Индивидуальное задание по курсу “Конструирование ядра ОС”

выполнил:
студент 327 группы
Миронов А. В.

ВМК МГУ
2017 год

1. Постановка задания

В рамках решения индивидуального задания требуется реализовать **именованные межпроцессные каналы**. Для этого потребуется реализовать специальный вид файлов в файловой системе JOS, а также предложить и реализовать API для создания (см. **mkfifo** из POSIX), открытия, удаления FIFO. Чтение и запись должны вестись при помощи стандартных функций **read** и **write** над файловым дескриптором.

В системе уже реализована возможность работы с неименованными каналами посредством вызова функции **pipe(int *fd)**. Отличием от именованных каналов является то, что они доступны лишь родственным процессам.

Именованные каналы (FIFO-файлы) расширяют свою область применения за счет того, что подключиться к ним может любой процесс в любое время, в том числе и после создания канала. Это возможно благодаря наличию у них имен.

Особенностью этих файлов является их организация по стратегии FIFO (т.е. невозможны операции, связанные с перемещением файлового указателя).

2. Реализация

2.1 Удаление файла любого типа

Для удаления реализуем подобие Unix-утилиты **rm.** Она будет принимать только один аргумент (тот файл, который надо удалить). Код **rm.c** будет вызывать функцию **int remove(const char *path) ([1]),** которая была объявлена еще в Лабораторной работе **№12** в файле **lib/file.c**, но не реализована. Функция **[1]** будет формировать запрос к файловой системе посредством вызова:
fsipc(FSREQ_REMOVE, NULL), перед отправкой запроса нужно в переменную **fsipcbuf** записать **req_path**, то имя файла, который мы хотим удалить.

На стороне сервера файловой системы диспетчеризующей функцией является **serve()**, которая примет наш запрос и вызовет функцию **serve_remove()**, которая вызовет функцию f**ile_remove()**, передавая ей имя файла. Функция **file_remove()** использует функцию **walk_path(),** чтобы найти файл по указанному имени, если находит, то в третий аргумент запишется указатель на **struct File**. Затем необходимо очистить все блоки в памяти, которые занимает удаляемый файл. Это делается с помощью функции **file_set_size(f,0)**, которая, делая новый размер файла равный 0, вызывает **file_truncate_blocks()**, которая в свою очередь вызывает **file_free_block()**.

После того, как очистили блоки занимаемые файлом, он все равно еще остался в нашей директории, чтобы совсем от него избавиться, нужно вызвать функцию **dir_remove_file(dir, f),** которая принимая два указателя на **struct File** (первый аргумент директория - откуда удаляем, второй - что удаляем), находит сначала блок директории, в котором находится наша структура, затем очищает память занимаемую память, занимаемую этой структурой (**memset(f, 0, sizeof(struct File))**), при необходимости будут очищены и блоки директории вместе с уменьшением размера директории.

2.2 Создание нового типа файла FIFO

Для использования FIFO-каналов будет реализована утилита **mkfifo**, которая принимает один аргумент (имя создаваемого канала). Будет произведен вызов функции **mkfifo(const char *path)**, которая отправит запрос файловой системе **fsipc(FSREQ_FIFO, fd).** После завершения выполнения запроса в файловой системе появится новый файл с типом **FTYPE_FIFO,** который командой **ls** будет виден как **FIFO file.** В этот раз после вызова **serve()** будет вызвана функция **serve_create_fifo(),** аналогично созданию файла будет вызвана функция **file_create().** Отличие лишь в том, что третий аргумент у этой функции будет типа файла **FTYPE_FIFO,** поэтому после выделения места в директории под новый **struct File**, будет в поле этой структуры **f_type** иметь значение **FTYPE_FIFO.**

2.3 Чтение и запись в FIFO файл

На данный момент мы можем писать в командной строке **mkfifo pipe.** В ФС появится новый файл с типом **FTYPE_FIFO.** Мы хотим реализовать чтение и запись в этот файл, чтобы было возможно выполнение следующих команд в командной строке:

mkfifo pipe
cat lorem > pipe &
num < pipe
rm pipe

Выполнение первой и четвертой команд уже возможно на данном этапе. Во второй команде используется символ **&,** который означает выполнение команды в фоновом режиме. Это нужно реализовать в коде **user/sh.c**. Реализовывать это будем с помощью вызова **fork(),** сыновний процесс будет выполнять эту команду, а отец заново выведет приглашение к вводу новой команды в командной строке.

Вторую команду **cat lorem>pipe & cat lorem>pipe &** будет выполнять сыновний процесс, он заблокируется на время, аналогично тому, как это сделано в функции записи в неименованный канал, до того момента, пока не начнет свое выполнение третья команда **num < pipe.**

При первом открытии этого файла будет вызываться функция **pipe(),** которая создаст и отобразит страницы под файловые дескрипторы на чтение и запись, а так же страницу под **struct Pipe,** в адресном пространстве процесса, который открывает этот **FIFO-файл**. В соответствующий эл-т массива **f_pad[ ],** запишется **envid** этого процесса. Если файл открывался на чтение, то в **f_pad[0]** запишется **envid** открывающего процесса, а если открывался на запись, то envid запишется в **f_pad[1]**. После первого открытия процесс блокируется до тех пор, пока не откроется этот **FIFO-файл** на противоположное действие(т.е. если был открыт на чтение, то блокируется до тех пор, пока не откроется другим процессом на запись, а если был открыт на запись, то блокируется, пока другой процесс не откроет файл на чтение). Когда файл открылся на противоположное действие, то в соответствующий эл-т **f_pad[ ]** запишется **envid** открывающего процесса. Процесс, который раньше открывал этот файл разблокируется и выполнит копирование отображения страниц, которые помечены как **PTE_SHARE** из своего адресного пространства в адресное пространство того envid, который находится в соответствующем эл-те **f_pad[ ]**. Для этого используется функция **copy_shared_pages().** После создания и настройки для работы **FIFO-файла,** чтение и запись будут вестись с помощью функций **read()** и **write(),** которые по номерам дескрипторов, которые им передаются, будут вызывать функции **devpipe_read()** и **devpipe_write()**. Функции **devpipe_*()** уже реализованы в нашей ОС. Наша задача была в том, чтобы правильно открыть **FIFO-файл** и настроить отображение страниц из двух разных адресных пространств не родственных процессов для общего использования ими.

Чтобы понять является ли открытие файла первым, будет реализована функция, которая обращается к файловой системе с соответствующим запросом, а в процессе сервера ФС будут прочитаны эл-ты **f_pad[0]** и **f_pad[1]** , если они оба нулевые значит файл открывается впервые. Также эта функция может нужным образом заполнить нужный эл-т **f_pad[ ].**

Для закрытия файла, будет реализована функция, которая делает запрос к файловой системе, а сервер ФС, получая этот запрос, будет в эл-ты **f_pad[0]** и **f_pad[1]** записывать нули.

Функция для закрытия файлового дескриптора самого файла уже реализована **devpipe_close().**

Также хотелось бы реализовать НЕ блокируемый вариант работы с **FIFO-файлами**. Для изменения режима работы будет реализована функция, которая посылает запрос файловой системе, сервер ФС, получая этот сигнал меняет нужным образом эл-т **f_pad[2].** Если там записан 0, то режим блокирующий, а если 1, то НЕ блокирующий. Работа в НЕ блокирующем режиме с **FIFO-файлами** аналогична работе с обычными файлами.

3. Тесты

Для утилиты “rm” следующие тесты:

1)Удаление пустого файла.

Выполняемые действия:

-Создать файл с помощью утилиты **touch**, затем убедиться в его создании, выполнив команду **ls -l.** Удалить файл при помощи команды **rm.** Снова проверить удаление с помощью **ls -l.**

Ожидаемый результат:

-При первом вызове **ls** должен вывести последним файлом в директории наш новый файл с размером равным 0, а при втором вызове **ls** должен вывести то, что файлов в директории стало на 1 меньше и уже нет имени нашего файла в списке.

2)Удаление файла, при условии, что он не последний в директории.

Выполняемые действия:

-Создать два файла (или любое другое кол-во) с помощью **touch.** Выполнить **ls -l,** чтобы убедиться в создании файлов. Удалить первый созданный файл. Снова выполнить **ls -l.** Заново файл, который удалили. Опять выполнить **ls -l.**

Ожидаемый результат:

-При первом вызове **ls** должен вывести двумя последними в списке файлов в директории наши новые файлы, а при втором вызове **ls** должен вывести то, что файлов в директории стало на 1 меньше и уже нет имени нашего первого созданного файла в списке. После третьего вызова **ls** должны увидеть, что наши файлы “будто” поменялись местами в директории.

3)Удаление файла, при условии, что он последний в последнем блоке.

Выполняемые действия:

-Создать столько файлов, чтобы их кол-во стало кратным 16 (именно столько находится структур файлов в одном блоке директории). Выполнить **ls -l,** чтобы убедиться в создании. Запомнить размер директории, который **ls** выводит почти в начале. Создать еще один файл, снова выполнить **ls -l,** убедиться, что размер директории вырос ровно на один блок (то есть + 4096 байт). Выполнить **rm** нашего файла, созданного последним.
Снова выполнить **ls -l.**

Ожидаемый результат:

-При первом вызове **ls** должен вывести список всех файлов директории. Если всего файлов **N**, то размер директории должен быть равен **(N / 16) * BLKSIZE (**не

забываем **N** должно быть кратно 16**).** После второго вызова **ls,** мы должны увидеть увеличение размера директории на 4096 и последним в списке наш новый файл, который является последним в блоке. При его удалении, блок должен освободиться, а размер директории, соответственно, должен уменьшиться на 4096. Все это проверяется выводом **ls -l.**

4)Удаление файла, при условии, что он не последний в не последнем блоке. (то есть самый общий случай удаления файла)

Выполняемые действия:

-Создать сколько угодно файлов. Убедиться в создании аналогично предыдущим тестам. Удалить какой-нибудь файл из середины (не обязательно из середины, удаляемый файл может быть на любом месте в директории). Снова выполнить **ls -l.** Проверить результаты удаления (правильный ли список выведен и правильный ли размер директории?)

Ожидаемый результат:

-Ожидается увидеть в выводе команды не противоречащий список файлов и правильный размер директории. Интересный случай здесь заключается в следующем. Попробуйте создавать файлы в таком же кол-ве как и в тесте №3, но удаляйте не последний созданный, а какой-нибудь другой файл. Ожидается, что кол-во файлов опять станет кратным 16 и должен освободиться блок директории, то есть уменьшится ее размер, за которым нужно проследить в выводе списка файлов. Здесь как раз и проверяется, чтобы файл не просто удалялся, но и другие файлы соответствующем способом перемещались в директории.

!!!Все тесты, которые проводятся над утилитой **rm** можно и нужно проводить как с пустыми, так и с НЕ пустыми файлами (чтобы сделать их не пустыми, можно просто после создания выполнять команды **cat lorem > new_file** или **cat bigfile > new_file**).

Для именованных каналов следующие тесты:

Выполняемые действия:

mkfifo pipe
cat lorem > pipe &
num < pipe

Ожидаемый результат:

На экран выведется то же самое, если бы вместо предыдущих команд была бы введена команда: **cat lorem | num.**