Методические указания по лабораторной работе (2 часть) «Виртуальная файловая система /proc»

1. Виртуальная файловая система /proc

Папки (каталоги, англ. directory) и файлы виртуальной файловой системы /proc не хранятся на диске. Они создаются динамически при обращении к ним. Linux предоставляет ядру и модулям ядра дополнительный механизм передачи информации заинтересованным в ней процессам - это файловая система `/proc`. Первоначально она создавалась с целью получения сведений о процессах (отсюда такое название). Теперь она интенсивно используется и самим ядром, которому есть что сообщить! Например, `/proc/modules` -- список загруженных модулей, `/proc/meminfo` -- статистика использования памяти.

Файловая система /proc фактически представляет собой интерфейс ядра, который позволяет получать информацию о процессах и ресурсах, которые они используют. При этом используется стандартный интерфейс файловой системы и системных вызовов. Из этого следует, что управление доступом к адресному пространству осуществляется при помощи обычных прав доступа – на чтение, на запись и выполнение.
Общее взаимодействие с proc заключается в использовании cat и echo из оболочки – командной строки. Например:

```
1  # cat / proc / cpuinfo
2  # echo " 50 " > /proc/sys/kernel / sched_rr_timeslice_ms
```

Данные о каждом процессе хранятся в поддиректории с именем, которым является идентификатор процесса: /proc/<PID>. В поддиректории процесса находятся файлы и поддиректории, содержащие данные о процессе (табл.1):

Таблица – файлы и поддиректории /proc/<PID>

Элемент	Тип	Содержание
cmdline	файл	Указывает на директорию процесса
cwd	символическая ссылка	Указывает на директорию процесса
environ	файл	Список окружения процесса
exe	символическая ссылка	Указывает на образ процесса (на его файл)
fd	директория	Ссылки на файлы, которые «открыл» процесс

root	символическая ссылка	Указывает на корень файловой системы процесса
stat	файл	Информация о процессе

Процесс может получить свой идентификатор с помощью функции getpid().

Другой способ – использовать ссылку self: /proc/self.

2. Загружаемы модули ядра и виртуальная файловая система /proc

Файлы и поддиректории файловой системы /proc могут создаваться, их можно регистрировать и прекращать их регистрацию. Поэтому /proc часто используются загружаемыми модулями ядра. Файлы и поддиректории файловой системы /proc используют структуру proc_dir_entry:

```
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 file_operations *proc_fops; // функции-
обработчики операций с файлом

    struct proc_dir_entry  *next, *parent,
*subdir;

    ...
```

```
    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

    ...

};
```

proc_dir_entry (9) (Linux man: Ядро)

```
struct proc_dir_entry {
    unsigned short low_ino;
    unsigned short namelen;
    const char *name;
    mode_t mode;
    nlink_t nlink;
    uid_t uid;
    gid_t gid;
    unsigned long size;
    struct inode_operations *proc_iops;
    struct file_operations  *proc_fops;
    struct proc_dir_entry *next, *parent, *subdir;
    void *data;
…..
};
```

ow_ino : номер inode для директории. Для **proc_register** этот номер должен быть уникальным в файловой системе /proc, значения определены в <*linux/proc_fs.h*>. Для **proc_register_dynamic** номер inode назначаются динамически.
namelen : длина имени
name : уникальное имя виртуального файла **(**имя данного узла).
mode : тип и права доступа к узлу.
The node's type and permissions. Взяты из <*linux/stat.h*>.
nlink : число линков к узлу. Инициализировать до 2, если режим включает S_IFDIR, 1 в противном случае.
uid : идентификатор пользователя **(**uid), которому принадлежит файл (узел), обычно 0.
gid : идентификатор группы (gid), которой принадлежит узел, обычно 0.

size : устанавливает размер узла, значение будет отображаться как размер inode в списках и будет возвращено stat. Если размер не нужен, то его устанавливают равным нулю.
ops : определяет набор операций inode для узла / proc. Для узла каталога, если нет особых требований, используются & proc_dir_inode_operations. Для листового узла, если нет специальных требований, устанавливается значение NULL.
data : непрозрачный указатель, который может использоваться обработчиками proc для передачи локальных данных. Допускается устанавливать свободно при вызове proc_register, обычно NULL. Этот указатель копируется в поле inode.de_generic inode (by proc_get_inode), поэтому он доступен для всех процедур proc, которые передаются inode.

3 . Файловая система `/proc`: создание файлов, доступных для чтения

Методика работы с файловой системой `/proc` очень похожа на работу драйверов с файлами устройств: создаётся структура со всей необходимой информацией, включая указатели на функции-обработчики (в нашем случае имеется только один обработчик, который обслуживает чтение файла в `/proc`). Функция `init_module()` регистрирует структуру, а `module_exit()` отменяет регистрацию (см. лабораторную работу «Загружаемые модули ядра»).

Основная причина, по которой используется `proc_register_dynamic` состоит в том, что номер inode, для создаваемого файла, заранее неизвестен, поэтому ядро может определить его самостоятельно, чтобы предотвратить возможные конфликты. В обычных файловых системах, размещенных на диске, не в памяти, как `/proc`, inode указывает на то место в дисковом пространстве, где размещена индексная запись (index node, сокращенно -- inode) о файле. Inode содержит все необходимые сведения о файле, например права доступа, указатель на первый блок с содержимым файла.

Начиная с ядра 3.10 больше не поддерживается функция create_proc_entry(). Вместо нее используются функции:

```
#include <linux/types.h>
#include <linux/fs.h>

extern struct proc_dir_entry *proc_create_data(const char *, umode_t, struct
                                    proc_dir_entry *, const struct file_operations *, void
                                    *);

struct proc_dir_entry *proc_create(const char *name, umode_t mode, struct
                            proc_dir_entry *parent, const struct file_operations *proc_fops);
```

В то время как отдельные элементы proc_dir_entry должны были быть инициализированы в некоторых более ранних версиях ядра, разработчики,

работающие с более новой версией ядра, используют структуру, которая хорошо знакома из разработки драйверов, struct file_operations, чтобы назначить методы доступа: open(), read(), write().

Структура file_operations используется для определения обратных вызовов (call back) чтения и записи.

Рассмотрим пример, который отображает текущее значение jiffies (счетчик тиков) всякий раз, когда он меняется.

Рассмотрим загружаемый модуль, в котором создается файл в файловой системе /proc:

```
1  #include <linux/module.h>
2  #include <linux/moduleparam.h>
3  #include <linux/init.h>
4  #include <linux/kernel.h>
5  #include <linux/proc_fs.h>
6  #include <asm/uaccess.h>
7  #define BUFSIZE 100
8
9
10 MODULE_LICENSE("Dual BSD/GPL");
11 MODULE_AUTHOR( "Liran B.H");
12
13
14 static struct proc_dir_entry * ent;
15
16 static ssize_t mywrite(struct file *file, const char __user *ubuf,size_t count, loff_t *ppos)
17 {
18          printk( KERN_ DEBUG "write handler\n");
19          return - 1;
20 }
21
22 static ssize_t myread(struct file *file, char __user *ubuf,size_t count, loff_t *ppos)
23 {
24          printk( KERN_ DEBUG "read handler\n");
25          return 0;
26 }
27
28 static struct file_operations myops =
29 {
30          .owner =  THIS_MODULE,
31          .read = myread,
32          .write = mywrite,
33 };
34
35 static int simple_init(void)
36 {
37          ent= proc_create("mydev", 0660, NULL, & myops);
38          return 0;
39 }
40
```

```
41 static void simple_cleanup( void)
42 {
43          proc_remove(ent);
44 }
45
46 module_init(simple_init);
47 module_exit(simple_cleanup);
```

Если откомпилировать и загрузить этот модуль, то будет создан новый файл /proc/mydev.

Можно протестировать функции read и write, используя cat и echo путем просмотра kernel log messages.

```
1 # echo "test" > /proc/mydev
2 bash: echo: write error: Operation not permitted
3
4 # cat /proc/mydev
5 # dmesg | tail -2
6 [  694.640306] write handler
7 [  714.661465] read handler
```

Родительский аргумент может быть NULL для корня /proc root или нескольких других значений, в зависимости от того, где нужно разместить файл. В таблице 2 перечислены некоторые другие родительские proc_dir_entrys, которые можно использовать, а также их расположение в файловой системе.

Table 2. Shortcut переменных proc_dir_entry

proc_dir_entry	Filesystem location
`proc_root_fs`	`/proc`
`proc_net`	`/proc/net`
`proc_bus`	`/proc/bus`
`proc_root_driver`	`/proc/driver`

Можно также создавать каталоги в файловой системе /proc, используя proc_mkdir(), а также символические ссылки с proc_symlink(). Для простых /proc-записей, для которых требуется только функция чтения, используется create_proc_read_entry(), которая создает запись /proc и инициализирует функцию read_proc в одном вызове. Прототипы этих функций:

```
extern struct proc_dir_entry *proc_symlink(const char *,
                                           struct proc_dir_entry *, const char *);

extern struct proc_dir_entry *proc_mkdir(const char *, struct proc_dir_entry *);
```

```
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 );
```

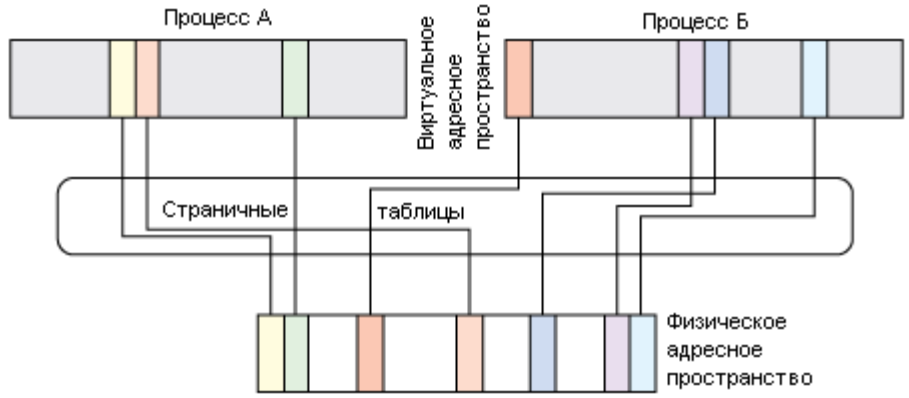
Передача данных из режима пользователя в режим ядра и из режима ядра в режим пользователя

Поскольку адресные пространства процессов являются виртуальными, в системе их может быть много. Фактически само ядро и каждый из процессов располагаются в своих собственных изолированных адресных пространствах. Эти адресные пространства состоят из адресов виртуальной памяти, что позволяет нескольким процессам с независимыми адресными пространствами обращаться к физическому адресному пространству (т. е. к памяти, физически установленной на устройстве), имеющему значительно меньший объем. Такой подход обеспечивает не только удобство, но и безопасность, поскольку все адресные пространства являются независимыми и изолированными друг от друга.

Однако такая безопасность не дается даром. Поскольку ядро и каждый из процессов могут иметь одинаковые адреса, ссылающиеся на различные области физической памяти, при использовании общей памяти могут возникать задержки. К счастью, существует несколько решений этой проблемы. Совместное использование памяти пользовательскими процессами может обеспечиваться с помощью POSIX-механизма совместно используемой памяти `shmem` (shared memory mechanism), причем процессы могут иметь различные виртуальные адреса, ссылающиеся на одну и ту же область физической памяти.

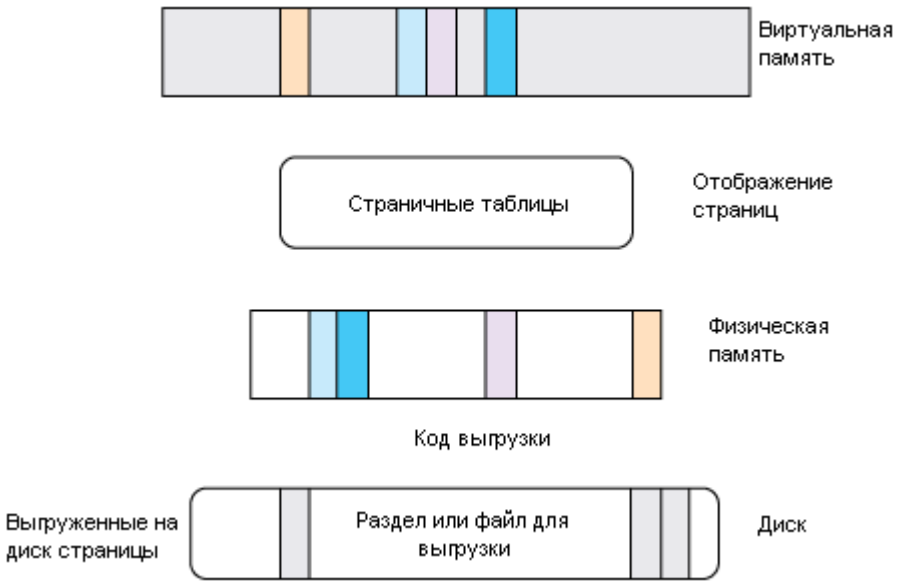
Отображение виртуальной памяти на физическую осуществляется через аппаратно реализованные страничные таблицы (рисунок 1). Аппаратная часть непосредственно выполняет отображение, а ядро управляет таблицами и их конфигурацией. Обратите внимание на то, что процесс может иметь большое, но не непрерывное, а «разбросанное» адресное пространство (как показано на рисунке); это означает, что небольшие области (страницы) адресного пространства отображаются на физическую память через таблицы страниц. Это позволяет процессу иметь большое виртуальное адресное пространство, но в физической памяти будут находиться только те страницы, с которыми процессор работает в текущий момент.

Рисунок 1. Страничные таблицы обеспечивают отображение виртуальных адресов на физические



Поскольку память может выделяться процессам только тогда, когда это необходимо (по запросу), объем используемой виртуальной памяти может быть больше, чем объем имеющейся в системе физической памяти. В процессе *замещения страниц* (paging), который в Linux называется *подкачкой* (swap), наиболее редко используемые страницы перемещаются на более медленное устройство хранения (например, на жесткий диск), что позволяет обеспечить доступ к страницам, с которыми необходимо работать в данный момент (рисунок 2). Данный подход позволяет размещать в ОЗУ наиболее часто используемые страницы, а наиболее редко используемые выгружать на жесткий диск. Таким образом, обеспечивается более эффективное использование физической памяти. Заметьте, что некоторые страницы могут ссылаться на файлы – в этом случае грязные (dirty) страницы могут быть вытеснены (replacement) на диск, а чистые (no edit) страницы быть просто помеченными как свободные (копирование на диск не выполняется).

Рисунок 2. Подкачка позволяет эффективнее использовать физическую память, перемещая редко используемые страницы на более медленные и дешевые устройства



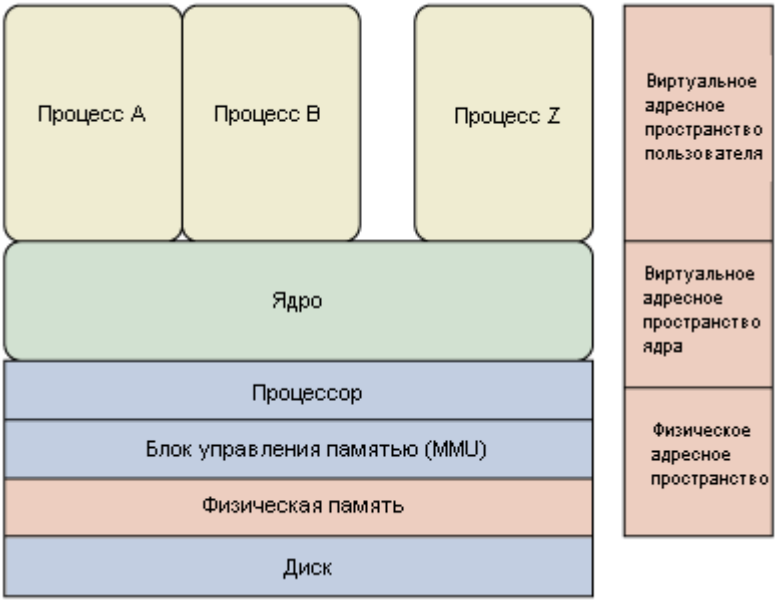
Процессоры без блока управления памятью (MMU)

Не все процессоры имеют блок управления памятью, поэтому дистрибутив uClinux (версия ОС Linux для микроконтроллеров) поддерживает одноадресное пространство операций. Архитектура без MMU не имеет защиты, обеспечиваемой MMU-блоком, но позволяет ОС Linux работать на процессорах с различными типами архитектур.

Процесс выбора страниц, которые должны быть выгружены на диск, называется *алгоритмом замещения страниц* и может быть реализован по-разному (например, существует алгоритм замещения блока данных с наиболее длительным отсутствием обращений). Этот процесс может начаться в момент обращения к адресу памяти, страница которого не находится в ОЗУ (page fault) (в блоке управления памятью отсутствует ее отображение). рис. 3.

В операционной системе Linux подкачка реализована интересным способом, обладающим рядом полезных свойств. Linux позволяет создавать и использовать несколько разделов подкачки и задавать приоритеты для устройств с различным быстродействием, на которые производится выгрузка данных (например, первоочередным устройством для выгрузки является твердотельный накопитель – SSD-диск, а второстепенным – более медленное устройство). Если SSD-накопителю назначен более высокий приоритет, выгрузка данных будет производиться на него до тех пор, пока на нем не закончится свободное место; лишь после этого страницы памяти будут выгружаться на более медленное устройство с меньшим приоритетом.

Рисунок 3. Адресные пространства и элементы отображения виртуальных адресов на физические



Не все страницы памяти могут быть выгружены (например, это может быть код ядра, отвечающий за прерывания, или код управления страничными таблицами и логикой подкачки). Очевидно, что такие страницы никогда не должны выгружаться и поэтому являются *закрепленными* (pinned), т. е. постоянно находятся в оперативной памяти. В отличие от страниц ядра, которые никогда не выгружаются, страницы адресного пространства пользователя могут быть выгружены, однако их можно закрепить с помощью

функции `mlock` (или `mlockall`). Рассмотрим следующую ситуацию: ядро предположило, что переданный пользователем адрес являлся допустимым и доступным, но по какой-либо причине возникло состояние отсутствия страницы в ядре (например, потому, что пользовательская страница была выгружена). В этом случае может наступить состояние паники ядра. API-функция mlock предназначена именно для того, чтобы обеспечивать корректную обработку таких ситуаций.

Для взаимодействия приложений с ядром и ядра с приложениями используются следующие функции:

- Функция (**User Space Memory Access)**

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

Формальные параметры

*void __user * to - Адрес назначения находится в пространстве пользователя.*

*const void * from - Адрес источника находится в пространстве ядра.*

unsigned long n - количество копируемых байт.

Контекст

Только пользовательский контекст. Эта функция может переводить процесс в состояние sleep, если включено pagefault.

Описание

Копирует данные из ядра в пространство пользователя. Вызывающий абонент должен проверить указанный блок с помощью `access_ok` до вызова этой функции.

Функция возвращает количество байт, которые не могут быть скопированы. В случае успешного выполнения будет возвращен 0.

- Функция (**User Space Memory Access)**

```
unsigned
long __copy_from_
user (
                        void * to,
                        const void __user * from,
                        unsigned long n);
```

Формальные параметры

*void * to - Адрес назначения находится в пространстве ядра.*

*const void __user * from - адрес источника находится в пространстве поль зователя.*

unsigned long n - количество копируемых байт.

Контекст

Только пользовательский контекст. Эта функция может переводить процесс в состояние sleep, если включено pagefault.

Описание

Копирует данные из пространства пользователя в пространство ядра. Вызывающий абонент должен проверить указанный заданный блок с помощью `access_ok` до вызова этой функции.

Функция возвращает количество байт, которые не могут быть скопированы. В случае успешного выполнения будет возвращен 0.

Если какие-то данные не могут быть скопированы, эта функция. Если некоторые данные не могут быть скопированы, эта функция добавит нулевые байты к скопированным данным до требуемого размера.

Альтернативная версия - __copy_from_user_inatomic - может быть вызвана из атомарного контекста и не сможет «спать». В этом случае не скопированные байты * НЕ * будут заполнены нулями. Смотрите fs / filemap.h для объяснения того, почему это необходимо.

Функция

int sprintf(char *_str_, const char *_format_, ...);

После успешного возврата эти функция возвращает количество напечатанных символов (исключая нулевой байт, используемый для завершения вывода в строки).
Если обнаружена ошибка вывода, возвращается отрицательное значение.
Строка формата - строка символов. Строка формата состоит из нуля или более директив: обычные символы (не %), которые копируются без изменений в выходной поток; и спецификации преобразования, каждая из которых приводит к получению нуля или более последующих аргументов.
Каждая спецификация преобразования вводится символом % и заканчивается спецификатором преобразования. Между ними может быть (в этом порядке) ноль или более флагов, необязательная минимальная ширина поля, необязательная точность и необязательный модификатор длины.

Так как **sprintf** получает строку произвольной длины, ее вызов может привести к переполнению буфера, что невозможно предотвратить. Обратите внимание, что длина строки определена как локально зависимая и ее трудно спрогнозировать. Используйте вместо них **snprintf**.

Перед использованием надо проверить возможность использования функций в конкретной версии ядра.

Данная функция может не поддерживаться разными версиями и подверсиями ядра Linux.

Задание на лабораторную работу

Написать программу – загружаемый модуль ядра (LKM) – которая поддерживает чтение из пространства пользователя и запись в пространство пользователя из пространства ядра.

После загрузки модуля пользователь должен иметь возможность загружать в него строки с помощью команды echo, а затем считывать их с помощью команды cat.

Функция init (например, init_fortune_module) выделяет пространство для «горшка с печеньем» (cookie pot) используя vmalloc(), а затем очищает его с помощью memset().

// **устаревшая версия – часто приводится на сайтах**

```
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 );//начиная с версии  //3.10 используется
 proc_create_data()
    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;
 ...
}
```

Когда cookie_pot (буфер, англ. Buffer) выделен и пуст, создается proc_dir_entry в корне /proc root, который называется, например, fortune.

Файл cookie_pot представляет собой страницу длиной (4 КБ) и управляется двумя индексами. Первый, cookie_index, определяет, где будет записываться следующий файл cookie. Переменная next_fortune определяет, где следующий файл cookie будет считаться для вывода. Я просто переношу next_fortune в начало, когда все состояния были прочитаны.

Запись нового файла cookie_pot: если буфер для записи не доступен, то возвращается -ENOSPC, который передается пользовательскому процессу. В противном случае это пространство существует, и используется функция copy_from_user() для копирования пользовательского буфера непосредственно в файл cookie_pot. Затем увеличивается файл cookie_index (в зависимости от длины пользовательского буфера) и NULL завершает строку. Наконец, возвращается количество символов, фактически записанных в файл cookie_pot.

```
ssize_t fortune_write( struct file *filp, const char
__user *buff,

                        unsigned long len, void *data )
{
    ...
if (copy_from_user( &cookie_pot[cookie_index], buff,
len )) {

    return -EFAULT;
  }
...
}
```

Чтение: поскольку объявляемый буфер уже находится в пространстве ядра, можно манипулировать им напрямую и использовать sprintf (может не поддерживаться ядром) или copy_to_user() для записи. Если индекс next_fortune больше, чем cookie_index (следующая позиция для записи), то next_fortune уменьшается до нуля, что зацикливает буфер. После того, как строка будет записана в пользовательский буфер, индекс next_fortune увеличивается на длину последней написанной строки.

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

```
{
  int len;
    ...
len = sprintf(page, "%s\n", &cookie_pot[next_fortune]);
  next_fortune += len;
  return len;
}
```

Листинг 1. Демонстрация работы «fortune cookie» LKM

[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

[root@plato]#

В программе необходимо создать файл, поддиректорию и символическую ссылку.

При демонстрации работы программы удаленно необходимо:

- Продемонстрировать передачу данных из режима пользователя в режим ядра и из режима ядра в режим пользователя с помощью скриншота.
- Показать в каталоге созданный файл, директорию и поддиректорию.