

```
struct proc_dir_entry *proc_create  
(const char *name, umode_t mode,  
struct proc_dir_entry *parent,  
const struct file_operations *proc_ops)  
proc_ops в версии ядра 5.16+
```

Ядро предоставляет возможность замены станд. ф-ций на свои:

Также ф-ции наз. hook и явл. точками входа, т.е. могут явл. точкой начала выполнения кода модуля ядра;

Сама ОС имеет много точек входа (сист. вызовы, исключения, аппаратные прерывания), но в каждом случае вызываются разные коды ядра (сист. вызов \rightarrow ф-ция ядра, т.е. интерфейс между kernel и user mode);

Ядро - хорошая школа для программиста;

В коде загр. модуля ядра можно исп. условную компиляцию:

```

#ifdef HAVE_PROC_OPS
static struct proc_ops file_ops =
{
    .proc_read = procfs_read,
    .proc_write = procfs_write,
    .proc_open = procfs_open,
    .proc_release = procfs_close
};
#else
static const struct file_operations file_ops =
{
    .read = procfs_read,
    .write = procfs_write,
    .open = procfs_open,
    .release = procfs_close
};

```

proc_open и open имеют одни и те же операционные параметры (указатель на struct inode и на struct file);

С остальными функциями аналогично;

Зачем так сделано? (~ 2 года назад)
 struct proc_ops сделана для того, 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 в.ф.с. proc;

Основная задача в. ф. с. прос - предоставление информации процессам о занимаемых ими ресурсах;

Подвижение struct proc_ops объясняется стремлением освободить ф-ции работы с файлами (struct file_operations) от работы с в. ф. с. прос;

У библиотек есть иное... интересный разгон, но продолжать не стали

Из библиотек можно создать модуль инструментальной среды, а сами языки очень маленькие

Что делать в л/р?

В курсовой передача kernel → user-область

2 функции:

- 1) Старый интерфейс (copy_to_user, copy_from_user);
- 2) см. glibc;

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

-- (по изуморичку) „апро галтно блее присталь-

ное внимание обратить на соотв. ф-цию";

- цитата К.Ю., но за такие слова не побьют?)

В ядре нельзя исп-ть станд. библиотеки;

`sprintf(char *str, ...)` - ф-ция ядра;

Название совпадает с названием из станд. библиотеки;

Такое сходство/совпадение - не редкость;

Ядро работает с физ. памятью

Не все задачи можно решить с помощью загр. модулей ядра, иногда его приходится перекомпилировать;

Ядро работает с физ. памятью, а у процессов адр. пр-во виртуальное. Это абстракция системы, создаваемая с помощью таблиц страниц;

Фреймы (физ. страницы) выделяются по прерываниям;

LRU

Может оказаться, что буфер, в который ядро пытается записать данные из буфера ядра,

чтобы передать их приложению, выгружен;

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

Это вероятностные вещи. Т.к. ядро работает с физ. адр. пр-вом, а приложения имеют вирт. адр. пр-ва, то нужен спец. ф-ция ядра;

Зон. модуль ядра - многоходовые программы. Все (.) входы всегда обязательны: `init` и `exit`;

Некоторые (.) входы могут вызываться из `init`, тогда их можно назвать (.) входы с инициализацией;

Прокорреспондирование названий `close` и `release`:
- `close` - общепринятое название библиотечных функций. Явл. одной из (.) входов модуля ядра;

- `release` - встречается в ядре;

Как создать виртуальные файлы:

Чтобы иметь возможность получать информацию из `user mode` и посылать информацию в `user mode`, необходимо создать файл.

в. ф. с. `proc`:

```

struct proc_dir_entry *my_proc_file; // дескриптор файла - не int
static int __init my_init(void)
{
    // файл можно создать с помощью proc_create_data
    my_proc_file = proc_create("my_file", 0644, NULL,
        &file_ops);
    ...
}
static void __exit my_exit(void)
{
    remove_proc_entry("my_file", NULL);
    ...
}

```

ан. методичку
(где будет создан файл proc)

Так создаётся файл proc;
Кроме того, надо создать директорию
и симв. ссылку:

```

extern struct proc_dir_entry *proc_mkdir(const char *,  
    struct proc_dir_entry *); // mkdir
// родительская директория
extern struct proc_dir_entry *proc_symlink(
    const char *, struct proc_dir_entry, const char *);

```

Симв. ссылка - файл спец. типа (L),
содержащая путь к файлу;

Часть точки входа procfs_write:

```

static char procfs_buffer[PROCFS_MAX_SIZE];
static ssize_t procfs_write(struct file *file,
    const char __user *buffer, size_t len,
    loff_t *off)
{
    ...
    copy_from_user(procfs_buffer, buffer,
        procfs_buffer_size);
    ...
}

```

Такие программы наз. „^(fortune)формулами“;

2 программа лабы:

Интерфейс sequence file;
 (более совр. интерфейсы для передачи
 kernel → user, но не наоборот)

Что можно передать полезного из user в kernel?
 Например, с помощью передачи из user mode
 выбрать режим работы загр. модуля ядра
 (какую информацию хотим получить из загр.
 модуля ядра в данный момент);

Такое „меню“ надо писать в user mode
 и передавать соотв. запросы модулям ядра;

```
struct seq_operations;  
struct seq_file  
{
```

```
    char *buf;
```

```
    size_t size;
```

```
    size_t from;
```

```
    size_t count;
```

```
    loff_t index;
```

```
    loff_t read_pos;
```

```
    struct mutex lock;
```

```
    const struct seq_operations *op;
```

```
    int poll_event;
```

```
    const struct file *file;
```

```
    void *private;
```

```
};  
struct seq_operations  
{
```

```
    void (*start)(struct seq_file *m, loff_t *pos);
```

```
    void (*stop)(struct seq_file *m, void *v);
```

```
    void *(*next)(struct seq_file *m, void *v, loff_t *pos);
```

```
    int (*show)(struct seq_file *m, void *v);
```

```
};  
↳ выполняем все действия по передаче данных;
```

```
user  $\xrightarrow{\text{write}}$  kernel  
kernel  $\xleftarrow{\text{read}}$  user
```