Модуль 5. Файловая система /proc

Особенность файловой системы /proc

Изначально файловая система /proc разрабатывалась как средство предоставления информации о выполняющихся в системе процессах. Но из-за ее удобства многие подсистемы ядра стали использовать эту файловую систему как средство предоставления информации и динамического конфигурирования.

Файловая система /proc содержит каталоги (для структурирования информации) и виртуальные файлы. Виртуальный файл, может предоставлять пользователю информацию, полученную из ядра и, кроме того, служить средством передачи в ядро пользовательской информации. На самом деле, виртуальный файл не обязательно выполняет обе функции. На занятии мы рассмотрим, как настроить файловую систему как для ввода, так и для вывода.

Продемонстрируем несколько вариантов использования файловой системы /proc, дающих представление о ее возможностях. Выдайте в терминале команду `ls /proc`. Мы увидим содержание корневого каталога файловой системы /proc. Обратите внимание на файлы с номерными именами. Это - каталоги, содержащие информацию о выполняющихся в системе процессах. Process-id равный 1 присвоен процессу init, который в системе GNU/Linux запускается первым. Давайте выполним команду ls для этого каталога. Будет отображен список находящихся в нем файлов. В каждом файле содержатся те или иные сведения о процессе. Например, для того, чтобы посмотреть сведения о параметрах командной строки, с которыми был запущен процесс init, достаточно просмотреть содержимое файла cmdline с помощью команды cat.

В /proc есть и другие интересные файлы. Например, cpuinfo, содержащий сведения о типе и производительности центрального процессора, pci, из которого можно получить информацию об устройствах на шине PCI и modules, в котором находится список загруженных в ядро модулей.

В следующем листинге показан пример чтения и записи параметров ядра в виртуальный файл, находящийся в /proc. Приведенный пример кода отображает значение параметра, управляющего режимом "IP forwarding" стека TCP/IP ядра и затем включает его.

```
[root@plato]# cat /proc/sys/net/ipv4/ip_forward
0
```

```
[root@plato]# echo "1" > /proc/sys/net/ipv4/ip_forward
[root@plato]# cat /proc/sys/net/ipv4/ip_forward
1
```

Другим способом изменения параметров конфигурации ядра является использование команды sysctl.

На самом деле, /proc - не единственная виртуальная файловая система в ОС Linux. Аналогичная файловая система sysfs имеет сходные функциональные возможности и немного более удачную структуру (при ее разработке был учтен опыт /proc). Тем не менее /proc является де-факто стандартом и, несмотря на то, что sysfs имеет некоторые преимущества, будет и впредь оставаться таковым. Можно упомянуть еще одну виртуальную файловую систему - debugfs, которая (как следует из ее названия), представляет собой скорее отладочный интерфейс. Ее преимуществом является простота, с которой происходит экспорт значения из ядра в пользовательское пространство (фактически, это требует единственного системного вызова).

Пример драйвера, создающего файл в /proc с доступом на чтение

Пример драйвера, создающего файл в /proc с доступом на чтение

Разработчики загружаемых модулей ядра могут использовать все интерфейсы, доступные их коллегам, разрабатывающим само ядро. Допускается даже использование ядром переменных и функций, экспортируемых загружаемым модулем.

Создание и удаление виртуального файла в /proc

Для того, чтобы создать виртуальный файл в файловой системе /proc, используется функция create_proc_entry. Эта функция принимает в качестве параметров имя создаваемого файла, режим доступа к нему и один из подкаталогов /proc для его размещения. Функция create_proc_entry возвращает указатель на структуру proc_dir_entry (или NULL в случае возникновения ошибки). Полученный указатель можно использовать для настройки остальных параметров виртуального файла, таких, как функция, вызываемая при чтении из файла. Прототип функции create_proc_entry и фрагмент структуры proc_dir_entry показаны в листинге ниже.

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

struct proc_dir_entry {
    const char *name;          // имя виртуального файла
    mode_t mode;               // режим доступа
    uid_t uid;                 // уникальный номер
                               // пользователя -
                               // владельца файла
    uid_t uid;                 // уникальный номер группы,
                               // которой принадлежит файл
    struct inode_operations *proc_iops;
                // функции-обработчики операций с inode
    struct inode_operations *proc_iops;
                // функции-обработчики операций с файлом
    struct proc_dir_entry *parent;
                // Родительский каталог
    ...
    read_proc_t *read_proc;
                // функция чтения из /proc
    write_proc_t *write_proc;
                // функция записи в /proc
```

```
    void *data;
                // Указатель на локальные данные
    atomic_t count;
                // счетчик ссылок на файл
    ...
};

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

Чуть позже мы расммотрим, как использовать команды read_proc and write_proc для задания функций чтения и записи виртуального файла.

Для удаления файла из /proc, используйте функцию remove_proc_entry. При вызове в эту функцию передается строка, содержащая имя удаляемого файла и его местонахождение в файловой системе /proc (родительский каталог).

Параметр parent принимает значение NULL если файл находится непосредственно в каталоге /proc или другое значение, соответствующее каталогу, в который вы хотите поместить файл. В таблице приведена часть предопределенных переменных proc_dir_entry, передаваемых как значение параметра parent, и соответствующих им каталогов файловой системы /proc.

Переменная proc_dir_entry	Каталог
proc_root_fs	/proc
proc_net	/proc/net
proc_bus	/proc/bus
proc_root_driver	/proc/driver

Callback-функция записи

Вы можете записывать данные в виртуальный файл (из пользовательского пространства в ядро) с помощью функции write_proc. Эта функция имеет прототип следующего вида:

```
int mod_write( struct file *filp, const char __user
*buff, unsigned long len, void *data );
```

Параметр filp представляет собой структуру, соответствующую открытому файлу устройства. Параметр buff соответствует строке, передаваемой в модуль. Поскольку буфер, в котором находится строка находится в пользовательском пространстве, к нему нельзя будет получить непосредственный доступ из модуля. Параметр len содержит количество подлежащих записи байт, находящихся в buff. Параметр

data содержит указатель на локальные данные. В нашем тестовом модуле callback-функция записи служит для обработки входящих данных.

В Linux предусмотрен набор API для перемещения данных между пользовательским пространством и пространством ядра. Для операций с данными, находящимися в пользовательском пространстве, в функции write_proc из нашего примера, используется семейство функций copy_from_user.

Callback-функция чтения

Вы можете считать данные из виртуального файла (из ядра в пользовательское пространство) с помощью функции read_proc. Ее прототип выглядит так:

```
int mod_read( char *page, char **start, off_t off, int
count, int *eof, void *data );
```

Параметр page содержит указатель на буфер, в который будут записаны данные, полученные из ядра, при этом параметр count определяет максимальное число символов, которое может быть записано в данный буфер. Если планируется получить более одной страницы данных (обычно, 4KB), следует использовать параметры start и off. После того, как все данные получены, установите признак eof (конец файла). По аналогии с кодом функции write, параметр data соответствует локальным данным. Буфер page, используемый в данной функции располагается в пространстве ядра. Следовательно, для записи в него не требуется вызов функции copy_to_user.

Другие полезные функции

Кроме обыкновенных файлов, в файловой системе /proc можно создавать каталоги, используя функцию proc_mkdir и символьные ссылки (symlinks), используя proc_symlink. Файлы /proc, для которых определена только операция чтения (функция read), можно создать единственным вызовом функции create_proc_read_entry, создающей файл и задающей для него функцию read_proc. Прототипы вышеупомянутых функций показаны в листинге.

```
/* Создание каталога в файловой системе proc */
struct proc_dir_entry *proc_mkdir( const char *name,
struct proc_dir_entry *parent );


/* Создание символической ссылки в файловой системе proc
*/
struct proc_dir_entry *proc_symlink(   const char *name,
                                    struct proc_dir_entry
*parent,
                                    const char *dest );
```

```
/* Создание proc_dir_entry с read_proc_t за один вызов */
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 );

/* Копирование буфера из пространства ядра в
пользовательское пространство */
unsigned long copy_to_user( void __user *to,
                            const void *from,
                            unsigned long n );

/* Копирование буфера из пользовательского пространства в
пространство ядра */
unsigned long copy_from_user( void *to,
                              const void __user *from,
                              unsigned long n );

/* Выделение 'виртуально' непрерывного блока памяти */
void *vmalloc( unsigned long size );

/* Освобождение блока памяти, выделенного функцией
vmalloc */
void vfree( void *addr );

/* Экспорт символа в ядро (после этого ядро сможет его
видеть) */
EXPORT_SYMBOL( symbol );

/* Экспорт в ядро всех символов, объявленных в файле
(должен предшествовать подключению
файла module.h) */

EXPORT_SYMTAB
```

Выдача 'фортунок' с помощью файловой системы /proc

В этом примере мы создадим загружаемый модуль ядра с поддержкой операций чтения и записи. Это простое приложение будет по запросу выдавать изречения-'фортунки'. После загрузки модуля, пользователь сможет записать в него текст 'фортунок' с помощью команды echo и затем считывать их по одной в случайном порядке, используя команду cat.

Исходный код данного модуля приведен в листинге ниже. Init-функция (init_fortune_module) выделяет блок памяти для хранения 'фортунок' вызовом vmalloc и затем заполняет его нулями с помощью memset. После того, как cookie_pot создан и обнулен, в каталоге /proc создается виртуальный файл (тип proc_dir_entry) с именем fortune. После того, как файл (proc_entry) успешно создан, происходит инициализация локальных переменных и структуры proc_entry. В соответствующие поля этой структуры записываются указатели на функции модуля read и write (см. листинги 9 и 10, а также информация о владельце модуля. Функция cleanup удаляет файл из файловой системы /proc и освобождает память, занимаемую cookie_pot.

Хранилище 'фортунок' cookie_pot занимает страницу памяти (4KB) и обслуживается двумя индексами. Первый из них, cookie_index, определяет адрес, по которому будет записана следующая 'фортунка'. торой индекс - переменная next_fortune, содержит адрес 'фортунки', которая будет выдана по следующему запросу. После того как выдана последняя 'фортунка', переменной next_fortune присваивается адрес первого элемента и выдача начинается сначала.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/string.h>
#include <linux/vmalloc.h>
#include <asm/uaccess.h>
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Fortune Cookie Kernel Module");
MODULE_AUTHOR("M. Tim Jones");
#define MAX_COOKIE_LENGTH       PAGE_SIZE
static struct proc_dir_entry *proc_entry;
static char *cookie_pot;  // Хранилище 'фортунок'
static int cookie_index;  // Индекс первого свободного
для записи элемента хранилища
static int cookie_index;
        // Индекс элемента хранилища, содержащего
        // следующую 'фортунку' для вывода по запросу
int init_fortune_module( void )
{
     int ret = 0;
```

```
    cookie_pot = (char *)vmalloc( MAX_COOKIE_LENGTH );
    if (!cookie_pot) {
        ret = -ENOMEM;
    } else {
        memset( cookie_pot, 0, MAX_COOKIE_LENGTH );
        proc_entry = create_proc_entry( "fortune", 0644,
NULL );
        if (proc_entry == NULL) {
            ret = -ENOMEM;
            vfree(cookie_pot);
            printk(KERN_INFO "fortune: Couldn't create
proc entry\n");
        } else {
            cookie_index = 0;
            next_fortune = 0;
            proc_entry->read_proc = fortune_read;
            proc_entry->write_proc = fortune_write;
            proc_entry->owner = THIS_MODULE;
            printk(KERN_INFO "fortune: Module loaded.
\n");
        }
    }
    return ret;
}

void cleanup_fortune_module( void )
{
    remove_proc_entry("fortune", &proc_root);
    vfree(cookie_pot);
    printk(KERN_INFO "fortune: Module unloaded.\n");
}
module_init( init_fortune_module );
module_exit( cleanup_fortune_module );
```

Записать 'фортунку' в хранилище очень просто. Зная длину записываемой 'фортунки', можно определить, достаточно ли места для ее размещения. Если места недостаточно, модуль возвратит пользовательскому процессу код -ENOSPC. В противном случае строка копируется в cookie_pot с помощью функции copy_from_user. После этого происходит увеличение значения переменной cookie_index (на величину, зависящую от длины полученной строки), в конец строки дописывается NULL. Алгоритм завершает свою работу тем, что возвращает пользовательскому процессу количество символов фактически записанных в cookie_pot.

Листинг 10. Функция записи 'фортунки'

```
ssize_t fortune_write( struct file *filp, const char
__user *buff,unsigned long len, void *data )

{
    int space_available = (MAX_COOKIE_LENGTH-
cookie_index)+1;
    if (len > space_available) {
        printk(KERN_INFO "fortune: cookie pot is full!
\n");
        return -ENOSPC;
    }
    if (copy_from_user( &cookie_pot[cookie_index], buff,
len )) {
        return -EFAULT;
    }
    cookie_index += len;
    cookie_pot[cookie_index-1] = 0;
    return len;
}
```

Чтение 'фортунки' нисколько не сложнее ее записи. Поскольку буфер, в который нужно произвести запись 'фортунки' (page), уже находится в пользовательском пространстве, для вывода фортунки можно использовать непосредственно функцию sprintf. Если значение индекса next_fortune превышает значение cookie_index (индекс следующего свободного для записи элемента), переменной next_fortune присваивается 0, то есть, индекс первого элемента. После того, как фортунка записана в буфер пользовательского процесса, я увеличиваю индекс next_fortune на ее длину. Теперь этот индекс содержит адрес 'фортунки', которая будет выдана следующей . Длина 'фортунки' также передается пользовательскому процессу в качестве возвращаемого значения.

Листинг 11. Функция чтения 'фортунки'

```
int fortune_read( char *page, char **start, off_t off,
        int count, int *eof, void *data )
{
    int len;
    if (off > 0) {
        *eof = 1;
        return 0;
    }
    /* Перевод индекса на первый элемент */
    if (next_fortune >= cookie_index) next_fortune = 0;
    len = sprintf(page, "%s\n", &cookie_pot[next_fortune]);
```

```
        next_fortune += len;
        return len;
}
```

Это простой пример показывает что обмен данными между ядром и пользовательским процессом является тривиальной задачей. А сейчас предлагаю посмотреть на модуль выдачи 'фортунок' в действии.

```
[root@plato]# insmod fortune.ko
[root@plato]# echo "Success is an individual proposition.
Thomas Watson" > /proc/fortune
[root@plato]# echo "If a man does his best, what else is
there?
Gen. Patton" > /proc/fortune
[root@plato]# echo "Cats: All your base are belong to us.
Zero Wing" > /proc/fortune
[root@plato]# cat /proc/fortune
Success is an individual proposition.  Thomas Watson
[root@plato]# cat /proc/fortune
If a man does his best, what else is there?  Gen. Patton
```

Виртуальная файловая система /proc широко используется как средство сбора информации о состоянии ядра и для его динамического конфигурирования. Она незаменима для разработки драйверов и модулей ядра, в чем несложно убедиться.

Блокировка процессов при конкурентном доступе к устройству
Пример драйвера, приостанавливающего работу процессов

Что вы делаете, когда кто-то просит вас о чем-то, а вы не можете сделать это немедленно? Пожалуй единственное, что вы можете ответить: "Пожалуйста, не сейчас, я пока занят." А что должен делать модуль ядра? У него есть другая возможность. Он может приостановить работу процесса до тех пор, пока не сможет обслужить его. Пример sleep.c демонстрирует такую возможность. Модуль создает файл /proc/sleep, который может быть открыт только одним процессом в каждый конкретный момент времени. Если файл уже был открыт кем-нибудь, то модуль вызывает wait_event_interruptible. Эта функция изменяет состояние "задачи" (здесь, под термином "задача" понимается структура данных в ядре, которая хранит информацию о процессе), присваивая ей значение TASK_INTERRUPTIBLE, это означает, что задача не будет выполняться до тех пор, пока не будет "разбужена" каким-либо образом, и добавляет процесс в очередь ожидания WaitQ, куда помещаются все процессы, желающие открыть файл /proc/sleep. Затем функция передает управление планировщику, который в свою очередь предоставляет возможность поработать другому процессу.

```
/*
 *  sleep.c - Создает файл в /proc, доступ к
 *  которому может получить только один процесс,
 *  все остальные будут приостановлены.
 */

#include <linux/kernel.h>   /* Все-таки мы работаем с
ядром! */
#include <linux/module.h>   /* Необходимо для любого
модуля */
#include <linux/proc_fs.h>  /* Необходимо для работы с /
proc */
#include <linux/sched.h>    /* Взаимодействие с
планировщиком */
#include <asm/uaccess.h>    /* определение функций
get_user и put_user */

/*
 * Место хранения последнего принятого сообщения,
 * которое будет выводиться в файл, чтобы показать, что
```

```
 * модуль действительно может получать ввод от
пользователя
 */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];

static struct proc_dir_entry *Our_Proc_File;
#define PROC_ENTRY_FILENAME "sleep"

/* см. include/linux/fs.h   */
static ssize_t module_output(struct file *file,
/* буфер с данными (в пространстве пользователя) */
                      char *buf,
/* размер буфера */
                      size_t len,
                      loff_t * offset)
{
  static int finished = 0;
  int i;
  char message[MESSAGE_LENGTH + 30];

  /*
   * Для индикации признака конца файла возвращается 0.
   * В противном случае процесс будет продолжать читать
из файла
   * угодив в бесконечный цикл.
   */
        if (finished) {
                finished = 0;
                return 0;
        }

  /*
   * Для передачи данных из пространства ядра в
пространство пользователя
   * следует использовать put_user.
   * В обратном направлении -- get_user.
   */
  sprintf(message, "Last input:%s\n", Message);
  for (i = 0; i < len && message[i]; i++)
    put_user(message[i], buf + i);

  finished = 1;
  return i;    /* Вернуть количество "прочитанных" байт
*/
```

```
}

/*
 * Эта функция принимает введенное пользователем
сообщение
 */
static ssize_t module_input(struct file *file,  /*
Собственно файл */
          const char *buf,                      /* Буфер
с сообщением */
          size_t length,                        /* размер
буфера */
          loff_t * offset)
{         /* смещение в файле - игнорируется */
  int i;


  /*
   * Переместить данные, полученные от пользователя в
буфер,
   * который позднее будет выведен функцией
module_output.
   */
  for (i = 0; i < MESSAGE_LENGTH - 1 && i < length; i++)
    get_user(Message[i], buf + i);

  /* Обычная строка, завершающаяся символом \0 */
  Message[i] = '\0';

  /*
   * Вернуть число принятых байт
   */
  return i;
}

/*
 * 1 -- если файл открыт
 */
int Already_Open = 0;

/*
 * Очередь ожидания
 */
DECLARE_WAIT_QUEUE_HEAD(WaitQ);
/*
```

```
 * Вызывается при открытии файла в /proc
 */
static int module_open(struct inode *inode, struct file
*file)
{
  /*
   * Если установлен флаг O_NONBLOCK,
   * то процесс не должен приостанавливаться
   * В этом случае, если файл уже открыт,
   * необходимо вернуть код ошибки
   * -EAGAIN, что означает "попробуйте в другой раз"
   */
  if ((file->f_flags & O_NONBLOCK) && Already_Open)
    return -EAGAIN;

  /*
   * Нарастить счетчик обращений,
   * чтобы невозможно было выгрузить модуль
   */
  try_module_get(THIS_MODULE);

  /*
   * Если файл уже открыт -- приостановить процесс
   */

  while (Already_Open) {
    int i, is_sig = 0;

    /*
     * Эта функция приостановит процесс и поместит его в
очередь ожидания.
     * Исполнение процесса будет продолжено с точки,
следующей за вызовом
     * этой функции, когда кто нибудь сделает вызов
     * wake_up(&WaitQ) (это возможно только внутри
module_close, когда
     * файл будет закрыт) или когда процессу поступит
сигнал Ctrl-C
     */
    wait_event_interruptible(WaitQ, !Already_Open);

    for (i = 0; i < _NSIG_WORDS && !is_sig; i++)
    is_sig =
          current->pending.signal.sig[i] & ~current->
          blocked.sig[i];
```

```
    if (is_sig) {
      /*
       * Не забыть вызвать здесь module_put(THIS_MODULE),
       * поскольку процесс был прерван
       * и никогда не вызовет функцию close.
       * Если не уменьшить счетчик обращений, то он
навсегда останется
       * больше нуля, в результате модуль можно будет
       * уничтожить только при перезагрузке системы
       */
      module_put(THIS_MODULE);
      return -EINTR;
    }
  }

  /*
   * В этой точке переменная Already_Open должна быть
равна нулю
   */

  /*
   * Открыть файл
   */
  Already_Open = 1;
  return 0;
}

/*
 * Вызывается при закрытии файла
 */
int module_close(struct inode *inode, struct file *file)
{
  /*
   * Записать ноль в Already_Open, тогда один
   * из процессов из WaitQ
   * сможет записать туда единицу и открыть файл.
   * Все остальные процессы, ожидающие доступа
   * к файлу опять будут приостановлены
   */
  Already_Open = 0;

  /*
   * Возобновить работу процессов из WaitQ.
   */
```

```
  wake_up(&WaitQ);

  module_put(THIS_MODULE);

  return 0;
}


/*
 * Указатели на функции-обработчики для нашего файла.
 */
static struct file_operations File_Ops_4_Our_Proc_File =
{
  .read = module_output,    /* чтение из файла */
  .write = module_input,    /* запись в файл */
  .open = module_open,      /* открытие файла */
  .release = module_close,  /* закрытие файла */
};

/*
 * Начальная и конечная функции модуля
 */

/*
 * Инициализация модуля - регистрация файла в /proc
 */

int init_module()
{
  int rv = 0;
  Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME,
0644, NULL);
  Our_Proc_File->proc_fops = &File_Ops_4_Our_Proc_File;
  Our_Proc_File->mode = S_IFREG | S_IRUGO | S_IWUSR;
  Our_Proc_File->uid = 0;
  Our_Proc_File->gid = 0;
  Our_Proc_File->size = 80;

  if (Our_Proc_File == NULL) {
    rv = -ENOMEM;
    remove_proc_entry(PROC_ENTRY_FILENAME, NULL);
    printk(KERN_INFO "Error: Could not initialize /proc/
test\n");
  }
```

```
  return rv;
}



/*
 * Завершение работы модуля - дерегистрация файла в /
proc.
 * Чревато последствиями
 * если в WaitQ остаются процессы, ожидающие своей
очереди,
 * поскольку точка их исполнения
 * практически находится в функции open, которая
 * будет выгружена при удалении модуля.
*/
void cleanup_module()
{
  remove_proc_entry(PROC_ENTRY_FILENAME, NULL);
}
```