Интерфейс /proc

Интерфейс к файловым именам `/proc` (`procfs`) и более поздний интерфейс к именам `/sys` (`sysfs`) рассматривается как канал передачи диагностической (из) и управляющей (в) информации для модуля. Такой способ взаимодействия с модулем может полностью заменить средства вызова `ioctl()` для устройств, который устаревший и считается опасным. В настоящее время сложилась тенденция многие управляющие функции переносить их `/proc` в `/sys`, отображения путевых имён модулем в эти две подсистемы по своему назначению и возможностям являются очень подобными. Содержимое имён-псевдофайлов в обоих системах является только **текстовым** отображением некоторых внутренних данных ядра. Но нужно иметь в виду и ряд отличий между ними:

- Файловая система `/proc` является общей, «родовой» принадлежностью всех UNIX систем (Free/Open/Net BSD, Solaris, QNX, MINIX 3, ...), её наличие и общие принципы использования оговариваются стандартом POSIX 2; а файловая система `/sys` является сугубо Linux «изоретением» и используется только этой системой.
- Так сложилось по традиции, что немногочисленные диагностические файлы в `/proc` содержат зачастую большие таблицы текстовой информации, в то время, как в `/sys` создаётся много больше по числу имён, но каждое из них даёт только информацию об ограниченном значении, часто соответствующем одной элементарной переменной языка C: `int, long, ...`

Сравним:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 14
model name      : Genuine Intel(R) CPU           T2300  @ 1.66GHz
stepping : 8
cpu MHz         : 1000.000
...
$ wc -l cpuinfo
58 cpuinfo
```

- это 58 строк текста. А вот образец информации (выбранной достаточно наугад) системы `/sys`:

```
$ tree /sys/module/cpufreq
/sys/module/cpufreq
└── parameters
    ├── debug
    └── debug_ratelimit
1 directory, 2 files
$ cat /sys/module/cpufreq/parameters/debug
0
$ cat /sys/module/cpufreq/parameters/debug_ratelimit
1
```

Различия в форматном представлении информации, часто используемой в той или иной файловой системе, породили заблуждение (мне приходилось не раз это слышать), что интерфейс в `/proc` создаётся только для чтения, а интерфейс `/sys` для чтения и записи. Это совершенно неверно, оба интерфейса допускают и чтение и запись.

Теперь, когда мы кратно пробежались на качественном уровне по свойствам интерфейсов, можно перейти к примерам кода модулей, реализующих первый из этих интерфейсов. Интерфейс `/proc` рассматривается на примерах из архива `proc.tgz`. Мы будем собирать несколько однотипных модулей, поэтому общую часть определений снесём в отдельный файл:

mod_proc.h :

```
#include <linux/module.h>
```

```
#include <linux/proc_fs.h>
#include <linux/stat.h>
#include <asm/uaccess.h>

MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );

static int  __init proc_init( void );      // предварительные определения
static void __exit proc_exit( void );
module_init( proc_init );
module_exit( proc_exit );

#define NAME_DIR  "mod_dir"
#define NAME_NODE "mod_node"
#define LEN_MSG 160                        // длина буфера и сам буфер обмена
static char buf_msg[ LEN_MSG + 1 ] = "Hello from module!";
```

Файл сборки общий для всех модулей :

Makefile :

```
CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)
DEST = /lib/modules/$(CURRENT)/misc
EXTRA_CFLAGS += -std=gnu99

TARGET1 = mod_procr
TARGET2 = mod_procr2
TARGET3 = mod_proc
TARGET4 = mod_proct
obj-m   := $(TARGET1).o $(TARGET2).o $(TARGET3).o $(TARGET4).o

default:
        $(MAKE) -C $(KDIR) M=$(PWD) modules
...
```

Основную работу по созданию и уничтожению имени в `/proc` выполняет пара вызовов (`<linux/proc_fs.h>`):

```
struct proc_dir_entry *create_proc_entry( const char *name, mode_t mode,
                                          struct proc_dir_entry *parent );
void remove_proc_entry( const char *name, struct proc_dir_entry *parent );
```

В результате создаётся изрядно сложная структура, в которой нас могут интересовать, в первую очередь, поля:

```
struct proc_dir_entry {
...
        const char *name;
        mode_t mode;
...
        uid_t uid;
        gid_t gid;
...
        const struct file_operations *proc_fops;
...
        read_proc_t *read_proc;
        write_proc_t *write_proc;
...
};
```

Смысл всех этих полей станет понятным без объяснений из рассмотрения примеров построения модулей.

Первый пример (архив `proc.tgz`) показывает создание интерфейса к модулю в `/proc` доступного только для чтения из пользовательских программ (наиболее частый случай):

mod_procr.c :

```
        #include "mod_proc.h"    //  в  точности  списан  прототип  read_proc_t  из
<linux/proc_fs.h> :
```

```
        ssize_t proc_node_read( char *buffer, char **start, off_t off,
                                int count, int *eof, void *data ) {
           static int offset = 0, i;
           printk( KERN_INFO "read: %d\n", count );
           for( i = 0; offset <= LEN_MSG && '\0' != buf_msg[ offset ]; offset++, i++ )
              *( buffer + i ) = buf_msg[ offset ];        // buffer не в пространстве
пользователя!
           *( buffer + i ) = '\n';                         // дополним переводом строки
           i++;
           if( offset >= LEN_MSG || '\0' == buf_msg[ offset ] ) {
              offset = 0;
              *eof = 1;                                    // возвращаем признак EOF
           }
           else *eof = 0;
           printk( KERN_INFO "return bytes: %d\n", i );
           if( *eof != 0 ) printk( KERN_INFO "EOF\n" );
           return i;
        };
        // в литературе утверждается, что для /proc нет API записи, аналогично API
чтения,
        // но сейчас в <linux/proc_fs.h> есть описание типа (аналогичного типу
read_proc_t)
        // typedef int (write_proc_t)( struct file *file, const char __user *buffer,
        //                             unsigned long count, void *data );

        static int __init proc_init( void ) {
           int ret;
           struct proc_dir_entry *own_proc_node;
           own_proc_node = create_proc_entry( NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO,
NULL );
           if( NULL == own_proc_node ) {
              ret = -ENOMEM;
              printk( KERN_ERR "can't create /proc/%s\n", NAME_NODE );
              goto err_node;
           }
           own_proc_node->uid = 0;
           own_proc_node->gid = 0;
           own_proc_node->read_proc = proc_node_read;
           printk( KERN_INFO "module : success!\n");
           return 0;
                     err_node:                         // обычная для модулей практика
использования goto по ошибке
                                                            return                ret;
}

        static void __exit proc_exit( void ) {
           remove_proc_entry( NAME_NODE, NULL );
           printk( KERN_INFO "/proc/%s removed\n", NAME_NODE );
        }
```

Здесь и далее, флаги прав доступа к файлу вида `S_I*` - ищите и заимствуйте в `<linux/stat.h>`.

Испытания:

```
$ make
...
$ sudo insmod ./mod_procr.ko
$ dmesg | tail -n1
module : success!
$ ls -l /proc/mod_*
-r--r--r-- 1 root root 0 Map 26 18:14 /proc/mod_node
$ cat /proc/mod_node
Hello from module!
$ dmesg | tail -n7
 module : success!
 read: 3072
 return bytes: 19
 EOF
```

```
read: 3072
return bytes: 19
EOF
```

Примечание: *Обратите внимание на характерную длину блока чтения в этой реализации, она будет отличаться в последующих реализациях.*

Несколько последовательно выполняемых операций:

```
$ cat /proc/mod_node
Hello from module!
$ cat /proc/mod_node
Hello from module!
$ cat /proc/mod_node
Hello from module!
$ sudo rmmod mod_procr
$ ls -l /proc/mod_*
ls: невозможно получить доступ к /proc/mod_*: Нет такого файла или
каталога
```

Второй пример делает то же самое, но более простым и более описанным в литературе способом `create_proc_read_entry()` (но этот способ просто скрывает суть происходящего, но делает в точности то же самое):

mod_procr2.c :

```
#include "mod_proc.h"
ssize_t proc_node_read( char *buffer, char **start, off_t off,
                        int count, int *eof, void *data ) {
// ... в точности то, что и в предыдущем случае ...
};

static int __init proc_init( void ) {
   if( create_proc_read_entry( NAME_NODE, 0, NULL, proc_node_read, NULL ) == 0
) {
      printk( KERN_ERR "can't create /proc/%s\n", NAME_NODE );
      return -ENOMEM;
   }
   printk( KERN_INFO "module : success!\n");
   return 0;
}

static void __exit proc_exit( void ) {
   remove_proc_entry( NAME_NODE, NULL );
   printk( KERN_INFO "/proc/%s removed\n", NAME_NODE );
}
```

Примечание *(важно!):* `create_proc_read_entry()` *пример того, что API ядра, доступный программисту,* ***намного*** *шире, чем список экспортируемых имён в* `/proc/kallsyms` *или* `/boot/System.map-2.6.*`*, это происходит за счёт множества* `inline` *определений (как и в этом случае):*

```
$ cat /proc/kallsyms | grep create_proc_
c0522237 T create_proc_entry
c0793101 T create_proc_profile
$ cat /proc/kallsyms | grep create_proc_read_entry
$
```

Смотрим файл определений `<linux/proc_fs.h>` *:*

```
static inline 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 ) {
...
```

```
}
```

Возвращаемся к испытаниям полученного модуля:

```
$ sudo insmod ./mod_procr2.ko
$ echo $?
0
$ cat /proc/mod_node
Hello from module!
$ cat /proc/mod_node
Hello from module!
$ sudo rmmod mod_procr2
$ cat /proc/mod_node
cat: /proc/mod_node: Нет такого файла или каталога
```

Третий пример показывает модуль, который создаёт имя в /proc, которое может и читаться и писаться; для этого используется не специальный вызов (типа read_proc_t), а структура указатели файловых операций в таблице операций (аналогично тому, как это делалось в драйверах интерфейса /dev):

mod_proc.c :

```
#include "mod_proc.h"

static ssize_t node_read( struct file *file, char *buf,
                          size_t count, loff_t *ppos ) {
   static int odd = 0;
   printk( KERN_INFO "read: %d\n", count );
   if( 0 == odd ) {
      int res = copy_to_user( (void*)buf, &buf_msg, strlen( buf_msg ) );
      odd = 1;
        put_user( '\n', buf + strlen( buf_msg ) );    // buf — это адресное
пространство пользователя
      res = strlen( buf_msg ) + 1;
      printk( KERN_INFO "return bytes :  %d\n", res );
      return res;
   }
   odd = 0;
   printk( KERN_INFO "EOF\n" );
   return 0;
}

static ssize_t node_write( struct file *file, const char *buf,
                           size_t count, loff_t *ppos ) {
   int res, len = count < LEN_MSG ? count : LEN_MSG;
   printk( KERN_INFO "write: %d\n", count );
   res = copy_from_user( &buf_msg, (void*)buf, len );
   if( '\n' == buf_msg[ len -1 ] ) buf_msg[ len -1 ] = '\0';
   else buf_msg[ len ] = '\0';
   printk( KERN_INFO "put bytes = %d\n", len );
   return len;
}

static const struct file_operations node_fops = {
   .owner = THIS_MODULE,
   .read  = node_read,
   .write  = node_write
};

static int __init proc_init( void ) {
   int ret;
   struct proc_dir_entry *own_proc_node;
    own_proc_node = create_proc_entry( NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO,
NULL );
   if( NULL == own_proc_node ) {
      ret = -ENOMEM;
      printk( KERN_ERR "can't create /proc/%s\n", NAME_NODE );
      goto err_node;
   }
```

```
    own_proc_node->uid = 0;
    own_proc_node->gid = 0;
    own_proc_node->proc_fops = &node_fops;
    printk( KERN_INFO "module : success!\n");
    return 0;
err_node:
    return ret;
}

static void __exit proc_exit( void ) {
    remove_proc_entry( NAME_NODE, NULL );
    printk(KERN_INFO "/proc/%s removed\n", NAME_NODE );
}
```

Обратите внимание, функция чтения `node_read()` в этом случае принципиально отличается от аналогичной функции с тем же именем в предыдущих примерах: не только своей реализацией, но и прототипом вызова, и тем, как она возвращает свои результаты.

Испытания того, что у нас получилось:

```
$ sudo insmod ./mod_proc.ko
$ ls -l /proc/mod_*
-rw-rw-rw- 1 root root 0 Июл 2 20:47 /proc/mod_node
$ dmesg | tail -n1
module : success!
$ cat /proc/mod_node
Hello from module!
$ echo новая строка > /proc/mod_node
$ cat /proc/mod_node
новая строка
$ cat /proc/mod_node
новая строка
$ dmesg | tail -n10
 write: 24
 put bytes = 24
 read: 32768
 return bytes :  24
 read: 32768
 EOF
 read: 32768
 return bytes :  24
 read: 32768
 EOF
$ sudo rmmod mod_proc
$ cat /proc/mod_node
cat: /proc/mod_node: Нет такого файла или каталога
```

Примечание: *Ещё раз обратите внимание на размер блока запроса на чтение (в системном журнале), и сравните с предыдущими случаями.*

Ну а если нам захочется создать в `/proc` не отдельное имя, а собственную иерархию имён? Как мы наблюдаем это, например, для системного каталога:

```
$ tree /proc/driver
/proc/driver
├── nvram
├── rtc
└── snd-page-alloc
0 directories, 3 files
```

Пожалуйста! Для этого придётся только слегка расширить функцию инициализации предыдущего модуля (ну, и привести ему в соответствие функцию выгрузки). Таким образом, по образу и подобию, вы можете создавать иерархию произвольной сложности и любой глубины вложенности (показана только изменённая часть предыдущего примера):

mod_proct.c :

```
...
```

```
static struct proc_dir_entry *own_proc_dir;

static int __init proc_init( void ) {
   int ret;
   struct proc_dir_entry *own_proc_node;
   own_proc_dir = create_proc_entry( NAME_DIR, S_IFDIR | S_IRWXUGO, NULL );
   if( NULL == own_proc_dir ) {
      ret = -ENOMEM;
      printk( KERN_ERR "can't create /proc/%s\n", NAME_DIR );
      goto err_dir;
   }
   own_proc_dir->uid = own_proc_dir->gid = 0;
    own_proc_node = create_proc_entry( NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO,
own_proc_dir );
   if( NULL == own_proc_node ) {
      ret = -ENOMEM;
      printk( KERN_ERR "can't create /proc/%s\n", NAME_NODE );
      goto err_node;
   }
   own_proc_node->uid = own_proc_node->gid = 0;
   own_proc_node->proc_fops = &node_fops;
   printk( KERN_INFO "module : success!\n");
   return 0;
err_node:
  remove_proc_entry( NAME_DIR, NULL  );
err_dir:
   return ret;
}

static void __exit proc_exit( void ) {
   remove_proc_entry( NAME_NODE, own_proc_dir );
   remove_proc_entry( NAME_DIR, NULL  );
   printk(KERN_INFO "/proc/%s removed\n", NAME_NODE );
}
```

Примечание: *Здесь любопытно обратить внимание на то, с какой лёгкостью имя в* `/proc` *создаётся то как каталог, то как терминальное имя (файл), в зависимости от выбора единственного бита в флагах создания:* `S_IFDIR` *или* `S_IFREG`*.*

Теперь смотрим что у нас получилось:

```
$ sudo insmod ./mod_proct.ko
$ cat /proc/modules | grep mod_
mod_proct 1454 0 - Live 0xf8722000
$ ls -l /proc/mod*
 -r--r--r-- 1 root root 0 Июл  2 23:24 /proc/modules
 /proc/mod_dir:
 итого 0
 -rw-rw-rw- 1 root root 0 Июл  2 23:24 mod_node
$ tree /proc/mod_dir
 /proc/mod_dir
 └── mod_node
 0 directories, 1 file
$ cat /proc/mod_dir/mod_node
Hello from module!
$ echo 'new string' > /proc/mod_dir/mod_node
$ cat /proc/mod_dir/mod_node
new string
$ sudo rmmod mod_proct
```