1. Билет №15

Загружаемые модули ядра. Структура загружаемых модулей. Информация о процессах, доступная в ядре. Пример вывода информации о запущенных процессах, символ current (лаб. раб.). Взаимодействие загружаемых модулей в ядре. Экспорт данных. Пример взаимодействия модулей (лаб. раб.). Функция printk() – назначение и особенности. Регистрация функций работы с файлами. Пример заполненной структуры. Передача данных из пространства ядра в пространство пользователя и из пространства пользователя в пространство ядра. Примеры из лабораторный работ.

1.1. Загружаемые модули ядра

Linux имеет монолитное ядро. Чтобы внести в него изменения, надо его перекомпилировать (патчи под глаза). Это опасно, так как можно получить неработающее ядро. В Unix/Linux можно вносить изменения без перекомпиляции, с помощью загружаемых модулей ядра. Это ПО, которое пишется по определенным шаблонам.

1.2. Структура загружаемых модулей

Загружаемые модули ядра имеют как минимум две точки входа — `init` и `exit`. Имена точек входа передаются с помощью макроса:

```
1 module_init(<точка входа init>)
2 module_exit(<точка входа exit>)
```

Макрос `module_init` служит для регистрации функции инициализации модуля. Макрос принимает в качестве параметра указатель на соответствующую функцию. В результате эта функция будет вызываться при загрузке модуля в ядро. Функция инициализации:

```
1 static int __init my_module_init();
```

Если функция инициализации завершилась успешно, то возвращается 0. В случае ошибки возвращается ненулевое значение.

> Как правило, функция инициализации предназначена для запроса ресурсов, выделения памяти под структуры данных и т. п. Так как функция инициализации редко вызывается за пределами модуля, ее обычно не нужно экспортировать и можно объявить с ключевым словом `static`. Если функция инициализации вернет ненулевое значение, модуль загружен не будет.

Макрос `module_exit` служит для регистрации функции, которая вызывается при выгрузке модуля из ядра. Обычно эта функция выполняет задачу освобождения занятых ресурсов. Функция выгрузки:

```
1 static void __exit my_module_exit();
```

Загружаемые модули ядра — многовходовые программы. Две точки входа всегда обязательны — `init` и `exit`.

Некоторые точки могут вызываться из `init`, тогда их можно будет назвать точками входа с натяжкой.

> Сама ОС имеет много точек входа (системные вызовы, исключения, аппаратные прерывания), но в каждом случае вызываются разные коды ядра (системный вызов → функция ядра, то есть интерфейс между kernel и user mode).

1.2.1. Пример модуля ядра «Hello World»

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4
5 // Указание лицензии обязательно
6 MODULE_LICENSE("GPL");
7
```

```
8  static int __init my_module_init()
9  {
10     printk(KERN_INFO "Hello, World!\n");
11     return 0;
12 }
13
14 static void __exit my_module_exit()
15 {
16     printk(KERN_INFO "Stop\n"); // print kernel – пишет в системный лог, д
           оступна в ядре.
17     // KERN_INFO – уровень протоколирования. Запятая между уровнем и строк
           ой не нужна.
18 }
19
20 module_init(my_module_init);
21 module_exit(my_module_exit); // Макрос. Ядро информируется о том, что в яд
       ре теперь есть эти функции.
```

В модулях ядра есть только библиотеки ядра, стандартные библиотеки недоступны.

1.2.2. Пример Makefile

Для компиляции нужен Makefile. C — опция смены каталога, переходим в каталог исходных кодов ядра. M — указывает Makefile вернуться в директорию исходных кодов модуля.

```
1 obj_m += module.o
2
3 all:
4     make -C /lib/modules/$(shell uname -r)/build M=$(pwd) modules
5
6 clean:
7     make -C /lib/modules/$(shell uname -r)/build M=$(pwd) clean
```

1.3. Информация о процессах, доступная в ядре. Пример вывода информации о запущенных процессах, символ current (лаб. раб.)

Модуль для получения информации о процессах из ядра — в ядре больше информации. Также, нужен доступ к
task_struct (доступна только в ядре).

Есть ссылка current на текущий процесс (insmod — в лабе). Проходим по кольцевому списку процессов в ядре.

> migration — перераспределение нагрузки (процессов) между логическими ядрами. У него приоритет 0 (макс), политика 1, так как важный процесс, чтобы никакой другой не мог его прервать. Их столько же, сколько логических ядер.
>
> Информация о: названии процесса, пид процесса, название и пид родителя, приоритет (120 — базовый), состояние процесса, политика планирования, название корневого каталога (точки монтирования) ФС, к которой относится файл процесса.
>
> Политики:
>
> 0 — по умолчанию
>
> 1 — FIFO (выполняется от начала до конца, нельзя вытеснить этот процесс, высокий приоритет)
>
> 2 — Round Robin (процессы могут быть возвращены в очередь после исчерпания кванта)

```
1     #include <linux/init_task.h>
2  #include <linux/module.h>
3  #include <linux/sched.h>
4  #include <linux/fs_struct.h>
5
6  MODULE_LICENSE("GPL");
7  MODULE_AUTHOR("Name");
8
9  static int __init mod_init(void)
10 {
11     printk(KERN_INFO "+module is loaded.\n");
```

```
12      struct task_struct *task = &init_task;
13      do
14      {
15          printk(KERN_INFO "+%s(%d)(%d-state,%d-prio,flags-%d,
                policy-%d),parent%s(%d),d_name%s",
16              task->comm, task->pid, task->__state, task->prio, task->flags,
                    task->policy, task->parent->comm, task->parent->pid, task
                    ->fs->root.dentry->d_name.name);
17      } while ((task = next_task(task)) != &init_task);
18
19      // task = current;
20      printk(KERN_INFO "+%s(%d)(%d-state,%d-prio,flags-%d,
            policy-%d),parent%s(%d),d_name%s",
21          current->comm, current->pid, current->__state, current->prio,
                current->flags, current->policy, current->parent->comm, current
                ->parent->pid, current->fs->root.dentry->d_name.name);
22      return 0;
23  }
24
25  static void __exit mod_exit(void)
26  {
27      printk(KERN_INFO "+%s-%d,parent%s-%d\n", current->comm,
28              current->pid, current->parent->comm, current->parent->pid);
29      printk(KERN_INFO "+module is unloaded.\n");
30  }
31
32  module_init(mod_init);
33  module_exit(mod_exit);
```

Дескрипторы процессов организованы в ядре в кольцевой список: начало - init_task, переход на следующий дескриптор - next_task().

insmod – загрузить модуль ядра

lsmod – посмотреть список модулей ядра

rmmod – выгрузить загруженный модуль из ядра

Все эти действия доступны только с правами superuser

Вывести инф. из лога: dmesg

При этом надо использовать ссылку current (текущий процесс)

Всего 8 уровней протоколирования (уровней вывода сообщений)

0 – KERN_EMERGE (опасность, упала система)

1 – KERN_ALERT (воздушная тревога на Украине, система скоро упадет)

2 – KERN_CRIT

3 – KERN_ERR

4 – KERN_WARNING

5 – KERN_NOTICE

6 – KERN_INFO

7 – KERN_DEBUG

Несмотря на то, что у user есть proc, в ядре информации все равно больше

1.4. Взаимодействие загружаемых модулей в ядре. Экспорт данных

Для того, чтобы в модуле использовать данные из другого модуля, нужно иметь абсолютные адреса экспортируемых имён ядра (символов) и модулей, по которым происходит связывание имён в компилируемом модуле.

Абсолютный адрес - адрес в физической памяти. Цилюрик называет это абсолютными адресами в пространстве ядра.

Процессы имеют виртуальное адресное пространство, а ядро оперирует физическими адресами.

1.5. Пример взаимодействия модулей (лаб. раб.)

kernel_module.h

```
1 extern char *module_1_data;
2 extern char *module_1_proc(void);
3 extern char *module_1_noexport(void);
```

module_1.c

```
1     #include <linux/init.h>
2 #include <linux/module.h>
3
4 // #include "kernel_module.h"
5
6 MODULE_LICENSE("GPL");
7 MODULE_AUTHOR("Inspirate789");
```

```
8
9  char *module_1_data = "ABCDE";
10
11 extern char *module_1_proc(void) { return module_1_data; }
12 static char *module_1_local(void) { return module_1_data; }
13 extern char *module_1_noexport(void) { return module_1_data; }
14
15 EXPORT_SYMBOL(module_1_data);
16 EXPORT_SYMBOL(module_1_proc);
17
18 static int __init module_1_init(void)
19 {
20      printk("+ module_1 started.\n");
21      printk("+ module_1 use local from module_1: %s\n", module_1_local());
22      printk("+ module_1 use noexport from module_1: %s\n",
            module_1_noexport());
23      return 0;
24 }
25
26 static void __exit module_1_exit(void) { printk("+ module module_1
        unloaded.\n"); }
27
28 module_init(module_1_init);
29 module_exit(module_1_exit);
```

module_2.c

```
1      #include <linux/init.h>
2  #include <linux/module.h>
3
4  #include "kernel_module.h"
5
6  MODULE_LICENSE("GPL");
7  MODULE_AUTHOR("Inspirate789");
8
9  static int __init module_2_init(void)
10 {
11      printk("+ module module_2 started.\n");
12      printk("+ data string exported from module_1: %s\n", module_1_data);
13      printk("+ string returned module_1_proc(): %s\n", module_1_proc());
```

```
14
15      //printk( "+ module_2 use local from module_1: %s\n", module_1_local()
            );
16
17      //printk( "+ module_2 use noexport from module_1: %s\n",
            module_1_noexport());
18
19      return 0;
20  }
21
22  static void __exit module_2_exit(void)
23  {
24      printk("+ module_2 unloaded.\n");
25  }
26
27  module_init(module_2_init);
28  module_exit(module_2_exit);
```

module_3.c

```
1       #include <linux/init.h>
2   #include <linux/module.h>
3
4   #include "kernel_module.h"
5
6   MODULE_LICENSE("GPL");
7   MODULE_AUTHOR("Inspirate789");
8
9   static int __init module_2_init(void)
10  {
11      printk("+ module_3 started.\n");
12      printk("+ data string exported from module_1: %s\n", module_1_data);
13      printk("+ string returned module_1_proc() is: %s\n", module_1_proc());
14
15      return -1;
16  }
17  module_init(module_2_init);
```

1.5.1. Ошибки

extern сообщает компилятору, что следующие за ним типы и имена определены в другом месте. Внешние модули могут использовать только те имена, которые экспортируются. Локальное имя не подходит для связывания.

EXPORT_SYMBOL позволяет предоставить API для других модулей/кода. Модуль не загрузится, если он ожидает символ, а его нет в ядре.

Один модуль может использовать данные и функции, используемые в другом модуле. Модуль, использующий экспортируемое имя, связывается с этим именем по абсолютному (физическому) адресу (адресу в оперативной памяти).

Если модуль вернет -1 на init (md3) — ошибка инициализации модуля, он не будет загружен.

1. Пробуем загрузить сначала только module_2:

```
1     $ sudo insmod module_2.ko
2 insmod: ERROR: could not insert module module_2.ko: Unknown symbol in
      module
```

В журнале:

```
1 $ dmesg
2 [ 6987.204306] module_2: Unknown symbol module_1_data (err -2)
3 [ 6987.204337] module_2: Unknown symbol module_1_proc (err -2)
```

Ошибка загрузки.

2. Теперь в правильном порядке:

```
1     $ sudo insmod module_1.ko
2 $ sudo insmod module_2.ko
3 $ lsmod | head -1 && lsmod | grep module_
4 Module                  Size  Used by
5 module_2                16384  0
6 module_1                16384  1 module_2
```

На модуль module_1 ссылается некоторые другие модули или объекты ядра: 1 — число ссылок на модуль (один модуль ссылается на другой).

3. Пытаемся выгрузить:

```
1     $ sudo rmmod module_1
2 rmmod: ERROR: Module module_1 is in use by: module_2
```

До тех пор, пока число ссылок на любой модуль в системе не станет равно 0, модуль не может быть выгружен.

```
$ sudo rmmod module_2
$ sudo rmmod module_1
$ lsmod | head -1 && lsmod | grep module_
Module                  Size  Used by
```

4. Используем module_1_local

```
static char *module_1_local(void) { return module_1_data; }
```

Функция не объявлена как extern, не указано EXPORT_SYMBOL, поэтому ошибка компиляции. implicit declaration of function module_1_local.

5. Используем module_1_noexport

```
extern char *module_1_noexport(void) { return module_1_data; }
```

Не указан EXPORT_SYMBOL для функции module_1_noexport. Ошибка сборки ERROR: modpost: "md1_noexport"
<путь_к_файлу>/md2.ko undefined!.

1.6. 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.

1.7. Регистрация функций работы с файлами. Пример заполненной структуры

Существует 2 типа файлов — файл, к-ый лежит на диске и открытый файл. Открытый файл – файл, который открывает процесс

Кратко

struct file описыает открытый файл. В ядре имеется сист. табл. открытых файлов. Каждый процесс имеет собственную таблицу открытых файлов, дескрипторы которой ссылаются на дескрипторы в таблице открытых файлов.

Подробно

Если файл просто лежит на диске, то через дерево каталогов можно увидеть это.

Увидеть можно только подмонтированную ФС.

А есть открытые файлы — файлы, с которыми работают процессы. Только процесс может открыть файл.

struct file описывает открытые файлы, которые нужны процессу для выполнения действий.

В системе сушествует **одна** табл. открытых файлов.

struct file – дескриптор открытого файла.

Открыть файл может только процесс. Если файл открывается потоком, то он в итоге все равно открывается процессом (как ресурс). Ресурсами владеет процесс.

Таблицы открытых векторов

Помимо таблицы открытых файлов процесса (есть у каждого процесса), в системе есть одна таблица на все открытые файлы.

Причем в этой таблице на один и тот же файл (с одним и тем же inode) мб создано большое кол-во дескрипторов открытых файлов, т.к. один и тот же файл мб открыт много раз.

Каждое открытие файла с одним и тем же inode приведет к созданию дескриптора открытого файла.

При открытии файла его дескриптор добавляется:

1. в таблицу открытых файлов процесса (struct file_struct)

2. в системную таблицу открытых файлов

Каждый дескриптор struct file имеет поле f_pos, это приводит к гонкам. При работе с файлами это надо учитывать.

Один и тот же файл, открытый много раз без соотв. способов взаимоискл. будет атакован, что приведет к потере данных.

Гоники при разделении файлов – один и тот же файл мб открыт разными процессами.

Определение struct file

```
1   struct file {
2   struct path    f_path;
3   struct inode     *f_inode; /* cached value */
4   const struct file_operations  *f_op;
5         ...
6   atomic_long_t    f_count;// кол-во жестких ссылок
7   unsigned int     f_flags;
```

```
8     fmode_t         f_mode;
9     struct mutex    f_pos_lock;
10    loff_t          f_pos;
11    ...
12    struct address_space  *f_mapping;
13    ...
14 };
```

Как осуществляется отображение файла на физ. страницы?

дескриптор открытого файла имеет указатель на inode (файл на диске).

1.7.1. Регистрация функций для работы с файлами

Разработчики драйверов должны регистрировать свои ф-ции read/write

Зачем в UNIX/Linux все файл?

Для того, чтобы все действия свести к однотипным операциям (read/write) и не размножать эти действия, а именно свести к небольшому набору операций.

Для регистрации своих функций
read/write в драйверах используется 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 и на struct file)

С остальными функциями аналогично. struct proc_ops сделана, чтобы не вешаться на функции struct file_operations, которые используются драйверами. Функции struct file_operations настолько важны для работы системы, что их решили освободить от работы с ф.с. proc

1.8. Передача данных из пространства ядра в пространство пользователя и из пространства пользователя в пространство ядра. Примеры из лабораторный работ

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.

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     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 чтении/записи), релиз при закрытии (во время чтении/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.