inode` — родительская нода
- `child_dentry` — объект, к которому мы пытаемся получить доступ
- `flag` — неиспользуемое значение

Пока ничего не будем делать: просто вернём `NULL`. Если мы заново попробуем повторить переход в директорию, у нас ничего не получится — но уже по другой причине:

```
$ cd /mnt/ct
-bash: cd: /mnt/ct: Permission denied
```

Решите эту проблему. Пока сложной системы прав у нас не будет — у всех объектов в файловой системе могут быть права *777*. В итоге должно получиться что-то такое:

```
$ ls -l /mnt/
total 0
drwxrwxrwx 1 root root 0 Oct 24 15:52 ct
```

После этого мы сможем перейти в `/mnt/ct`, но не можем вывести содержимое директории. На этот раз нам понадобится не `i_op`, а `i_fop` — структура типа `file_operations`(https://elixir.bootlin.com/linux/v5.15.53/source/include/linux/fs.h#L1995). Реализуем в ней первую функцию — `iterate`.

```
struct file_operations networkfs_dir_ops =
{
.iterate = networkfs_iterate,
};
```

Эта функция вызывается только для директорий и выводит список объектов в ней (нерекурсивно): для каждого объекта вызывается функция `dir_emit`, в которую передаётся имя объекта, номер ноды и его тип.

Пример функции `networkfs_iterate` приведён ниже:

```
int networkfs_iterate(struct file *filp, struct dir_context
*ctx)
{
char fsname[10];
struct dentry *dentry;
struct inode *inode;
unsigned long offset;
int stored;
unsigned char ftype;
ino_t ino;
ino_t dino;
dentry = filp->f_path.dentry;
inode = dentry->d_inode;
offset = filp->f_pos;
stored = 0;
ino = inode->i_ino;
while (true)
{
if (ino == 100)
{
if (offset == 0)
{
strcpy(fsname, ".");
ftype = DT_DIR;
dino = ino;
}
else if (offset == 1)
{
strcpy(fsname, "..");
ftype = DT_DIR;
dino = dentry->d_parent->d_inode->i_ino;
}
else if (offset == 2)
{
strcpy(fsname, "test.txt");
ftype = DT_REG;
dino = 101;
}
else
{
return stored;
}
}
```

Попробуем снова получить список файлов:

```
$ ls /mnt/ct
ls: cannot access '/mnt/ct/test.txt': No such file or directory
test.txt
```

Эта ошибка возникла из-за того, что `lookup` работает только для корневой директории — но не для файла `test.txt`. Это мы исправим в следующих частях.
Вам осталось реализовать `iterate` для корневой директории с запросом к серверу.

Часть 5. Навигация по директориям

Теперь мы хотим научиться переходить по директориям. На этом шаге функцию `networkfs_lookup` придётся немного расширить: если такой файл есть, нужно вызывать функцию `d_add`, передавая ноду файла. Например, так:

```
struct dentry *networkfs_lookup(struct inode *parent_inode, struct dentry
*child_dentry, unsigned int flag)
{
ino_t root;
struct inode *inode;
const char *name = child_dentry->d_name.name;
root = parent_inode->i_ino;
if (root == 100 && !strcmp(name, "test.txt"))
{
inode = networkfs_get_inode(parent_inode->i_sb, NULL, S_IFREG, 101);
d_add(child_dentry, inode);
}
else if (root == 100 && !strcmp(name, "dir"))
{
inode = networkfs_get_inode(parent_inode->i_sb, NULL, S_IFDIR, 200);
d_add(child_dentry, inode);
}
return NULL;
}
```

Реализуйте навигацию по файлам и директориям, используя данные с сервера.

Часть 6. Создание и удаление файлов

Теперь научимся создавать и удалять файлы. Добавим ещё два поля в `inode_operations` — `create` и `unlink`:
Функция `networkfs_create` вызывается при создании файла и должна возвращать новую inode с помощью `d_add`, если создать файл получилось. Рассмотрим простой пример:

```
int networkfs_create(struct inode *parent_inode, struct dentry
*child_dentry, umode_t mode, bool b)
{
ino_t root;
struct inode *inode;
const char *name = child_dentry->d_name.name;
root = parent_inode->i_ino;
if (root == 100 && !strcmp(name, "test.txt"))
{
inode = networkfs_get_inode(parent_inode->i_sb, NULL, S_IFREG |
S_IRWXUGO, 101);
inode->i_op = &networkfs_inode_ops;
inode->i_fop = NULL;
d_add(child_dentry, inode);
mask |= 1;
}
else if (root == 100 && !strcmp(name, "new_file.txt"))
{
inode = networkfs_get_inode(parent_inode->i_sb, NULL, S_IFREG |
S_IRWXUGO, 102);
inode->i_op = &networkfs_inode_ops;
inode->i_fop = NULL;
d_add(child_dentry, inode);
mask |= 2;
}
return 0;
}
```

Чтобы проверить, как создаются файлы, воспользуемся утилитой `touch`:

```
$ touch test.txt
$ ls
test.txt
$ touch new_file.txt
$ ls
test.txt new_file.txt
$
```

Для удаления файлов определим ещё одну функцию — `networkfs_unlink`.

```
int networkfs_unlink(struct inode *parent_inode, struct dentry
*child_dentry)
{
const char *name = child_dentry->d_name.name;
ino_t root;
root = parent_inode->i_ino;
if (root == 100 && !strcmp(name, "test.txt"))
{
mask &= ~1;
}
else if (root == 100 && !strcmp(name, "new_file.txt"))
{
mask &= ~2;
}
return 0;
}
```

Теперь у нас получится выполнять и команду `rm`.

```
$ ls
test.txt new_file.txt
$ rm test.txt
$ ls
new_file.txt
$ rm new_file.txt
$ ls
$
```

Обратите внимание, что утилита touch проверяет существование файла: для этого вызывается функция lookup.

Часть 7. Создание и удаление директорий

Следующая часть нашего задания — создание и удаление директорий. Добавим в `inode_operations` ещё два поля — `mkdir` и `rmdir`. Их сигнатуры можно найти тут (https://elixir.bootlin.com/linux/v5.15.53/source/include/linux/fs.h#L2051).

Если мы всё сделали правильно, теперь мы сможем запустить тесты.
Для запуска тестов вам понадобится `Python 3` и библиотека `requests`. Запустите тесты и проверьте, что ваше решение работает:

```
$ sudo make tests
<...>
Ran 12 tests in 32.323s
OK
```

Часть 8*. Произвольные имена файлов

Реализуйте возможность создания файлов и директорий, состоящих из любых печатных символов, кроме символа / и '. Пример команды, которая можно будет исполнить:

```
$ touch '!@#$%^&*()-+ '
$ ls
'!@#$%^&*()-+ "
```

Если вы всё сделали правильно, пройдёт тестовый набор `bonus-name`:

```
$ sudo make bonus-name
<...>
Ran 3 tests in 1.814s
OK
```

Часть 9*. Чтение и запись в файлы

Реализуйте чтение из файлов и запись в файлы. Для этого вам понадобится структура `file_operations` не только для директорий, но и для обычных файлов.
В неё вам понадобится добавить два поля — `read` (https://elixir.bootlin.com/linux/v5.15.53/source/include/linux/fs.h#L1998) и `write` (https://elixir.bootlin.com/linux/v5.15.53/source/include/linux/fs.h#L1999).

Соответствующие функции имеют следующие сигнатуры:

```
ssize_t networkfs_read(struct file *filp, char *buffer, size_t
len, loff_t *offset);
ssize_t networkfs_write(struct file *filp, const char *buffer,
size_t len, loff_t *offset);
```

Аргументы такие:

- `filp` — файловый дескриптор
- `buffer` — буфер в user-space для чтения и записи соответственно
- `len` — длина данных для записи
- `offset` — смещение

Обратите внимание, что просто так обратиться в `buffer` нельзя, поскольку он находится в user-space.
Используйте специальные функции (https://www.kernel.org/doc/htmldocs/kernel-hacking/routines-copy.html) для чтения и записи.
В результате вы сможете сделать вот так:

```
$ cat file1
hello world from file1
$ cat file2
file2 content here
$ echo "test" > file1
$ cat file1
test
$
```

Обратите внимание, что файл должен уметь содержать любые ASCII-символы с кодами от 0 до 127 включительно

Если вы всё сделали правильно, пройдёт тестовый набор `bonus-wr`:

```
$ sudo make bonus-wr
<...>
Ran 5 tests in 1.953s
OK
```

Часть 10*. Жёсткие ссылки

Вам необходимо поддержать возможность сослаться из разных мест файловой системы на одну и ту же `inode`.

Обратите внимание: сервер поддерживает жёсткие ссылки только для регулярных файлов, но не для директорий.

Для этого добавьте поле `link` в структуру `inode_operations`. Сигнатура соответствующей функции выглядит так:

```
int networkfs_link(struct dentry *old_dentry, struct inode
*parent_dir, struct dentry *new_dentry);
```

После реализации функции вы сможете выполнить следующие команды:

```
$ ln file1 file3
$ cat file1
hello world from file1
$ cat file3
hello world from file1
$ echo "test" > file1
$ rm file1
$ cat file3
test
$
```

Если вы всё сделали правильно, пройдёт тестовый набор `bonus-link`:

```
$ sudo make bonus-link
<...>
Ran 2 tests in 1.535s
OK
```

Требования к сдаче ЛР преподавателю:

- Наличие отчета, который включает в себя ссылку на репозиторий, вывод о проделанной работе
- Готовность запустить тесты по просьбе преподавателя