ЛР. Файловая система

Ядро ОС `Linux` — монолитное. 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.
Подробнее смотри глава 2.10 в книге «Операционные системы: внутренняя структура и принципы проектирования», Вильям Столлингс.

С помощью модулей можно реализовать свои файловые системы, причём со стороны пользователя такая файловая система ничем не будет отличаться от ext4 или NTFS. В этом задании мы с Вами реализуем простую файловую систему: все файлы будут храниться в оперативной памяти компьютера или на удаленном сервере (по желанию), однако пользователь сможет пользоваться ими точно так же, как и файлами на жёстком диске.

Мы рекомендуем при выполнении этого задания использовать отдельную виртуальную машину: любая ошибка может вывести всю систему из строя, и вы можете потерять ваши данные. Именно виртуальную машину, а не контейнер типа Docker (почему?).

Вспомогательные материалы

- The Linux Kernel Module Programming Guide — справочный материал по разработке модулей ядра Linux.
- A simple native file system for Linux kernel — пример реализации модуля для создания простой файловой системы от автора книги «The Linux Kernel Module Programming Guide».
- Linux Kernel Development, 3rd Edition — книга с описанием устройства виртуальных файловых систем в Linux, глава 13.
- Understanding the Linux Kernel: From I/O Ports to Process Management 3rd Edition — книга с описанием работы и структур виртуальных файловых систем в Linux, глава 12.

Часть 1. Знакомство с простым модулем

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

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

Мы уже подготовили основу для вашего будущего модуля в файлах vtfs.c и 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`. Чтобы получить обратную связь от программы, мы можем печатать данные в системный лог с помощью функции printk.

В Makefile указано, что наш модуль `vtfs` состоит из одной единицы трансляции — `vtfs`. Вы можете самостоятельно добавлять новые единицы, чтобы декомпозировать ваш код удобным образом.

Соберём модуль.

```
make
```

Сборка модуля отличается от сборки обычных программ тем, что при этом происходит некоторая «✨магия✨». А именно, Makefile обрабатывается не обычным make, а особым, с дополнительными целями и переменными, так же выполняются другие незаметные операции.

Если наш код скомпилировался успешно, в директории `source` появится файл `vtfs.ko` — это и есть наш модуль. Осталось загрузить его в ядро. Загружается именно файл, поэтому указывается путь до содержимого модуля (с расширением `.ko`).

```
sudo insmod source/vtfs.ko
```

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

```
$ dmesg
<...>
[ 123.456789] [vtfs] VTFS joined the kernel
```

Для выгрузки модуля нам понадобится команда `rmmod`. Выгрузка уже происходит по имени модуля (без расширения `.ko`, см. команду `lsmod`).

```
$ sudo rmmod vtfs
$ dmesg
<...>
[ 123.987654] [vtfs] VTFS left the kernel
```

Часть 2. Подготовка файловой системы

Мы собираемся сделать простую файловую систему, которая сначала будет хранить данные в оперативной памяти, в самом начале даже просто симулировать наличие файлов, а позже будет предложено реализовать отправку по сети и хранение их на удаленной машине.

Операционная система предоставляет две функции для управления файловыми системами:

- register_filesystem — сообщает о появлении нового драйвера файловой системы.
- unregister_filesystem — удаляет драйвер файловой системы.

В этой части мы начнём работать с несколькими структурами ядра:

- inode — описание метаданных файла: имя файла, расположение, тип файла (в нашем случае — регулярный файл или директория)
- dentry — описание директории: список inode внутри неё, информация о родительской директории, ...
- super_block — 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 vtfs_fs_type = {
  .name = "vtfs",
  .mount = vtfs_mount,
  .kill_sb = vtfs_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* vtfs_mount(
  struct file_system_type* fs_type,
  int flags,
```

```
  const char* token,
  void* data
) {
  struct dentry* ret = mount_nodev(fs_type, flags, data, vtfs_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 командой mount:

```
sudo mount -t vtfs "<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, 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 vtfs_fill_super(struct super_block *sb, void *data, int silent) {
  struct inode* inode = vtfs_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 — `vtfs_get_inode`. Она будет создавать новую структуру `inode`, в нашем случае — для корня файловой системы:

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

  inode->i_ino = i_ino;
```

```
  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, передавая ему корневую `inode`. Для определенности корневая директория всегда будет иметь номер `1000`.

Для создания новой `inode` используем функцию new_inode. Кроме этого, с помощью функции inode_init_owner 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 — 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 vtfs_kill_sb(struct super_block* sb) {
  printk(KERN_INFO "vtfs 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 vtfs.ko
sudo mount -t vtfs "TODO" /mnt/vt
```

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

Теперь отмонтируем файловую систему:

```
sudo umount /mnt/vt
```

Часть 3. Вывод файлов и директорий

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

```
$ sudo mount -t vtfs "TODO" /mnt/vt
$ cd /mnt/vt
-bash: cd: /mnt/vt: 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. Например, такую:

```
struct inode_operations vtfs_inode_ops = {
  .lookup = vtfs_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* vtfs_lookup(
  struct inode* parent_inode,  // родительская нода
```

```
  struct dentry* child_dentry, // объект, к которому мы пытаемся получить доступ
  unsigned int 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/vt
-bash: cd: /mnt/vt: 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 vt
```

После этого мы сможем перейти в `/mnt/vt`, но не можем вывести содержимое директории. На этот раз нам понадобится не `i_op`, а `i_fop` — структура типа file_operations. Реализуем в ней первую функцию — `iterate`.

```
struct file_operations vtfs_dir_ops = {
  .iterate = vtfs_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 `vtfs_iterate` приведён ниже:

```
int vtfs_iterate(struct file* filp, struct dir_context* ctx) {
  char fsname[10];
  struct dentry* dentry = filp->f_path.dentry;
  struct inode* inode   = dentry->d_inode;
  unsigned long offset  = filp->f_pos;
  int stored            = 0;
  ino_t ino             = inode->i_ino;

  unsigned char ftype;
  ino_t dino;
  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/vt
ls: cannot access '/mnt/vt/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.

Часть 4. Навигация по директориям

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

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

Часть 5. Создание и удаление файлов

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

```
int vtfs_create(
  struct inode *parent_inode,
  struct dentry *child_dentry,
  umode_t mode,
  bool b
) {
  ino_t root = parent_inode->i_ino;
  const char *name = child_dentry->d_name.name;
  if (root == 100 && !strcmp(name, "test.txt")) {
    struct inode *inode = vtfs_get_inode(
        parent_inode->i_sb, NULL, S_IFREG | S_IRWXUGO, 101);
    inode->i_op = &vtfs_inode_ops;
    inode->i_fop = NULL;

    d_add(child_dentry, inode);
    mask |= 1;
  } else if (root == 100 && !strcmp(name, "new_file.txt")) {
    struct inode *inode = vtfs_get_inode(
        parent_inode->i_sb, NULL, S_IFREG | S_IRWXUGO, 102);
    inode->i_op = &vtfs_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 — `vtfs_unlink`.

```
int vtfs_unlink(struct inode *parent_inode, struct dentry *child_dentry) {
  const char *name = child_dentry->d_name.name;
  ino_t 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 6. Реализация с хранилищем данных в RAM

До сих пор мы писали лишь заглушку файловой системы - содержимое ФС было зашито в коде.

Теперь пришло время реализовать функции из предыдущих этапов, но используя оперативную память в качестве хранилища данных для нашей ФС. Например, можно хранить данные просто в массивах и связных списках.

Рекомендуем вам заранее выделить интерфейс файловой системы, чтобы упростить смену хранилища данных - это понадобится нам в следующих этапах.

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.

Часть 8*. 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 и write.

Соответствующие функции имеют следующие сигнатуры:

```
ssize_t vtfs_read(
  struct file *filp, // файловый дескриптор
  char *buffer,      // буфер в user-space для чтения и записи соответственно
  size_t len,        // длина данных для записи
```

```
  loff_t *offset     // смещение
);

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

Обратите внимание, что просто так обратиться в `buffer` нельзя, поскольку он находится в `user-space`. Используйте специальные функции 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

Часть 9*. 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 vtfs_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
```

Часть 10. Сервер файловой системы

При текущей реализации содержимое файловой системы не переживает перезапуск компьютера. Давайте исправим это и будем хранить данные на каком-то долговечном носителе. Чтобы было интереснее, вам придется реализовать удаленную файловую систему (подобную `netfs`).

Вам нужно на любом языке программирования реализовать сервер вашей файловой системы. В качестве хранилища на сервере можете использовать СУБД. Например, это может быть связка

Spring и PostgreSQL или ZIO и YDB.

Поскольку в ядре используются не совсем привычные функции для работы с сетью, вам предоставляется собственный HTTP-клиент в виде функции `vtfs_http_call` в файле http.c:

```
int64_t vtfs_http_call(
    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 или `errno-base.h` (`ENOMEM`, `ENOSPC`) в случае ошибки при выполнении запроса (отсутствие подключения, сбой в сети, некорректный ответ сервера, ...).

Требования к сдаче ЛР преподавателю

- Наличие отчета, который включает в себя ссылку на репозиторий, вывод о проделанной работе
- Готовность запустить тесты по просьбе преподавателя
- За хранилище данных в RAM вы сможете получить не более 10 баллов за ЛР, а за реализацию с сервером до 15 баллов.