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). 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 int func_init(void);
```

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`.

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 void func_exit(void);
```

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

```
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.)

```
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("Inspirate789");
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 (тревога, 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 оперируют виртуальными адресами (процессы имеют виртуальное адресное пространство), а ядро - физическими.

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 module_2, использующий экспортируемое имя, связывается с этим именем по абсолютному (физическому) адресу (адресу в оперативной памяти). Модуль может использовать только экспортируемые имена.

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:

```
$ sudo rmmod module_1
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

ЗДЕСЬ ССЫЛКА НА 14