

1. Билет №4

Файловая подсистема `/proc` – назначение, особенности, файлы, поддиректории, ссылка `self`, информация об окружении, состоянии процесса, прерываниях. Структура `proc_dir_entry`: функции для работы с элементами `/proc`. Структура, перечисляющая функции, определенные на файлах. Использование структуры `file_operations` для регистрации собственных функций работы с файлами. Передача данных из пространства пользователя в пространство ядра и из ядра в пространство пользователя. Обоснование необходимости этих функций. Функция `printk()` – назначение и особенности. Пример программы «Фортуны» из лаб. работы.

1.1. Файловая подсистема

Файл — важнейшее понятие в файловой подсистеме. Файл — информация, хранимая во вторичной памяти или во вспомогательном ЗУ с целью ее сохранения после завершения отдельного задания или преодоления ограничений, связанных в объеме основного ЗУ.

Файл — поименованная совокупность данных, хранимая во вторичной памяти (возможно даже целая). Файл — каждая индивидуально идентифицированная единица информации.

Существует 2 ипостаси файла:

1. файл, который лежит на диске;
2. открытый файл (с которым работает процесс).

Открытый файл — файл, который открывает процесс.

Файл != место на диске. В мире современной вычислительной техники файлы имеют настолько большие размеры, что не могут храниться в непрерывном физическом адресном пространстве, они хранятся вразброс (несвязанное распределение).

Файл может занимать разные блоки/сектора/дорожки на диске аналогично тому, как память поделена на страницы. В любой фрейм может быть загружена новая страница, как и файл.

Также, важно понимать адресацию.

Соответственно, система должна обеспечить адресацию каждого такого участка.

ОС является загружаемой программой, её не называют файлом, но когда компьютер включается, ОС находится во вторичной памяти. Затем с помощью нескольких команд, которые находятся в ПЗУ, ОС (программа) загружается в ОЗУ. При этом выполняется огромное количество действий, связанных с управлением памятью, и без ФС это сделать невозможно. Любая ОС без ФС не может быть полноценной.

Задача ФС — обеспечивать сохранение данных и доступ к сохраненным данным (обеспечивать работу с файлами).

Чтобы обеспечить хранение файла и последующий доступ к нему, файл должен быть изолирован, то есть занимать некоторое адресное пространство, и это адресное пространство должно быть защищено. Доступ обеспечивается по тому, как файл идентифицируется в системе (доступ осуществляется по его имени).

ФС — порядок, определяющий способ организации хранения, именования и доступа к данным на вторичных носителях информации.

File management (управление файлами) — программные процессы, связанные с общим управлением файлами, то есть с размещением во вторичной памяти, контролем доступа к файлам, записью резервных копий, ведением справочников (directory).

Основные функции управления файлами обычно возлагаются на ОС, а дополнительные — на системы управления файлами.

Доступ к файлам: open, read, write, rename, delete, remove.

Разработка UNIX началась с ФС. Без ФС невозможно создание приложений, работающих в режиме пользователя (сложно разделить user mode и kernel mode).

Файловая подсистема взаимодействует практически со всеми модулями ОС, предоставляя пользователю возможность долговременного хранения данных, а также ОС возможность работать с объектами ядра.

1.2. Файловая подсистема /proc – назначение, особенности

Виртуальная файловая система proc не является монтируемой файловой системой поэтому и называется виртуальной. Ее корневым каталогом является каталог **/proc**, ее поддиректории и файлы создаются при обращении, чтобы предоставить информацию из структур ядра.

Proc нужна для того, чтобы в режиме пользователя была возможность получить информацию о системе и ее ресурсах (например прерываниях).

Основная задача файловой системы proc – предоставление информации процессам о занимаемых ими ресурсах.

Для того чтобы ФС была доступна, она должна быть подмонтирована, в результате должна быть доступна информация из суперблока, который является основной структурой, описывающей файловую систему. Когда происходит обращение к ФС proc, информация к которой идет обращение создается на лету, то есть файловая система виртуальная.

Файловая система монтируется при загрузке системы. Но ее также можно смонтировать вручную:

```
mount -t proc proc/proc
```

Это сделано для общности - система работает единообразно со всеми файловыми системами.

1.3. Файлы, поддиректории, ссылка self, информация об окружении, состоянии процесса, прерываниях

Каждый процесс в фс proc имеет поддиректорию: **/proc/<PID>**. Для данной поддиректории для каждого процесса существует символическая ссылка **/proc/self** для того, чтобы не вызывать функцию `getpid()` — when a process accesses this magic symbolic link, it resolves to the process's own **/proc/[pid]** directory.

№	Элемент	Тип	Описание
1	cmdline	файл	Командная строка запуска процесса
2	cwd	символическая ссылка	Рабочая директория процесса
3	environ	файл	Содержит список окружения процесса
4	exe	символическая ссылка	Указывает на образ процесса
5	fd	директория	Содержит ссылки на файлы, открытые процессом
6	maps	файл (регионы виртуального адресного пространства)	Содержит список регионов (выделенных процессу участков памяти) виртуального адресного пространства процесса (У процессов только виртуал. адресное пространство, а физ. память выделяется по прерыванию pagefault)
7	page map	файл	Отображение каждой виртуальной страницы адресного пространства на физический фрейм или область свопинга
8	tasks	директория	Содержит поддиректории потоков
9	root	символическая ссылка	Указывает на корень фс процесса
10	stat	файл	Информация о состоянии процесса (pid, comm, state, ppid, pgrp, session, tty_nr, tpgid, flags и др.)
11	status	файл	Большая часть инфы из stat и statm в норм формате
12	statm	файл	Информация о памяти

Про maps:

1. address — начальный и конечный адреса региона виртуальной памяти
2. perms — права доступа к региону
3. offset — если процесс отображен из файла, то это смещение (региона) в этом файле

4. `dev` — если процесс отображен из файла, то это старший и младший номера устройства, на котором находится этот файл
5. `inode` — № `inode` для файла, если процесс отображен из файла
6. `pathname` — если процесс отображен из файла, то это путь к файлу

Про `pagemap` (информация о страницах виртуальной памяти процесса):

1. `addr` — начальный адрес страницы
2. `pfm` — № фрейма (физическая страница), на котором находится страница

Флаги `pagemap`:

1. `soft-dirty` — была ли страница изменена в оперативной памяти: если она была изменена, то нужно перезаписать точную копию страницы с диска
2. `file/shared` — является ли страница разделяемой
3. `swapped` — находится ли страница в файле подкачки (области свопинга)
4. `present` — загружена ли страница в оперативную память

Файл `/proc/interrupts` предоставляет таблицу о прерываниях на каждом из процессоров в следующем виде:

- Первая колонка: линия `IRQ`, по которой приходит сигнал от данного прерывания
- Колонки `CPUx`: счётчики прерываний на каждом из процессоров
- Следующая колонка: вид прерывания:
 - `IO-APIC-edge` — прерывание по фронту (срабатывает каждый раз до выполнения) на контроллер `I/O APIC`
 - `IO-APIC-fasteoi` — прерывание по уровню (срабатывает 1 раз по факту появления) на контроллер `I/O APIC`
 - `PCI-MSI-edge` — `MSI` прерывание
 - `XT-PIC-XT-PIC` — прерывание на `PIC` контроллер
- Последняя колонка: названия обработчиков данного прерывания

Код для чтения `/proc/self/environ`:

Листинг 1.1: код для чтения /proc/self/environ

```
1 #include <stdio.h>
2 #define BUF_SIZE 0x100
3 int main(int args, char * argv[])
4 {
5     char buf[BUF_SIZE];
6     int len;
7     FILE *f;
8     f = open("/proc/self/environ", "r");
9     while((len = fread(buf, 1, BUF_SIZE, f) > 0)
10    {
11        // Строки в файле разделены не \n, а \0 (\n = 10 = 0
12        // x0A)
13        for (i = 0; i < len; i++)
14            if (buf[i] == 0)
15                buf[i] = 10; // for 0x0A
16        buf[len] = 0;
17        printf("%s", buf);
18    }
19    fclose(f);
20    return 0;
21 }
```

1.4. Структура proc_dir_entry: функции для работы с элементами /proc

Чтобы работать с proc в ядре, надо создать в ней файл. В ядре определена структура

Листинг 1.2: Структура proc_dir_entry

```
1 <linux/proc_fs.h>
2 struct proc_dir_entry {
3     atomic_t in_use;
4     refcount_t refcnt;
5     struct list_head pde_openers;
6     spinlock_t pde_unload_lock; // Собственное средство взаимногоисключения
7     ...
8 }
```

```

8  const struct inode_operations *proc_iops; // Операции определенные на
      inode фс proc
9  union {
10   const struct proc_ops *proc_ops;
11   const struct file_operations *proc_dir_ops;
12 };
13 const struct dentry_operations *proc_dops; // Используется для регистрации
      своих операций над файлом в proc
14 union {
15   const struct seq_operations *seq_ops;
16   int (*single_show)(struct seq_file *, void *);
17 };
18 proc_write_t write;
19 void *data;
20 unsigned int state_size;
21 unsigned int low_ino;
22 nlink_t nlink;
23     ...
24 loff_t size;
25 struct proc_dir_entry *parent;
26     ...
27 char *name;
28 u8 flags;
29     ...
30 }

```

Структура proc_ops позволяет определять операции для работы с файлами в драйверах. Флаги определяют особенности работы со структурами.

Листинг 1.3: Структура proc_ops

```

1  struct proc_ops {
2   unsigned int proc_flags;
3   int (*proc_open)(struct inode *, struct file *);
4       // в таблице открытых файлов об одном файле находится столько записей,
      сколько раз он был открыт
5   ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
6   ...
7   ssize_t (*proc_write)(struct file *, const char __user *, size_t, loff_t
      *);

```

```

8  loff_t (*proc_lseek)(struct file *, loff_t , int);
9  int (*proc_release)(struct inode *, struct file *);
10 ...
11 long (*proc_ioctl)(struct file *, unsigned int , unsigned long);
12 }

```

1.5. Функции для работы с элементами /proc

На `proc` определена функция `proc_create_data`:

Листинг 1.4: Функция `proc_create_data`

```

1 extern struct proc_dir_entry *proc_create_data(const char *, umode_t,
2   struct proc_dir_entry *,
3   const struct proc_ops *, void *);

```

Есть более популярная обертка – `proc_create`:

Листинг 1.5: Функция `proc_create`

```

1 static inline struct proc_dir_entry *proc_create(const char *name, umode_t
   mode,
2   struct proc_dir_entry *parent ,
3   const struct proc_ops *proc_ops)
4 {
5   return proc_create_data(name, mode, parent , proc_ops , NULL);
6 }

```

Создавать каталоги в файловой системе `/proc` можно используя `proc_mkdir()`, а также символические ссылки с `proc_symlink()`. Для простых элементов `/proc`, для которых требуется только функция чтения, используется `create_proc_read_entry()`, которая создает запись `/proc` и инициализирует функцию `read_proc` в одном вызове.

Листинг 1.6: Хз чо это

```

1 #include <linux/types.h>
2 #include <linux/fs.h>
3
4 extern struct proc_dir_entry *proc_symlink(const char *, struct
   proc_dir_entry *, const char *);
5 extern struct proc_dir_entry *proc_mkdir(const char *, struct
   proc_dir_entry *);

```



```

6 struct proc_dir_entry *create_proc_read_entry( const char *name, mode_t
    mode, struct proc_dir_entry *base, read_proc_t *read_proc, void *data )
    ;
7 }

```

1.6. Структура, перечисляющая функции, определенные на файлах. Использование структуры `file_operations` для регистрации собственных функций работы с файлами

Связь между `struct file` и `struct file_operations`

Файл должен быть открыт. Соответственно для открытого файла должен быть создан дескриптор. В этом дескрипторе имеется указатель на `struct file_operations`. Это либо стандартные (установленные по умолчанию) операции на файлах для конкретной файловой системы, либо зарегистрированные разработчиком (собственные функции работы с файлами собственной файловой системы).

```

1 struct file_operations {
2 struct module *owner;
3 loff_t (*llseek) (struct file *, loff_t, int);
4 ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5 ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
6 ...
7 int (*open) (struct inode *, struct file *);
8 ...
9 int (*release) (struct inode *, struct file *);
10 ...
11 } __randomize_layout;

```

Разработчики драйверов должны регистрировать свои функции `read/write`. В UNIX/Linux все файл как раз для того, чтобы свести се действия к однотипным операциям `read/write` и не размножать их, а свести к большому набору операций.

Для регистрации своих функций используется(-лась) `struct file_operations`. С некоторой версии ядра 5.16+ (примерно) появилась `struct proc_ops`. В загружаемых модулях ядра можно использовать условную компиляцию:

```

1 #if LINUX_VERSION_CODE >= KERNEL_VERSION(5,6,0)

```

```

2 #define HAVE_PROC_OPS
3 #endif
4
5 #ifdef HAVE_PROC_OPS
6 static struct proc_ops fops = {
7     .proc_read = fortune_read ,
8     .proc_write = fortune_write ,
9     .proc_open = fortune_open ,
10    .proc_release = fortune_release ,
11 };
12 #else
13 static struct file_operations fops = {
14     .owner = THIS_MODULE,
15     .read = fortune_read ,
16     .write = fortune_write ,
17     .open = fortune_open ,
18     .release = fortune_release ,
19 };
20 #endif

```

proc_open и open имеют одни и те же формальные параметры (указатели на struct inode, struct file). С другими функциями аналогично.

Зачем так сделано? — proc_ops сделана для того, чтобы не вешаться на file_operations, которые используются драйверами. Функции file_operations настолько важны для системы, что их решили освободить от работы с ФС proc.

1.7. Передача данных из пространства пользователя в пространство ядра и из ядра в пространство пользователя. Обоснование необходимости этих функций

Чтобы передать данные из адресного пространства пользователя в адресное пространство ядра и обратно используются функции copy_from_user() и copy_to_user():

```

1 unsigned long __copy_to_user(void __user *to, const void *from, unsigned
    long n);
2 unsigned long __copy_from_user(void *to, const void __user *from,
    unsigned long n);

```

Если некоторые данные не могут быть скопированы, эта функция добавит нулевые байты к скопированным данным до требуемого размера.

Обе функции возвращают количество байт, которые не могут быть скопированы. В случае выполнения будет возвращен 0.

Обоснование необходимости этих функций

Ядро работает с физическими адресами (адреса оперативной памяти), а у процессов адресное пространство виртуальное (это абстракция системы, создаваемая с помощью таблиц страниц).

~~Фреймы (физические страницы) выделяются по прерываниям.~~

Может оказаться, что буфер пространства пользователя, в который ядро пытается записать данные, выгружен.

И наоборот, когда приложение пытается передать данные в ядро, может произойти аналогичная ситуация.

Поэтому нужны специальные функции ядра, которые выполняют необходимые проверки.

Что можно передать из user в kernel?

Например, с помощью передачи из user mode выбрать режим работы загружаемого модуля ядра (какую информацию хотим получить из загружаемого модуля ядра в данный момент).

Такое "меню" надо писать в user mode и передавать соответствующие запросы модулям ядра.

1.8. Функция `printk()` – назначение и особенности

Функция `printk()` определена в ядре Linux и доступна модулям. Функция аналогична библиотечной функции `printf()`. Загружаемый модуль ядра не может вызывать обычные библиотечные функции, поэтому ядро предоставляет модулю функцию `printk()`. Функция пишет сообщения в системный лог, который можно посмотреть, используя `sudo dmesg`.

Пример из лабораторной «Фортунки»:

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/vmalloc.h>
```

```

5 #include <linux/proc_fs.h>
6 #include <linux/uaccess.h>
7
8 MODULE_LICENSE("GPL");
9 MODULE_AUTHOR("Karpova_Ekaterina");
10
11 #define BUF_SIZE PAGE_SIZE
12
13 #define DIRNAME "fortunes"
14 #define FILENAME "fortune"
15 #define SYMLINK "fortune_link"
16 #define FILEPATH DIRNAME "/" FILENAME
17
18 static struct proc_dir_entry *fortune_dir = NULL;
19 static struct proc_dir_entry *fortune_file = NULL;
20 static struct proc_dir_entry *fortune_link = NULL;
21
22 static char *cookie_buffer;
23 static int write_index;
24 static int read_index;
25
26 static char tmp[BUF_SIZE];
27
28 ssize_t fortune_read(struct file *filp, char __user *buf, size_t count,
    loff_t *offp)
29 {
30     int len;
31     printk(KERN_INFO "+_fortune:_read_called");
32     if (*offp > 0 || !write_index)
33     {
34         printk(KERN_INFO "+_fortune:_empty");
35         return 0;
36     }
37     if (read_index >= write_index)
38         read_index = 0;
39     len = snprintf(tmp, BUF_SIZE, "%s\n", &cookie_buffer[read_index]);
40     if (copy_to_user(buf, tmp, len))
41     {

```

```

42     printk(KERN_ERR "+_fortune:_copy_to_user_error");
43     return -EFAULT;
44 }
45 read_index += len;
46 *offp += len;
47 return len;
48 }
49
50 ssize_t fortune_write(struct file *filp, const char __user *buf, size_t
    len, loff_t *offp)
51 {
52     printk(KERN_INFO "+_fortune:_write_called");
53     if (len > BUF_SIZE - write_index + 1)
54     {
55         printk(KERN_ERR "+_fortune:_cookie_buffer_overflow");
56         return -ENOSPC;
57     }
58     if (copy_from_user(&cookie_buffer[write_index], buf, len))
59     {
60         printk(KERN_ERR "+_fortune:_copy_to_user_error");
61         return -EFAULT;
62     }
63     write_index += len;
64     cookie_buffer[write_index - 1] = '\\0';
65     return len;
66 }
67
68 int fortune_open(struct inode *inode, struct file *file)
69 {
70     printk(KERN_INFO "+_fortune:_called_open");
71     return 0;
72 }
73
74 int fortune_release(struct inode *inode, struct file *file)
75 {
76     printk(KERN_INFO "+_fortune:_called_release");
77     return 0;
78 }

```

```

79
80 static const struct proc_ops fops = {
81     proc_read: fortune_read ,
82     proc_write: fortune_write ,
83     proc_open: fortune_open ,
84     proc_release: fortune_release
85 };
86
87 static void freemem(void)
88 {
89     if (fortune_link)
90         remove_proc_entry(SYMLINK, NULL);
91     if (fortune_file)
92         remove_proc_entry(FILENAME, fortune_dir);
93     if (fortune_dir)
94         remove_proc_entry(DIRNAME, NULL);
95     if (cookie_buffer)
96         vfree(cookie_buffer);
97 }
98
99 static int __init fortune_init(void)
100 {
101     if (!(cookie_buffer = vmalloc(BUF_SIZE)))
102     {
103         freemem();
104         printk(KERN_ERR "+_fortune:_error_during_vmalloc");
105         return -ENOMEM;
106     }
107     memset(cookie_buffer , 0, BUF_SIZE);
108     if (!(fortune_dir = proc_mkdir(DIRNAME, NULL)))
109     {
110         freemem();
111         printk(KERN_ERR "+_fortune:_error_during_directory_creation");
112         return -ENOMEM;
113     }
114     else if (!(fortune_file = proc_create(FILENAME, 0666, fortune_dir, &fops
115         )))
116     {

```

```

116     freemem();
117     printk(KERN_ERR "+_fortune:_error_during_file_creation");
118     return -ENOMEM;
119 }
120 else if (!(fortune_link = proc_symlink(SYMLINK, NULL, FILEPATH)))
121 {
122     freemem();
123     printk(KERN_ERR "+_fortune:_error_during_symlink_creation");
124     return -ENOMEM;
125 }
126 write_index = 0;
127 read_index = 0;
128 printk(KERN_INFO "+_fortune:_module_loaded");
129 return 0;
130 }
131
132 static void __exit fortune_exit(void)
133 {
134     freemem();
135     printk(KERN_INFO "+_fortune:_module_unloaded");
136 }
137
138 module_init(fortune_init)
139 module_exit(fortune_exit)

```

Загадки

Зачем модуль ядра? — чтобы передать информацию из kernel в user и наоборот (в ядре много важной инфы; из юзера — например, для управления режимом работы модуля)

Какой буфер? — кольцевой

Чей буфер? — Путина (пользователя)

Точки входа? — 6 штук: инит, ехит, рид, райт, опен, релиз

Когда вызывается какая точка? — инит на загрузке, ехит при выгрузке, рид когда вызывает кат, райт когда эхо, опен при открытии (во время чтения/записи), релиз при закрытии (во время чтения/записи)

Какие функции ядра вызываем? (Какие основные для передачи данных) — copy_from_user и copy_to_user

Когда вызываем `copy_from_user` и `copy_to_user`? — `copy_from_user` на записи (при вызове функции `write`), `copy_to_user` на записи (при вызове функции `read`)

Обоснование необходимости функций `copy from/to`

Ядро работает с физическими адресами (адреса оперативной памяти), а у процессов адресное пространство виртуальное (это абстракция системы, создаваемая с помощью таблиц страниц).

~~Фреймы (физические страницы) выделяются по прерываниям.~~

Может оказаться, что буфер пространства пользователя, в который ядро пытается записать данные, выгружен.

И наоборот, когда приложение пытается передать данные в ядро, может произойти аналогичная ситуация.

Поэтому нужны специальные функции ядра, которые выполняют необходимые проверки.

Что можно передать из `user` в `kernel`?

Например, с помощью передачи из `user mode` выбрать режим работы загружаемого модуля ядра (какую информацию хотим получить из загружаемого модуля ядра в данный момент).

Такое "меню" надо писать в `user mode` и передавать соответствующие запросы модулям ядра.