МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ»

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ
Зав. кафедрой АСУ, профессор

А.М. Кориков

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Тема 2. Модульная структура ядра ОС

Учебно-методическое пособие

для студентов уровня основной образовательной программы: **магистратура**
направление подготовки: **09.04.01 - Информатика и вычислительная техника**

Разработчик
доцент кафедры АСУ

В.Г. Резник

2017

Резник В.Г.

Современные операционные системы. Тема 2. Модульная структура ядра ОС. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 58 с.

Учебно-методическое пособие предназначено для изучения темы №2 по дисциплине «Современные операционные системы» для студентов уровня основной образовательной программы магистратура направления подготовки: 09.04.01 «Информатика и вычислительная техника».

Оглавление

Введение

Тема 2 посвящена изучению составных частей ядра ОС, известных как загружаемые модули. Хотя эта технология развивается достаточно давно и ей посвящено множество публикаций, ее изучение требует не только достаточно высокой профессиональной подготовки программиста, но и знание ряда принципов и исключений, кторые обычно не обсуждаются в учебных курсах по программированию. С другой стороны, отсутствие должной теоретической и практической подготовки по этим вопросам ограничивают потенциал развития магистранта в плане понимания современных тенденций развития программного обеспечения вычислительной техники.

Данное учебное пособие содержит как теоретический, так и практический материал в объеме достаточном для получения начальных знаний по технологии создания и применения загружаемых модулей ядра ОС.

В теоретическом плане рассматриваются следующие вопросы:

- Назначение и функции модулей ядра ОС.
- Системные вызовы ядра ОС.
- Траектория системного вызова.
- Модули ядра и пользовательские процессы.
- Интерфейсы модуля для взаимодействия с ядром.
- Коды ошибок.
- Параметры, передаваемые модулю.
- Инструментарий разработчика.
- Модуль как драйвер.
- Динамические устройства.
- Управление устройствами.

Теоретический материал конкретизируется на примерах современных ядер ОС Linux, которые достаточно подробно описаны в литературе.

Теоретические знания закрепляются во время проведения лабораторной работы, которая обеспечивает как закрепление теоретических знаний, так и нужные навыки для освоения объявленной темы. По завершению лабораторной работы студент должен уметь самостоятельно разрабатывать простейшие модули и драйвера ОС.

Методика проведения лабораторных работ по данной дисциплине предполагает использование специального дистрибутива ОС УПК АСУ, которое установлено в компьютерных классах кафедры АСУ.

1 Тема 2. Модульная структура ядра ОС

О модульной структуре ядра ОС говорилось очень много. В частности, об этом упоминалось в курсе «***Операционные системы***», а также в первой теме нашего курса [5].
Обычно, о модулях ОС говорят в контексте ее базовой архитектуры, которая часто рассматривается только в двух вариантах:

- *монолитное ядро*, содержащее в себе весь системный функционал;
- *микроядро*, часть системного функционала которого вынесено в пространство пользователя и реализованное в виде системных сервисов.

В 1960-е годы, в Массачусетском Технологическом институте (***MIT***) и в ряде других компаний, была разработана экспериментальная операционная система *Multics* (*Multiplexed Information and Computing Service*) для машины GE-645.
Десять лет спустя, на ЭВМ ***PDP-7*** создается ОС UNIX, часы которой идут от даты ***01 января 1970 года***.
Двадцать лет спустя, Эндрю Танненбаум (***Andrew Tanenbaum***) создал *микроядерную версию UNIX®* под названием *MINIX* (***minimal UNIX***), которая могла работать на небольших персональных компьютерах.
В начале 1990-х, Линус Торвальдс (***Linus Torvalds***) разработал первую версию ОС *Linux*, которая работала на микропроцессорах ***80386***, но имела *монолитное ядро*.

1.1 Назначение и функции модулей ядра ОС

С целью обеспечения должного уровня конкретизации, основной учебный материал излагается далее для ОС Linux дистрибутива ***Arch Linux***.

Термин — *модульное ядро* — применим для всех архитектур ядер ОС, хотя системный контекст у них принципиально различный:

- *модуль монолитного ядра* предназначен для работы в самом ядре ОС, используя его единое адресное пространство, поэтому его структура привязана не только к архитектуре самой ОС, но и к конкретной сборке одного и того же ядра;
- *модуль микроядра* — программа, предназначенная для работы в пользовательском пространстве ОС, поэтому такой модуль выделяется своим системным назначением (***системный сервис***), но управляется как обычный пользовательский процесс.

Замечание

ОС Linux использует оба типа модулей, хотя первый тип модуля является основным.
В качестве примера второго типа модулей следует указать на систему *fuse* (***file system in user space***), которая используется для поддержки файловой системы типа *ntfs*.

Модульная структура ОС Linux обеспечивает ей гибкие возможности использования системного ПО, *особенно на этапе ее загрузки*, требующей подключения как нужных драйверов устройств, так и нужного системного ПО, например:

- *загрузка корневой ФС из сети* требует наличия не только драйверов сетевых устройств, но и системного ПО для работы с серверами архивов;
- *загрузка ОС УПК АСУ*, в рабочих классах кафедры АСУ, требует системного ПО технологии *fuse*, для обеспечения работы с ФС типа *ntfs*.

В современных ОС, основная работа с модулями осуществляется автоматически.
Для «ручного» управления модулями используются утилиты:

lsmod	Вывод списка загруженных модулей.
insmod	Загрузка нужного модуля.
rmmod	Выгрузка конкретного модуля.
modinfo	Получение информации о конкретном модуле.
modprobe	Универсальная утилита, обеспечивающая многие функции предыдущих.

Основное местоположение модулей — директория */lib/modules/<версия ядра>*.
Версию ядра можно получить командой:

```
uname -r
```

Список всех модулей в системе можно получить командой:

```
find /lib/modules/`uname -r` -name '*.ko'
```

Замечание
Модули ОС Arch Linux хранятся в сжатом виде, поэтому следует использовать команду:

```
find /lib/modules/`uname -r` -name '*.ko.gz'
```

Основные проблемы использования модулей ОС — отсутствие нужного модуля или его конфликт с другими модулями или с аппаратным обеспечением ЭВМ.
Часто, эти проблемы вызваны отсутствием нужного драйвера или специального ПО, называемого *firmware*.

Firmware — встроенное изготовителем аппаратных средств программное обеспечение, обычно поставляемое («***прошитое***») с оборудованием.

Устранение указанных проблем всегда связано с получением нужной информации о модуле.
Рассмотрим это - на примере работы с модулем ***rt73usb***.

Выполним в терминале команду:

```
modinfo rt73usb
```

На рисунке 1.1 показана начальная часть вывода этой команды.

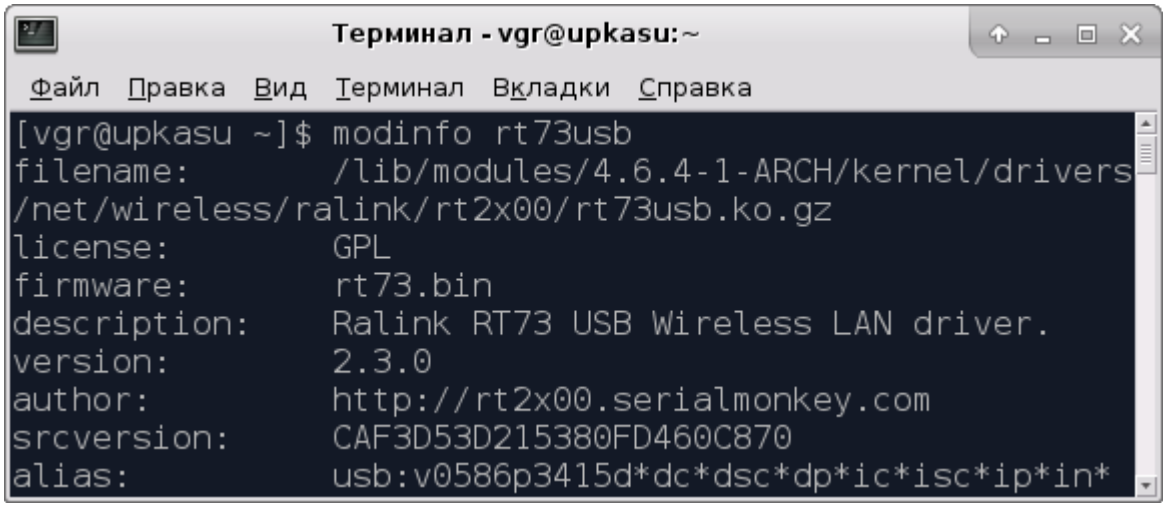

Рисунок 1.1 – Вывод команды modinfo

Содержание вывода этой команды несет важную информацию:

- *filename:* - полный (абсолютный) путь к файлу модуля; модуль находится в виде, сжатом архиватором *gzip*;
- *license:* - обозначение лицензии;
- *firmware:* – имя файла "прошивки", которую использует этот драйвер; файл с указанным именем (*rt73.bin*) обязательно должен находиться в системной директории */lib/firmware*;
- *description:* - краткое описание модуля, из которого понятно, что модуль является драйвером и относится к службе сети (проводная сеть);
- *version:* - версия продукта (необходима для жалоб и конкретизации диалога);
- *author:* - информация об авторе (в данном случае – ссылка URL, возможно – адрес компании, создавшей модуль);
- *srcversion:* - версия исходного текста модуля (наверно - помогает, при личном общении с автором).

Замечание

Важно, что нужный нам модуль - существует, но следует так же убедиться в существовании firmware – *rt73.bin*. Сделать это – проще командой:

```
ls -la /lib/firmware/rt73.bin
```

Если такой файл – отсутствует, то следует искать его на сайте производителя, например, по адресу указанного URL.

Автоматической загрузкой модулей, в современных ОС, занимается менеджер устройств ***udev*** (***systemd-udevd***), работа которого будет изучена в третьей теме данного курса.
Управление загрузкой/выгрузкой модулей осуществляется с учетом конфигурационных файлов, расположенных в директории */etc/modprobe.d*.
Назначение файлов конфигурации определяется двумя основными функциями:

- *передача значений параметров* для модулей, которые их используют при загрузке;
- *перечисление модулей "**черного списка**"*, указывающие имена запрещенных к загрузке модулей.

Как программное обеспечение, модули ядра являются объектными файлами, способными подключаться к ПО ядра (загружаться), так и отключаться от ПО ядра (выгружаться, удаляться). В литературе, для них используют разные термины:

- *LKM* – loadable kernel module – загружаемые модули ядра;
- *kld* – kernel loadable module;
- *kext* – kernel extention;
- *KLM* – Kernel Loadable Modules;
- *KMOD* – Kernel Modules.

Непосредственно для Linux, имеется документация, поставляемая с исходными текстами ядра, а также инструкция "*The Linux Kernel Module Programming Guide*", расположенная на сайте: *http://tldp.org/LDP/lkmpg/2.6/html/index.html*, примеры из которой, с минимальными изменениями, используются в данном методическом пособии.

Каждый загружаемый модуль ядра должен иметь минимум две функции:

- *init_module()* - функция старта (инициализации) модуля, выполняемая в процессе загрузки модуля;
- *cleanup_module()* - функция завершения, вызываемая в момент удаления модуля из ядра ОС.

На листинге 1.1 приведен пример простейшего модуля, использующего только эти две функции, и печатающий в системный журнал сообщения о своей работе.

Листинг 1.1 – Пример загружаемого модуля ядра hello-1.c

```
/*
 *  hello-1.c – Простейший модуль ядра.
 */
#include <linux/module.h>       /* Нужны для всех модулей */
#include <linux/kernel.h>       /* Нужен для KERN_INFO    */

int init_module(void)
{
        printk(KERN_INFO "Здравствуй, мир 1.\n");

        /*
```

```
         * Если init_module() возвратит не нулевое значание, то
         * модуль не может быть загружен.
         */
        return 0;
}


void cleanup_module(void)
{
        printk(KERN_INFO "До свидания, мир 1.\n");
}
```

Замечание

Обе функции модуля используют системную функцию ядра *printk()*, которая выводит текстовые сообщения в системный журнал.
В свою очередь, вывод текста обрабатывается макросом KERN_INFO, задающим уровень привилегии сообщений, определенных в файле *kernel.h*.

На листинге 1.2 представлен список констант квалификаторов возможных уровней сообщений функции *printk()*.

Листинг 1.2 – Константы квалификаторы уровней сообщений

```
#define KERN_EMERG      "<0>"   /* использование системы невозможно          */
#define KERN_ALERT      "<1>"   /* действие должно быть выполнено немедленно */
#define KERN_CRIT       "<2>"   /* критическая ситуация                      */
#define KERN_ERR        "<3>"   /* сообщения об ошибках                      */
#define KERN_WARNING    "<4>"   /* предупреждающие сообщения                 */
#define KERN_NOTICE     "<5>"   /* ситуация нормальная, но требует внимания  */
#define KERN_INFO       "<6>"   /* информационные сообщения                  */
#define KERN_DEBUG      "<7>"   /* отладочные сообщения                      */
```

Использование одинаковых функций, во всех загружаемых модулях ядра, не является хорошей практикой программирования, поэтому современные технологии разработки рекомендуют использование макросов *module_init()* и *module_exit()*, как это показано на примере листинга 1.3.

Листинг 1.3 – Пример загружаемого модуля ядра hello-2.c

```
/*
 *  hello-2.c – Демонстрация использования макросов module_init() и module_exit().
 *  Это предпочитается использованию init_module() и cleanup_module().
 */
#include <linux/module.h>       /* Нужны для всех модулей */
#include <linux/kernel.h>       /* Нужен для KERN_INFO    */
#include <linux/init.h>         /* Нужен для макросов     */

static int __init hello_2_init(void)
{
        printk(KERN_INFO "Здравствуй, мир 2\n");
        return 0;
}
```

```
static void __exit hello_2_exit(void)
{
        printk(KERN_INFO "До свидания, мир 2\n");
}

module_init(hello_2_init);
module_exit(hello_2_exit);
```

Реализация модулей допускает задание констант и элементов документации, как это показано на примере листинга 1.4.

Листинг 1.4 – Пример загружаемого модуля ядра hello-3-4.c

```
/*
 *  hello-3-4.c – Демонстрация задания параметра и документации.
 */
#include <linux/module.h>       /* Нужны для всех модулей */
#include <linux/kernel.h>       /* Нужен для KERN_INFO    */
#include <linux/init.h>         /* Нужен для макросов     */

#define DRIVER_AUTHOR "Reznik V.G. <vgr@asu.tusur.ru>"
#define DRIVER_DESC   "Пример драйвера"

static int hello3_data __initdata = 3;

static int __init init_hello_4(void)
{
        printk(KERN_INFO "Здравствуй, мир %d-4\n", hello3_data);
        return 0;
}

static void __exit cleanup_hello_4(void)
{
        printk(KERN_INFO "До свидания, мир 3-4\n");
}

module_init(init_hello_4);
module_exit(cleanup_hello_4);

/*
 *  Документировать можно так:
 */

MODULE_LICENSE("GPL");           /* Указание лицензии   */
MODULE_AUTHOR(DRIVER_AUTHOR);    /* Кто написал модуль? */
MODULE_DESCRIPTION(DRIVER_DESC); /* Что модуль делает?  */

/*
 *  Этот модуль использует /dev/testdevice, что декларируется
 *  макросом MODULE_SUPPORTED_DEVICE.
 */
MODULE_SUPPORTED_DEVICE("testdevice");
```

1.2 Системные вызовы ядра ОС

Общую функциональную архитектуру ОС GNU/Linux можно представить иерархической структурой, показанной рисунком 1.2, который был заимствован с сайта: *http://www.ibm.com/developerworks/ru/library/l-linux-kernel/index.html*.

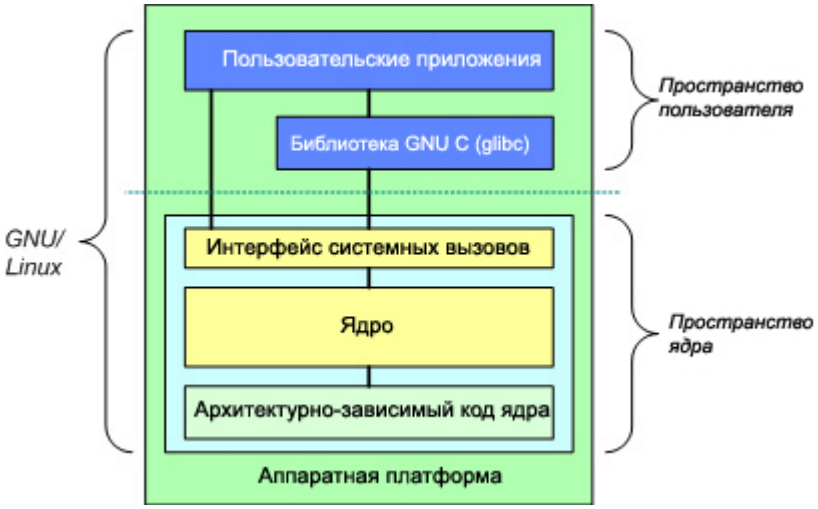

Рисунок 1.2 — Функциональная архитектура ОС GNU/Linux

На верхнем уровне иерархии находится пользовательское пространство или *пространство приложений*. Здесь исполняются приложения пользователя.
Под пользовательским пространством располагается *пространство ядра*. Здесь функционирует ядро Linux.
Пользовательские приложения обращаются к ядру:

- *напрямую*, через интерфейс системных вызовов;
- *через библиотеку* GNU C (***glibc***), которая предоставляет интерфейс системных вызовов, обеспечивающий связь с ядром и дающий механизм для перехода от приложения, работающего в пространстве пользователя, к ядру.

Интерфейс системных вызовов (***SCI — System Call Interface***) - это тонкий механизм ПО, предоставляющий средства для вызова функций ядра из пространства пользователя.
SCI - фактически представляет собой *службу мультиплексирования и демультиплексирования* вызова функций.
Для аппаратной архитектуры Intel x86, механизм ***SCI*** традиционно релизуется на основе программного прерывания с различными векторами:

MS-DOS	-	21h
Windows	-	2Eh
Linux	-	80h
QNX	-	21h
MINIX 3	-	21h

Замечание

Начиная с 2008 года, многие операционные системы (***Windows, Linux***) отказались от использования программного прерывания *int*, и перешли к реализации системного вызова и возврата из него через новые команды процессора *sysenter* (*sysexit*).
При таком подходе, ничего принципиально нового не появилось, так как ключевые параметры перехода (CS, SP, IP) просто стали загружаться не из памяти, а из специальных внутренних регистров *MSR* (***Model Specific Registers***) с предопределёнными (***0x174, 0x175, 0x176***) номерами, куда они записываются специальной новой командой процессора *wmsr*.

Независимо от реализации, непосредственное исполнение SCI, в ОС Linux, выполняет функция *syscall()*:

```
# syscall – непрямой системный вызов

#define _GNU_SOURCE         /* Смотри feature_test_macros(7) */
#include <unistd.h>
#include <sys/syscall.h>   /* Для определений SYS_xxx       */

long syscall(long number, ...);
```

где *number* — целочисленное значение, соответствующее строке системной таблицы *sys_call_table*, ряд значений которых показан на рисунке 1.3.
<другие параметры> - аргументы вызываемой фнкции.
sys_call_table — таблица ссылок на функции ядра, реализующие функционал системного вызова.

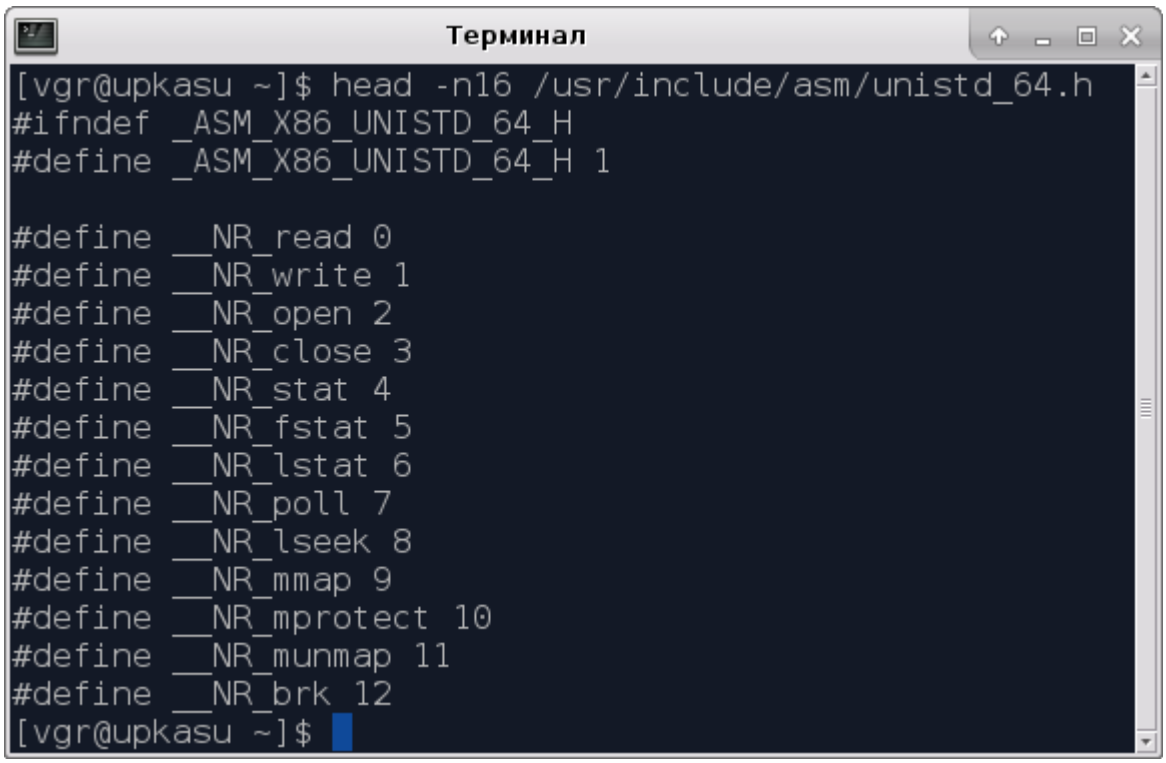

Рисунок 1.3 — Целочисленные номера системных вызовов

Замечание

Таблица *sys_call_table*, а следовательно и номера системных вызовов, создаются в процессе компиляции ядра ОС, при условии, что имена статических или загружаемых модулей указаны во время конфигурации ядра ОС.

Поскольку, учебная ОС УПК АСУ использует уже готовое ядро ОС Arch Linux, то отсутствует нормальная возможность написать модуль, использующий системные вызовы ОС.
Поэтому, *для учебных целей*, используется модифицированный текст загружаемого модуля (*syscall.c*), написанный ***в 2001 году*** (*Peter Jay Salzman*) и размещенный на сайте: *http://tldp.org/LDP/lkmpg/2.6/html/x978.html*.

Идея реализации такого модуля, приведенная на листинге 1.5, следующая:

- используется внешняя ссылка модуля на таблицу *sys_call_table[]* и хорошо известный системный вызов, в данном случае *open()*, для которого известны все параметры вызова;
- функция инициализации *init_module()* сохраняет в модуле ссылку на оригинальный обработчик системного вызова *open()*, а на место оригинального обработчика указывает свою функцию, *our_sys_open()*;
- когда пользовательский процесс выполняет системную функцию *open()*, то запускается функция *our_sys_open()*, которая выводит в системный журнал сообщение, а затем вызывает оригинальный обработчик системного запроса;
- когда модуль выгружается из ядра ОС, функция *cleanup_module()* восстанавливает значение таблицы *sys_call_table[]*.

Листинг 1.5 – Учебный вариант загружаемого модуля ядра upk_syscall.c

```
/*
 *  syscall.c
 *
 *  System call "stealing" sample.
 *  Copyright (C) 2001 by Peter Jay Salzman
 *
 * Модифицирована Reznik, 15.08.2016, upk_syscall.c
 *
 * Необходимые файлы заголовков
 *
 * Стандартные в модулях ядра
 */
#include <linux/kernel.h>       /* We're doing kernel work */
#include <linux/module.h>       /* Specifically, a module, */
#include <linux/moduleparam.h>  /* which will have params */
#include <linux/unistd.h>       /* The list of system calls */

/*
 * For the current (process) structure, we need
 * this to know who the current user is.
 */
#include <linux/sched.h>
#include <asm/uaccess.h>
```

```
/*
 * The system call table (a table of functions). We
 * just define this as external, and the kernel will
 * fill it up for us when we are insmod'ed
 *
 * sys_call_table is no longer exported in 2.6.x kernels.
 * If you really want to try this DANGEROUS module you will
 * have to apply the supplied patch against your current kernel
 * and recompile it.
 */
extern void *sys_call_table[];

/*
 * UID we want to spy on - will be filled from the
 * command line
 */
static int uid;
module_param(uid, int, 0644);

/*
 * A pointer to the original system call. The reason
 * we keep this, rather than call the original function
 * (sys_open), is because somebody else might have
 * replaced the system call before us. Note that this
 * is not 100% safe, because if another module
 * replaced sys_open before us, then when we're inserted
 * we'll call the function in that module - and it
 * might be removed before we are.
 *
 * Another reason for this is that we can't get sys_open.
 * It's a static variable, so it is not exported.
 */
asmlinkage int (*original_call) (const char *, int, int);

/*
 * The function we'll replace sys_open (the function
 * called when you call the open system call) with. To
 * find the exact prototype, with the number and type
 * of arguments, we find the original function first
 * (it's at fs/open.c).
 *
 * In theory, this means that we're tied to the
 * current version of the kernel. In practice, the
 * system calls almost never change (it would wreck havoc
 * and require programs to be recompiled, since the system
 * calls are the interface between the kernel and the
 * processes).
 */
asmlinkage int our_sys_open(const char *filename, int flags, int mode)
{
        int i = 0;
        char ch;

        /*
         * Check if this is the user we're spying on
         */
        if (uid == current->uid) {
                /*
                 * Report the file, if relevant
                 */
```

```
                printk("Opened file by %d: ", uid);
                do {
                        get_user(ch, filename + i);
                        i++;
                        printk("%c", ch);
                } while (ch != 0);
                printk("\n");
        }

        /*
         * Call the original sys_open - otherwise, we lose
         * the ability to open files
         */
        return original_call(filename, flags, mode);
}

/*
 * Initialize the module - replace the system call
 */
int init_module()
{
        /*
         * Warning - too late for it now, but maybe for
         * next time...
         */
        printk(KERN_ALERT "I'm dangerous. I hope you did a ");
        printk(KERN_ALERT "sync before you insmod'ed me.\n");
        printk(KERN_ALERT "My counterpart, cleanup_module(), is even");
        printk(KERN_ALERT "more dangerous. If\n");
        printk(KERN_ALERT "you value your file system, it will ");
        printk(KERN_ALERT "be \"sync; rmmod\" \n");
        printk(KERN_ALERT "when you remove this module.\n");

        /*
         * Keep a pointer to the original function in
         * original_call, and then replace the system call
         * in the system call table with our_sys_open
         */
        original_call = sys_call_table[__NR_open];
        sys_call_table[__NR_open] = our_sys_open;

        /*
         * To get the address of the function for system
         * call foo, go to sys_call_table[__NR_foo].
         */

        printk(KERN_INFO "Spying on UID:%d\n", uid);

        return 0;
}

/*
 * Cleanup - unregister the appropriate file from /proc
 */
void cleanup_module()
{
        /*
         * Return the system call back to normal
         */
        if (sys_call_table[__NR_open] != our_sys_open) {
```

```
                printk(KERN_ALERT "Somebody else also played with the ");
                printk(KERN_ALERT "open system call\n");
                printk(KERN_ALERT "The system may be left in ");
                printk(KERN_ALERT "an unstable state.\n");
        }

        sys_call_table[__NR_open] = original_call;
}
```

1.3 Траектория системного вызова

Как показано ранее на рисунке 1.2, пользовательское приложение может напрямую взаимодействовать с ядром ОС, через интерфейс системных вызовов (SCI). Такая ситуация возможна, но для очень ограниченного круга приложений, например, для приложений (набора системных утилит), реализуемых в проекте *busybox*.

В общем случае, прикладной процесс вызывает требуемые ему службы выполняя:

- *динамическое связывание* посредством библиотечного вызова к множеству библиотек вида ****.so***;
- *статическое связывание*, прикомпоновывая к себе фрагмент из библиотеки вида ****.a***.

Самые известные примеры:

- стандартная библиотека языка С - ***libc.so***;
- библиотека POSIX потоков - ***libpthread.so***.

Значительная часть вызовов обслуживается непосредственно внутри библиотеки, не требуя никакого вмешательства со стороны ядра.
Другая часть - требует дальнейшего обслуживания со стороны ядра.
Все подобные вызовы называются *API* (***Application Program Interface***) и классифицируются как *библиотечные вызовы*.

В конечном итоге, та часть библиотечных функций, которая требует обращения к ядру ОС, закончится системным вызовом *syscall()*, где ее первым аргументом будет номер строки в таблице *sys_call_table[]*.

В качестве примера, рассмотрим целаую группу функций вида *exec*()*, служащие для разных варианов запуска дочерних процессов:
`execl(), execlp(), execle(), execv(), execvp()`.
Все эти библиотечные функции лишь ретранслируют вызов к единственному системному вызову *execve()*, который, как показано на рисунке 1.4, реализуется через *syscall(59, ...)*.

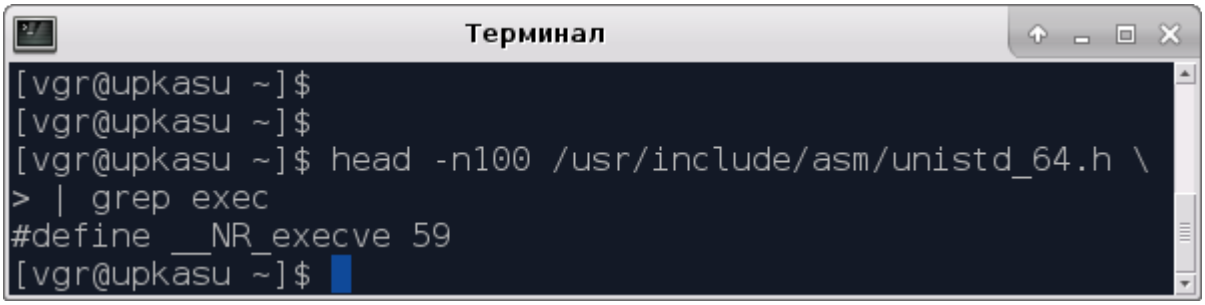

Рисунок 1.4 — Номер сервисной функции ядра для execve()

Аналогично, библиотечный вызов `printf()` выполняется как **системный** вызов `write( 1, string, strlen( string ))`, который трансформируется в вызов `syscall( __NR_write, …)`.

1.4 Модули ядра и пользовательские процессы

Любой разработчик приложений решает не простой вопрос о том, *куда разместить часть функционала ПО*: в библиотеку или в загружаемый модуль ядра ОС.
Очевидно, что вопрос о реализации модуля ядра может стоять только в том случае, если его функционал может быть реализован на языке ***C***, в противном случае — сам вопрос снимается с повестки дня.

Преимущество модулей ядра — более тесная связь с аппаратным обеспечением ЭВМ, что позволяет рассчитывать на *повышение быстродействия* работы приложения в целом.

Преимущество библиотек функций — более универсальная реализация приложения, *повышающая его переносимость* на другие платформы и реализации ОС.

Основой системного программирования является *язык* ***C***, который был создан как язык для написания операционных систем, в качестве максимальной замены платформозависимых *языков ассемблера*.
Его использование позволило освободиться от *технологий адресного программирования*, возложив эту задачу на инструментальные средства поддержки языка.
Тем не менее, сама подготовка программиста и современные интегрированные средства разработки на языке ***C*** ориентированы на *использование технологии библиотек функций*, обеспечивающих разработчику небходимый уровень абстракции мышления. Все это создает дополнительные моральные и технологические проблемы при реализации модулей ядра ОС.

Далее, выделим *четыре основные особенности* в технологии написания модулей ядра, которые необходимо обязательно учитывать как при планировании, так и в процессе реализации системного ПО ОС.

1.4.1 Использование стандартных библиотек функций

Ядро ОС не имеет доступа к стандартным библиотекам языка ***C***, поэтому:

- *отсутствует* стандартизация POSIX API;
- *присутствует* собственная API, синтаксически и семантически создаваемая разработчиками ядра ОС;
- *имеются* слабо документированные части ПО;
- *содержится* большое количество экспортируемых имен ядра, которые необходимо учитывать при написании собственного ПО.

Общее количество имен ядра может достигать ***100000*** наименований, для некоторых реализаций. Все эти имена отражаются в псевдофайле */proc/kallsyms*. На рисунке 1.5 приведен способ подсчета этих имен для текущего ядра ОС, показывающий, что их количество рано ***44384***.

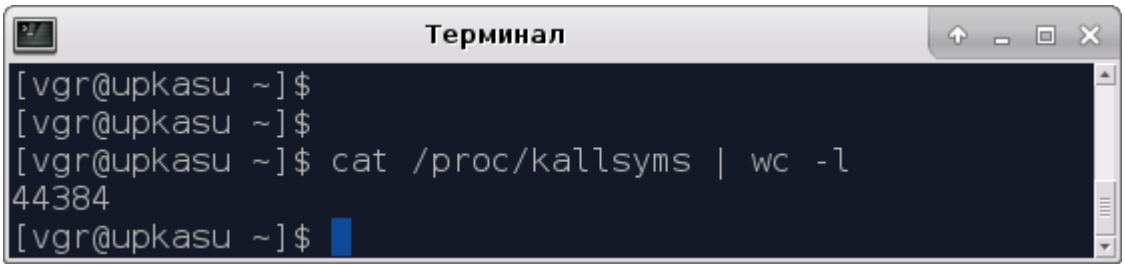

Рисунок 1.5 — Количество имен текущего ядра ОС

В таблице 1.1 приведены имена функций, семантика которых во многом совпадает как в пространстве ядра, так и в пространстве пользователя.

Таблица 1.1 Вызовы в пространствах пользователя и ядра

API процессов (POSIX)	**API ядра**
strcpy(), strncpy(), strcat(), strncat(), strcmp (), strncmp(), strchr (), strlen(), strnlen(), strstr(), strrchr()	strcpy (), strncpy (), strcat(), strncat(), strcmp(), strncmp(), strchr(), strlen(), strnlen(), strstr(), strrchr()
printf()	printk()
execl(), execlp(), execle(), execv(), execvp(), execve()	call_usermodehelper()
malloc(), calloc(), alloca()	kmalloc(), vmalloc()

API процессов (POSIX)	API ядра
kill(), sigqueue()	send_sig()
open(), lseek(), read(), write(), close()	filp_open(), kernel_write(), vfs_llseek(), vfs_read(), vfs_write(), filp_close(), kernel_read()
pthread_create()	kernel_thread()
pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_unlock()	rt_mutex_lock(), rt_mutex_trylock(), rt_mutex_unlock()

1.4.2 Защита памяти ЭВМ

В ядре *отсутсвуют*:

- ***защита памяти***, обеспечивающая контроль границ каждого процесса, запущенного в пространстве пользователя;
- **замещение страниц** - каждый байт ядра – это один байт физической памяти.

1.4.3 Использование вычислений с плавающей запятой

В ядре нельзя использовать вычисления с плавающей точкой (***FPU***).
Если это потребуется, то вычисления с плавающей точкой *следует эмулировать* через целочисленные вычисления.

1.4.4 Локальные переменные и стек

В ядре – фиксированный стек.
Обычно стек равен *двум страницам памяти*, что для ***x86***, соответствует:

- 8 КБайт - для 32 разрядных систем;
- 16 Кбайт - для 64 разрядных.

Стек - область адресного пространства программы, в котором выделяются и сохраняются:

- регистры процессора, в момент его прерывания;
- регистры процессора и аргументы функций, в момент вызова этих функций;
- локальные переменные программы.

Локальные переменные – это все переменные, объявленные внутри любого программного блока (функции), не имеющие ключевого слова ***static***.

Стек в режиме ядра - ограничен по размеру и **не может быть** изменён!
Поэтому, в коде ядра желательно *не использовать конструкции*, сокращающие пространство стека:

- рекурсивные вызовы;
- передавать структуру в функцию, в качестве параметра «***по значению***», или возвращать структуру из функции;
- объявлять крупные локальные структуры внутри функций и другие.

1.5 Интерфейсы модуля

С прикладной точки зрения, модуль должен взаимодействовать с другими программными объектами ОС.
В данном подразделе рассматриваются три таких группы объектов:

- взаимодействие с программными единицами ядра ОС;
- коды ошибок, возвращаемые функциями API ядра;
- взаимодействие модуля с уровнем пользователя.

1.5.1 Взаимодействие с ядром ОС

Модуль, находясь в ядре ОС:

- *сам обладает* некоторой функциональностью;
- *пользуется функциональностью* других модулей и функций ядра ОС;
- *совместно с другими* модулями и функциями, обеспечивает функциональность некоторой задачи ядра.

Ядро ОС и подгруженные модули экспортируют набор имен, которые используются в качестве *API ядра*.
Все имена ядра содержатся в текстовом псевдофайле */proc/kallsyms*.
Для примера, выведем часть имен, включающих в себя слово ***printk***, как показано на рисунке 1.6.

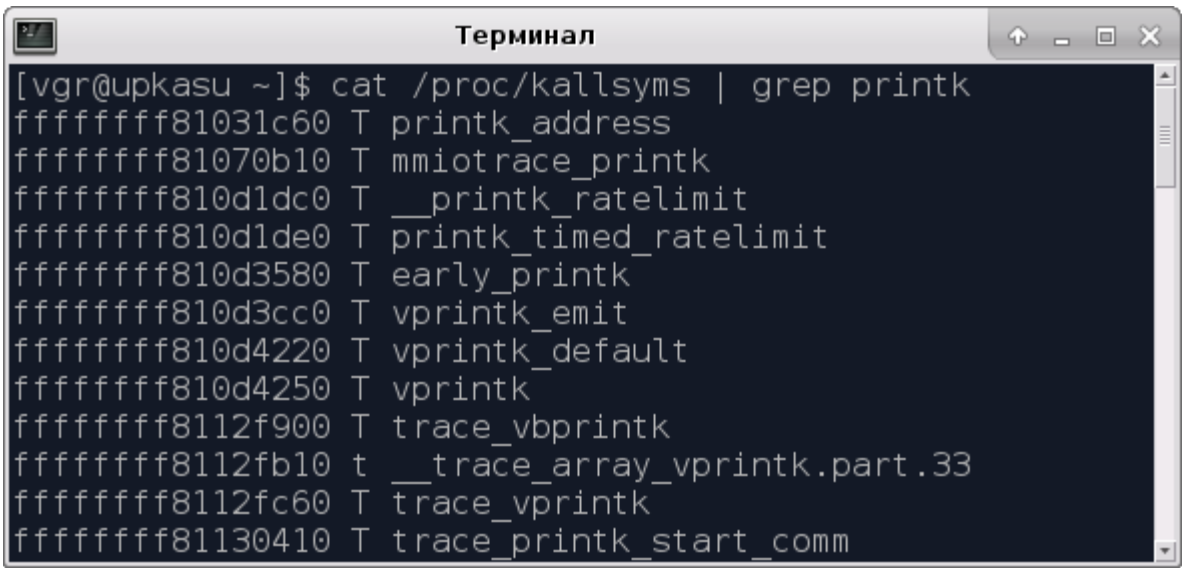

Рисунок 1.6 — Вывод имен из псевдофайла /proc/kallsyms

Первый столбец каждой строки указывает *абсолюный адрес* экспортируемого ядром ***имени***, указанного в **третьем столбце**.

Символьная литера, указанная во втором столбце, определяет функциональное назначение имени:

- *Большая буква* — символ определяет глобальное имя (***external***);
- *Маленькая буква* — символ определяет локальное имя;
- *"D"* - символ в инициализированной секции данных (***data***);
- *"R"* - символ в секции данных (***data***) только для чтения;
- *"T"* - символ в секции исполнимого кода (***text***);
- *"U"* - символ в секции неопределен.

Полный перечень значений символьных литер можно почерпнуть из описания утилиты *nm*.
Утилита nm предназначена для анализа символов *файлов объектного формата*.

Список имен ядра в псевдофайле */proc/kallsyms* формируется динамически, при каждой загрузке операционной системы.
Изменения, вносимые в состав системы (установка нового оборудования, драйверов, целевых модулей) будут изменять и содержимое этой таблицы.
Загрузка и выгрузка модулей будут также динамически отслеживаться в этой таблице.

Замечание

В отличие от очень жёстких ограничений для пользовательских API, налагаемых стандартом POSIX, разработчики ядра Linux **не связаны** *требованиями совместимости снизу вверх*, поэтому API ядра может изменяться даже между **подверсиями** ядра.
Функции ядра довольно плохо документированы, по крайней мере, в сравнении с документацией POSIX-вызовов для пользовательского пространства.
Источниками для изучения особенностей реализации функций API ядра могут служить прототипы и определения из заголовочных файлов в дереве каталогов:
/lib/modules/`uname -r`/build/include.

1.5.2 Коды ошибок

Общее правило, которое не всегда соблюдается, требует чтобы функции API ядра, в случае ошибки выполнения, возвращали ***отрицательный*** *результат завершения* (код ошибки).

В качестве отрицательного результата, при возникновении ошибки API ядра, обычно используются ***те же числовые коды*** ошибок, что и POSIX, *но со знаком минус*.
Этого же правила рекомендуется придерживаться и при написании собственных функций в теле модуля, в частности, ***функции инициализации*** модуля.

Положительные возвращаемые результаты в API ядра используются для возврата численных результатов.

Возвращаемое нулевое значение — как логический признак в некоторых API.

Коды ошибок, возвращаемые функциями API ядра, определены в двух файлах: <*asm-generic/errno.h*> и <*asm-generic/errno-base.h*>.
Пример части таких ошибок представлен на листинге 1.6.

Листинг 1.6 — Пример части ошибок, возвращаемых ядром ОС

```
#define EPERM            1      /* Операция не поддерживается */
#define ENOENT           2      /* Нет такого файла или каталога */
#define ESRCH            3      /* Нет такого процесса */
#define EINTR            4      /* Прерванный системный вызов */
#define EIO              5      /* Ошибка ввода/вывода */
#define ENXIO            6      /* Нет такого устройства или адреса */
#define E2BIG            7      /* Список аргументов слишком длинный */
#define ENOEXEC          8      /* Ошибка формата вызова exec */
#define EBADF            9      /* Ошибочный дескриптор файла */
#define ECHILD          10      /* Нет дочернего процесса */
#define EAGAIN          11      /* Повторите попытку снова */
#define ENOMEM          12      /* Недостаточно памяти */
#define EACCES          13      /* Доступ запрещён */
#define EFAULT          14      /* Неверный адрес */
#define ENOTBLK         15      /* Требуется блочное устройство */
#define EBUSY           16      /* Устройство или ресурс заняты */
```

Замечание

Полный перечень ошибок следует искать в указанных выше файлах.
Для перехода в директорию, содержащую эти файлы, следует выполнить в терминале команду:

```
cd "/lib/modules/`uname -r`/build/include/uapi/asm-generic"
```

1.5.3 Взаимодействие с уровнем пользователя

Модули, а также другие функциональные подсистемы ядра ОС должны иметь интерфейсы взаимодействия с уровнем пользователя, в котором работают системные и прикладные процессы. Рассмотрим основные из них.

Вывод диагностической информации из модуля в системный журнал.
с помощью вызова *printk()*:

- осуществляет вывод в ***текстовую консоль*** (не в графический терминал!);
- осуществляет вывод в файл системного журнала */var/log/messages*;

Содержимое системного файла журнала можно смотреть и изучать при помощи команды (утилиты) *dmesg*;

Эти диагностические сообщения, до настоящего времени, остаются основным и самым массовым инструментом для ***отладки*** кода модулей ядра.
По этому каналу, модуль также сообщает о серьезных или критических ошибках в своей работе.

Копирование данных между адресными пространствами пользователя и ядра.
Такие операции требуют привилегированного режима процессора, могут выполняться только кодом ядра.

Замечание

Операции выполняются исключительно по инициативе модуля и никогда - по инициативе пользователя.

Базовые оперции выполняются макросами:

```
int put_user( void *x, void __user       *ptr );
int get_user( void *x, const void __user *ptr );
```

Сами макросы пытаются определить размерность передаваемых данных, а затем вызывают функцию, соответствующую передаваемому типу.
Например, на рисунке 1.7 показаны функции ядра ОС, обслуживающие макрос *put_user()*.
Аналогичные функции имеются и для макроса *get_user()*.

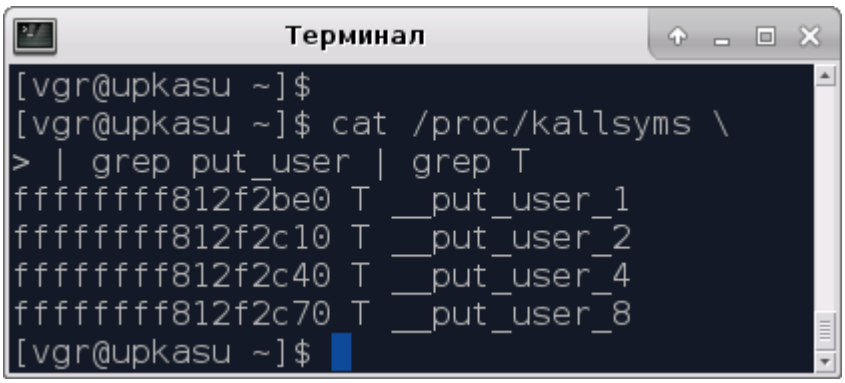

Рисунок 1.7 — Функции, обслуживающие макрос put_user()

Для обмена данными произвольного размера, существуют два других макроса, которые на самом деле используют в цикле два предыдущих:

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

Определения всех этих API можно найти в файле <***asm/uaccess.h***>.

Интерфейс взаимодействия с каталогом устройств.
Это - большая группа структур данных и функций, которые обеспечивают модулю возможности по созданию и поддержанию ***символьных имён устройств*** вида */dev/XXX*.
С помощью этого инструментария модуль может осуществлять стандартные операции ввода-вывода на устройстве (как символьном, там и блочном), и тем самым становится ***драйвером устройства***.

Это - *основной интерфейс* от модуля к пользовательскому уровню.
Он является одним из самых старых механизмов в системах UNIX/Linux.

Взаимодействие через файлы системы /proc.
Модуль может создавать, специфические для него, *индикативные псевдофайлы* в файловой системе ***/proc*** типа *proc*, а затем выводить туда отладочную или диагностическую информацию. Возможно также считывание из этих файлов управляющей информации.

Взаимодействие через файлы системы /sys.
Эта файловая система типа *sysfs* по назначению подобна **/proc**, но возникла заметно позже и имеет более строгий формат.
Считается, что ее функциональность значительно богаче и, по многим аспектам, она рассматривается в Linux в качестве замены **/proc**.

Замечание

В отличие от файловых систем **/dev** и **/proc**, файловая система **/sys** не является общим понятием для всех UNIX систем, никак не связана со стандартами POSIX и не поддерживается этими стандартами. Это собственное "изобретение" разработчиков **ядра** Linux.

Взаимодействие модуля со стеком сетевых протоколов.
Взаимодействие возможно для любых сетевых протоколов, хотя в литературе, главным образом, рассматривается только работа со стеком протоколов TCP/IP.
Ограничение относительно TCP/IP не принципиально, но так как стек протокола IP развит в Linux сильнее, нежели стеки других протоколов, то и используется он чаще.

Таким образом, модули и ядро ОС могут обеспечивать различные типы взаимодействий, которые можно изучать как совместно, так и независимо друг от друга.
Отдельные типы этих взаимодействий будут рассмотрены более подробно в следующих подразделах данного пособия.

1.6 Параметры, передаваемые модулю

При загрузке модуля ему могут быть переданы значения параметров.
Здесь наблюдается полная аналогия по передаче параметров пользовательскому процессу из командной строки, через параметры: *argc* и *массив argv[]*.

Значения параметров передаются во время загрузки модуля посредством утилит *insmod* или *modprobe*.
Утилита *modprobe* может считывать значения параметров из своих файлов конфигурации, находящихся в директории ***/etc/modprobe.d***.

Обработка входных параметров модуля обеспечивается использованием макросов, определенных в файле <***linux/moduleparam.h***>.
Для примера, рассмотрим основные часто употребляемые макросы:

```
        module_param_named( name, value, type, perm )

        #define module_param(name, type, perm)  \
              module_param_named(name, name, type, perm)

        module_param_string( name, string, len, perm )

        module_param_array_named( name, array, type, nump, perm )

        #define module_param_array( name, type, nump, perm ) \
              module_param_array_named( name, name, type, nump, perm )
```

где *name* — имя параметра, передаваемого в модуль;
value — значение параметра;
string — имя переменной модуля, которое соответствует имени параметра *name*;
array — имя переменной модуля, которое соответствует имени параметра *name*;
type — тип параметра;
len и *nump* — размерность строки и массива, соответственно;
perm — права доступа к параметру.

Замечание

Параметр *perm* указывает права доступа, например, **S_IRUGO | S_IWUSR**), которые будут относится к имени параметра, отображаемому в подсистеме ***/sys***.
Если имя параметра, отображаемое в ***/sys***, нас не интересует, то правильным значением для параметра *perm* будет **0**.

Для часто используемого макроса *module_param()*, могут использоваться следующие типы ***type***:

- *bool, invbool* - булева величина (***true*** или ***false***) - связанная переменная должна быть типа *int* (тип *invbool* инвертирует значение, так что значение ***true*** приходит как ***false*** и наоборот).

- *charp* - значение указателя на *char* - выделяется память для строки, заданной пользователем, при этом, не нужно предварительно выделять место для строки, и указатель устанавливается соответствующим образом.
- *int, long, short, uint, ulong, ushort* - базовые целые величины разной размерности; версии, начинающиеся с **u**, являются беззнаковыми величинами.

Для лучшего понимания технологии применения макросов чтения параметров модуля, следует изучить исходный текст модуля, который был заимствован с сайта: http://www.ibm.com/developerworks/ru/library/l-linux_kernel_11/ и с минимальными изменениями представлен на листинге 1.7.

Листинг 1.7 — Пример модуля читающего параметры загрузки

```
/*
 * mod-params.c – Демонстрация использования параметров модуля.
 *
 * Модифицировано: Reznik, 20.08.2016
 */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/string.h>

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

static int iparam = 0;
module_param( iparam, int, 0 );

static int k = 0;             // имена параметра и переменной различаются
module_param_named( nparam, k, int, 0 );

static char* sparam;
module_param( sparam, charp, 0 );

#define FIXLEN 5
static char s[ FIXLEN ] = ""; // имена параметра и переменной различаются
module_param_string( cparam, s, sizeof( s ), 0 );

static int aparam[] = { 0, 0, 0, 0, 0 };
static int arnum = sizeof( aparam ) / sizeof( aparam[ 0 ] );
module_param_array( aparam, int, &arnum, S_IRUGO | S_IWUSR );

static int __init mod_init( void ) {
   int j;
   char msg[ 40 ] = "";
   printk( "=======================================\n" );
   printk( "iparam = %d\n", iparam );
   printk( "nparam = %d\n", k );
   printk( "sparam = %s\n", sparam );
   printk( "cparam = %s {%d}\n", s, strlen( s ) );
   sprintf( msg, "aparam [ %d ] = ", arnum );
   for( j = 0; j < arnum; j++ )
      sprintf( msg + strlen( msg ), " %d ", aparam[ j ] );
   printk( "%s\n=======================================\n", msg );
   return -10000;
}

module_init( mod_init );
```

1.7 Инструментарий разработчика

Технология создания загружаемых модулей ядра во многом отличается от современных технологий разработки приложений на языках ***C/C++***, использующих инструментарий IDE (Интегрированных сред разработки).
В общем случае, модули ядра ОС создаются во время компиляции ядра ОС для чего:

- *с сайта дистрибьютора ОС* скачивается архив исходных текстов ядра нужной версии, который распаковывается в директории /usr/src; при этом, создается нужное дерево директорий с нужными ссылками;
- *запускается один из конфигураторов* ядра ОС, с помощью которого определяются нужные загружаемые модули и формируются файлы конфигурации и создаются нужные сценарии для компилляции ядра;
- *запускается утилита make* с набором заранее определенных параметров, в результате чего компилируется ядро и основные библиотеки, компилируются указанные модули и происходит установка всей системы.

Студенты, желающие освоить эту технологию, *должны сделать это самостоятельно*, на собственных инсталляциях ОС, поскольку *дистрибутив ОС УПК АСУ не предназначен* для полной компиляции ядра ОС!

В дистрибутивах ОС Linux, с установленным готовым ядром и модулями, возможно создание загружаемых модулей именно для этого ядра. Для этого, необходимо дополнительно установить пакет разработчика *base-devel*, содержащий нужный инструментарий программирования на языке ***C*** (***gcc***).
Этим условиям, удовлетворяет и *дистрибутив ОС УПК АСУ*, содержащий весь необходимый инструментарий для проведения лабораторных работ.

Основное отличие технологий прикладного и системного программирования связаны с разными директориями расположения заголовочных файлов языка ***C***:

- для разработки приложений - ***/usr/include***;
- для разработки модулей - ***/lib/modules/$(uname -r)/build/include***.

Дополнительно, для разработки модулей ядра:

- *разработка* ведется от имени пользователя *root*;
- *в процессе разработки* используется технология утилиты *make*;
- *необходимо правильное создание* конфигурационного файла *Makefile*, используемого утилитой *make*.

Замечание

Подробное изучение утилиты make и ее конфигурационных файлов выходит за рамки нашего курса.
Теоретический и демострационный материал данной темы опирается на документ сайта: http://tldp.org/LDP/lkmpg/2.6/html/index.html: «***The Linux Kernel Module Programming Guide***».

Непосредственное применение инструментария разработчика продемонстрируем на примере создания загружаемого модуля *hello-1.ko*, исходный текст которого представлен листингом 1.1 (на стр. 8-9) данного методического пособия.

Запускаем терминал от имени пользователя *root* и, в домашней директории пользователя *upk*, выполняем следующие действия:

- создаем директорию *~/lab2_sos/1*;
- переходим в эту директорию и, командой `mousepad &`, запускаем редактор текста;
- копируем в редактор текст листинга 1.1, как показано на рисунке 1.8;
- запоминаем исходный текст модуля как файл с именем *~/lab2_sos/1/hello-1.c*.

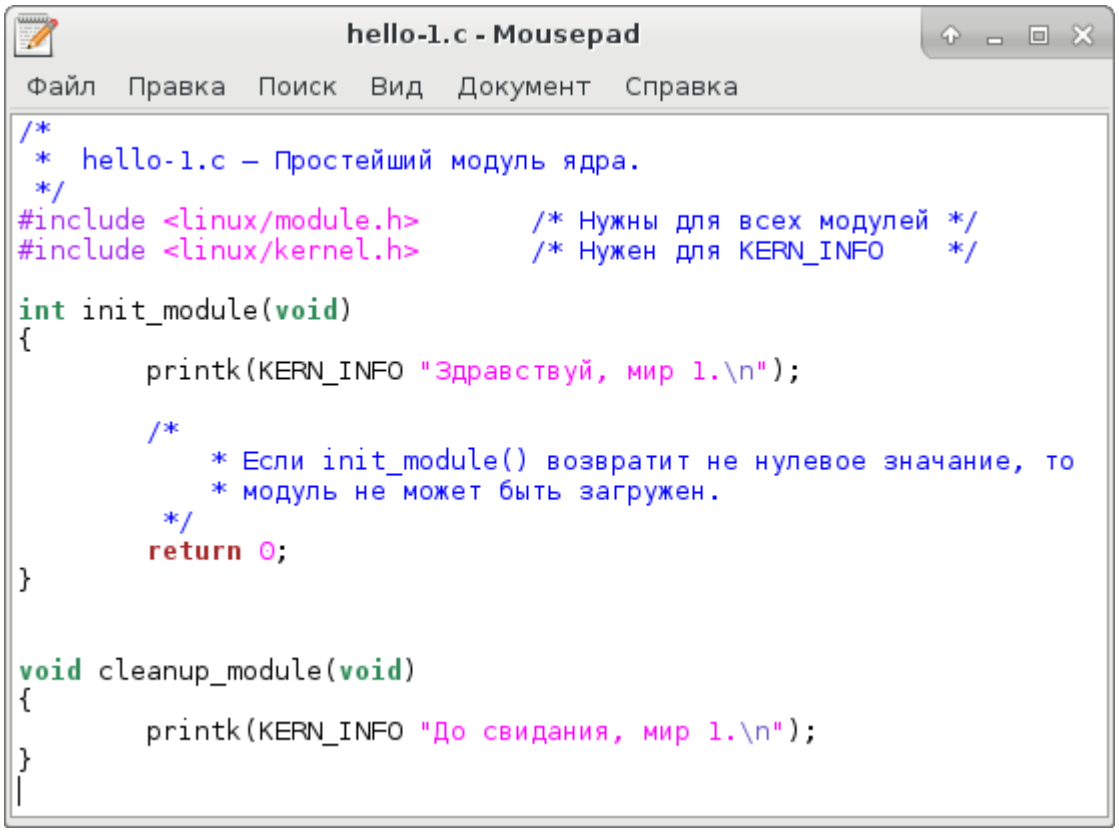

Рисунок 1.8 — Исходный текст модуля hello-1.ko

Теперь, в этой же директории, создаем файл конфигурации *Makefile*, как показано на рисунке 1.9. Обратите внимание, что *Makefile* содержит две цели:

- *all:* - предполагается по умолчанию и создает все файлы проекта;
- *clean:* - удаляет из директории все результаты разработки, кроме файлов *.c и *Makefile*; она достигается выполненим команды:

```
make clean
```

Таким образом, для создания модуля нужны два файла и выполние команды:

```
make
```

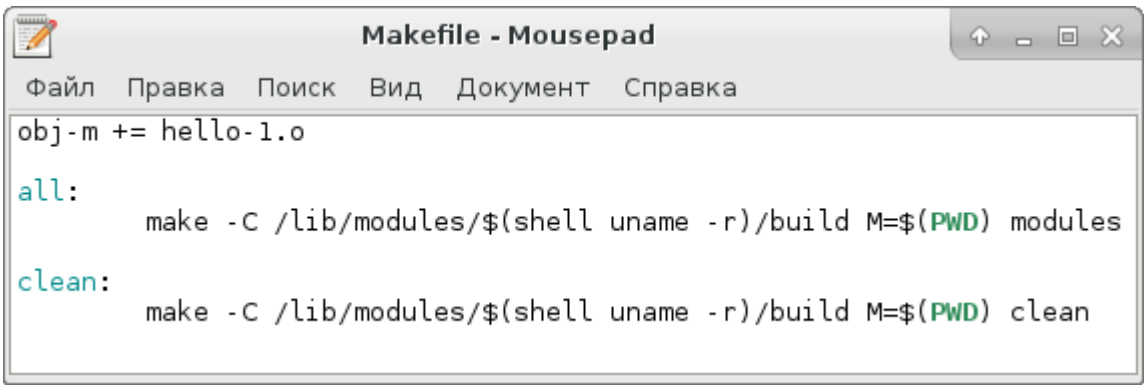

Рисунок 1.9 — Файл конфигурации для утилиты make

В результате указанных действий, в директории *~/lab2_sos/1* появится ряд файлов, как показано на рисунке 1.10, среди которых будет файл модуля *hello-1.ko*.

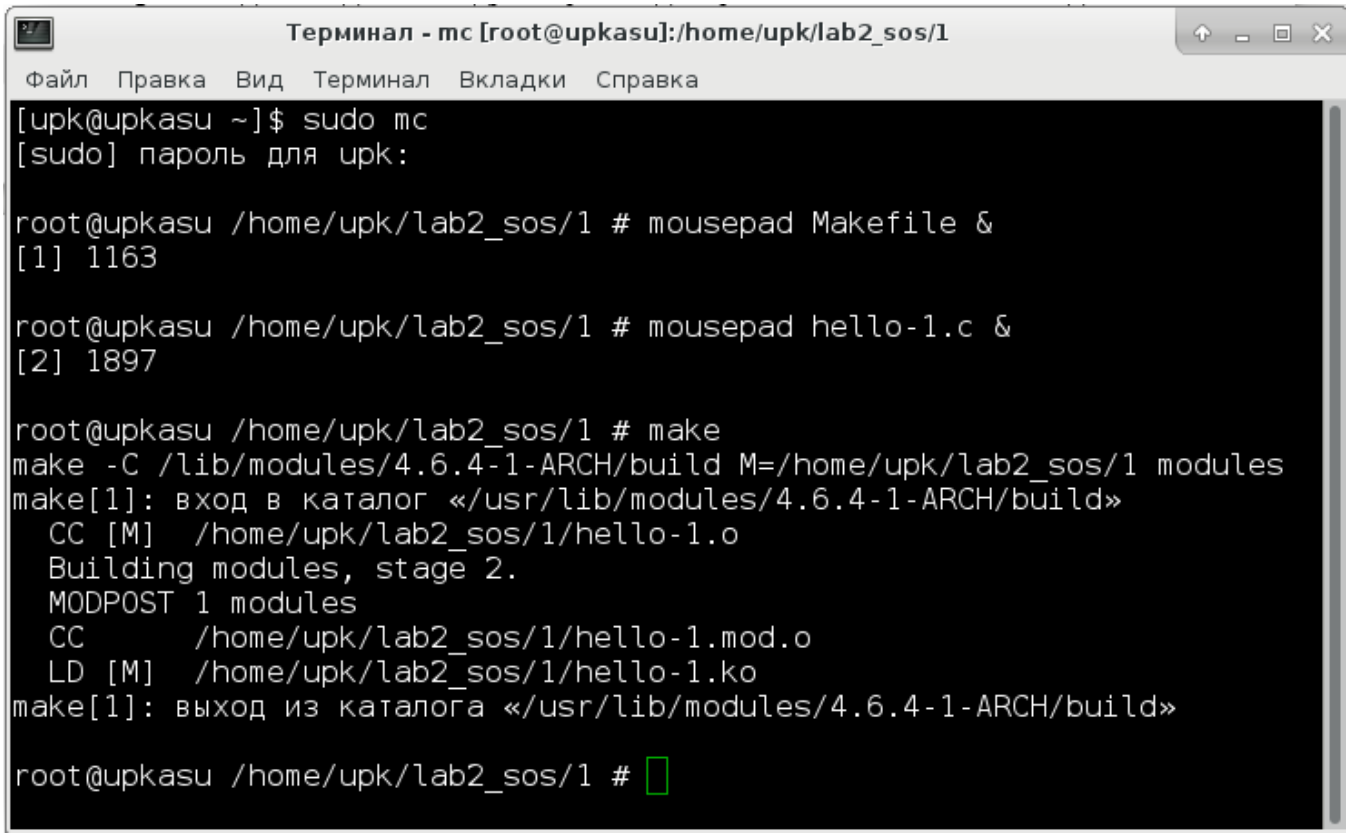

Рисунок 1.10 — Результат работы утилиты make

Чтобы убедиться в правильности создания проекта, достаточно выполнить в командной строке четыре команды, как показано на рисунке 1.11:

- очистить файл системного журнала;
- загрузить модуль;
- выгрузить модуль;
- просмотреть системные сообщения файла журнала.

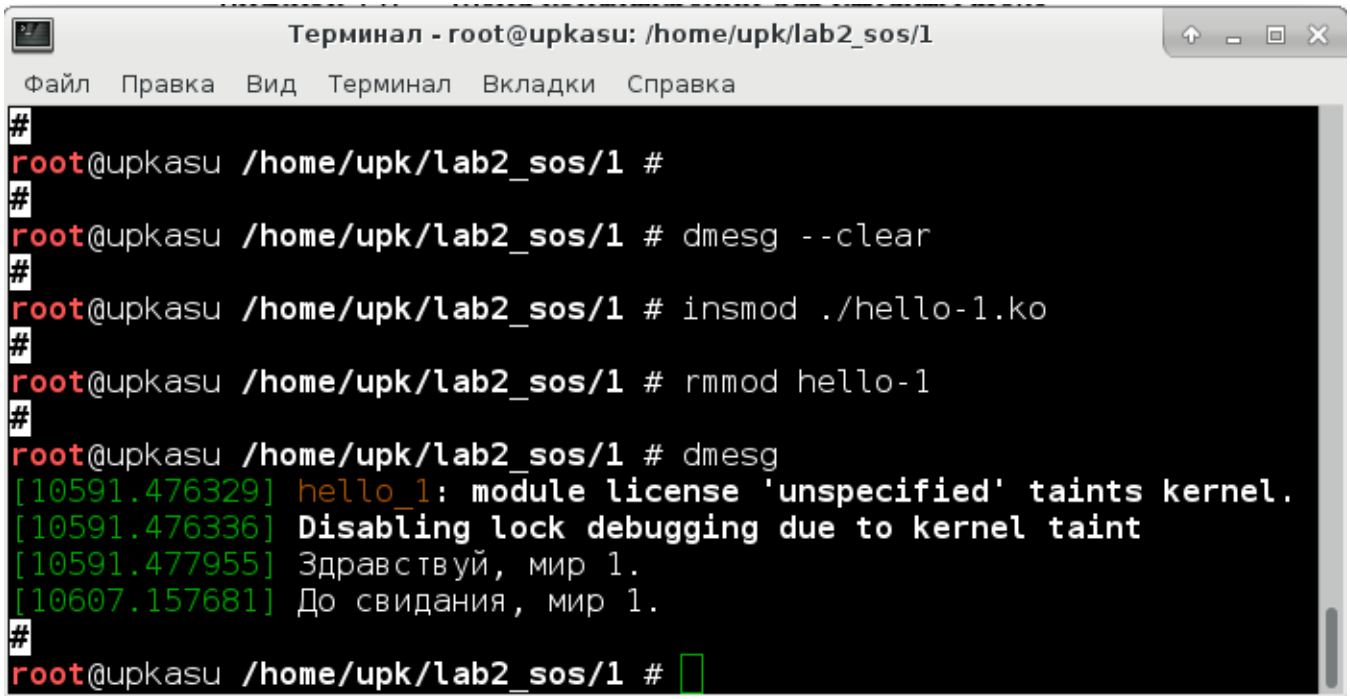

Рисунок 1.11 — Результат проверки работы модуля hello-1.ko

Другой вариант проверки показан на рисунке 1.12, где:

- загружается модуль;
- в файле */proc/kallsyms* ищется его имя.

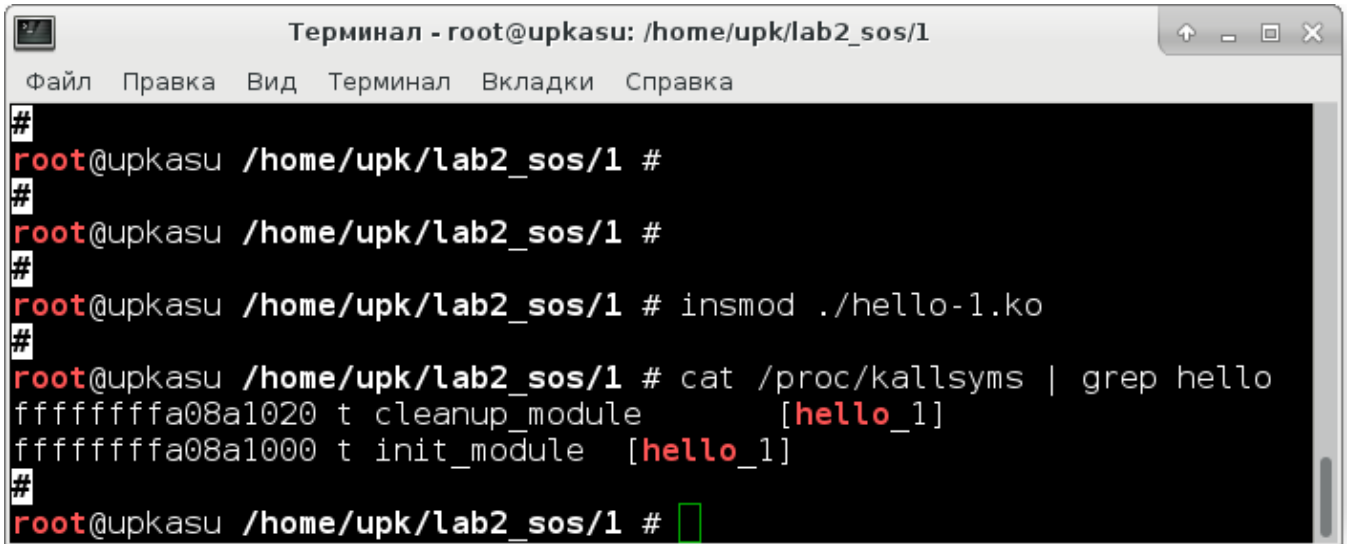

Рисунок 1.12 — Поиск имени модуля в псевдофайле /proc/kallsyms

1.8 Модуль как драйвер

Минимальный функционал модуля ядра — реализация двух функций:

- *инициализация модуля*, - в процессе его загрузки, включая чтение параметров, передаваемых ему утилитами *instmod* или *modprobe*;
- *деинициализация модуля*, - в процессе его выгрузки.

Драйвер — это *загружаемый* или *статический* модуль ядра, который обеспечивает работу трех интерфейсов:

- *внешний интерфейс* в пространство пользователя;
- *внешний интерфейс* с аппаратными средствами ЭВМ;
- *внутренний интерфейс* к API ядра.

Псевдодрайвер — драйвер, реализующий внешний интерфейс в пространство пользователя.

Внешний интерфейс в пространство пользователя.
Это - интерфейс, который модуль устанавливает с «***внешним пространством***» пользователя, обеспечивая видимость ***пользователю*** системных объектов, называемых ***устройствами***.
Устройства — системные объекты ОС, с которыми пользователь может взаимодействовать из своего программного кода или посредством консольных команд системы, через *интерфейсы-связи* с ядром ОС.
Такими интерфейсами-связями служат, например: имена в файловых системах **/dev**, **/proc** или **/sys**, сетевые интерфейсы и сетевые протоколы.

Внешний интерфейс с аппаратными средствами ЭВМ.
Интерфейс, обеспечивающий связь с аппаратной спецификой поддерживаемого оборудования.
Сюда относятся такие механизмы, как техника обработки аппаратных прерываний, обнаружение и работа устройств на шине PCI или обслуживание USB- устройств.

Замечание
Изучение интерфейса взаимодействия с аппартными средствами ЭВМ выходит за рамки нашего курса обучения, поэтому в данной теме рассматриваться не будет.

Внутренний интерфейс к API ядра.
Это - механизмы ядра (***функции*** или ***макросы***), используя которые модуль и реализует два внешних интерфейса, указанных выше.
Благодаря этому интерфейсу модуль может зарегистрировать новый драйвер устройства так, чтобы он стал известен ядру, или запросить динамическое выделение памяти для размещения буферов обмена этого устройства.

Замечание
Внутренний интерфейса к API ядра рассматривается только по необходимости.

Таким образом, основная цель нашего обучения концентрируется на внешнем интерфейсе в пространство пользователя.

Любое устройство в ядре ОС однозначно индентифицируется двумя целочисленными номерами:

- *major* — старший номер, отвечающий за отдельный класс устройств;
- *minor* — младший номер, идентифицирующий устройство в пределах класса;
- *имя устройства* — является второстепенным идентификатором и может быть любым (в пределах возможности конфликта с другими именами).

Все эти характеристики отображаются в специальной файловой системе типа *devfs* (***devtmpfs***), которая монтируется в директорию */dev*.
Например, на рисунке 1.13 показан вывод характерискик жесткого диска, для диска с контроллером ***SATA***.

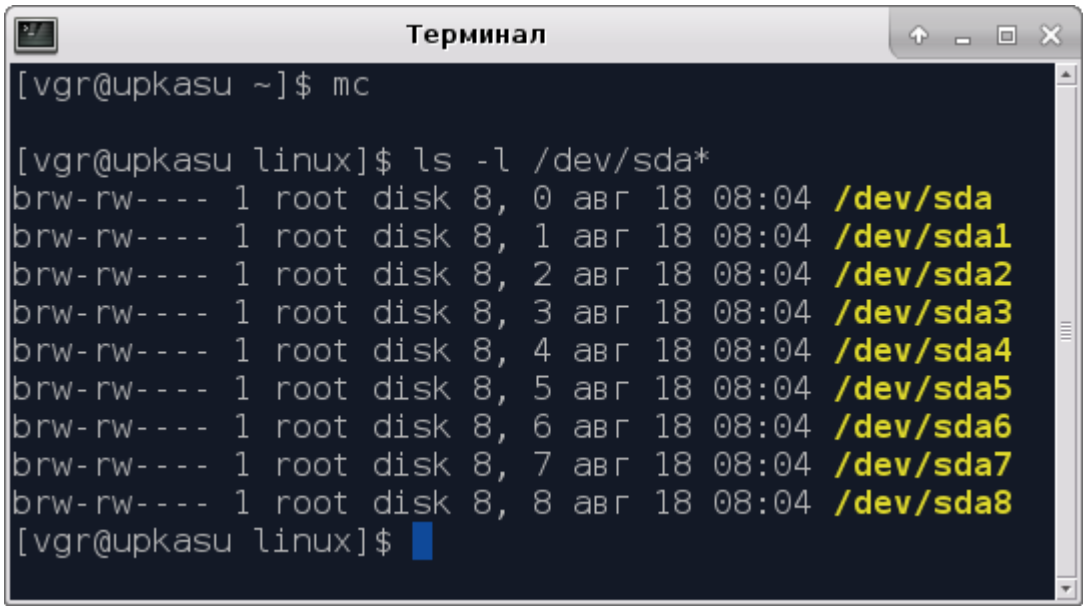

Рисунок 1.13 — Определение maior/minor для диска SATA

Драйвер устройства, являющийся модулем и работающий с конкретным оборудованием ЭВМ, обязан создать эти характеристики в директории **/dev**, *во время выполнения функции инициализации*, но для этого он должен их знать, что в общем случае является проблемой.

Замечание

Общая проблематика современных ОС - распределение *major/minor* номеров устройств.

В разное время и в разных ОС, эта проблема решалась по разному.
Непосредственно для ОС Linux, используются два основных подхода:

- *использование документации* ***Documentation/devices.txt***, поставляемой с исходными кодами ядра;
- *модель динамического распределения номеров*, которая поддерживается файловой системой *sysfs* (**/sys**) и программным проектом ***udev***, обеспечивающим

работу *sysfs* в пользовательском пространстве.

В пространстве пользователя, все устройства ЭВМ разделены на две группы:

- *блочные устройства*, обеспечивающие работу с файловыми системами ОС;
- *символьные устройства*, - все другие устройства, не предполагающие работу с ними - как с фаловыми системами.

Замечание

Любое устройство может рассматриваться как блочное или как символьное, например:

- утилита *dd* работает с жесткими дисками как с символьными устройствами;
- файловые системы *proc* и *sysfs* работают с памятью компьютера как с блочным устройством;
- *псевдоустройства* — вообще могут не работать с оборудованием ЭВМ, а эмитировать такую работу или использовать другие модули (драйвера).

Таким образом, драйвера устройств предоставляют в пространство пользователя два различных внешних интерфеса: *блочный интерфейс* и *символьный интерфейс*.

1.8.1 Драйвер символьного устройства

Драйвер символьного устройства предоставляет в пространство пользователя *символьный интерфейс*.
Он реализуется через ***таблицу файловых операций***, которая является структурой типа *file_operations*, определенную в файле ***<linux/fs.h>***.
Эта структура содержт указатели на функции ядра ОС, которые отвечают за выполнение всех операций с устройством.
Часть этой большой структуры представлена на листинге 1.8.

Листинг 1.8 — Часть структуры типа file_operations

```
struct file_operations {
        struct module *owner;
        loff_t  (*llseek) (struct file *, loff_t, int);
        ssize_t (*read)   (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write)  (struct file *, const char __user *, size_t, loff_t *);
        ...
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        ...
        int (*open)    (struct inode *, struct file *);
        int (*flush)   (struct file  *);
        int (*release) (struct inode *, struct file *);
        ...
   };
```

Загружаемый драйвер символьного устройства должен создать в своей локальной области такую структуру во время процедуры инициализации, что сразу обеспечит ему стандартный набор функциональности.
Драйвер может переопределить все или только часть указателей этой структуры на свои функции, что собственно и используется при написании драйверов.

Вторая по значимости структура *inode_operations*, часть которой показана на листинге 1.9, преднасначена для работы с ***путевым именем*** самого устройства.

Листинг 1.9 — Часть структуры типа inode_operations

```
struct inode_operations {
        int (*create) (struct inode *, struct dentry *, int, struct nameidata *);
   ...
        int (*mkdir)  (struct inode *, struct dentry *,int);
        int (*rmdir)  (struct inode *, struct dentry *);
        int (*mknod)  (struct inode *, struct dentry *, int, dev_t);
        int (*rename) (struct inode *, struct dentry *,
                       struct inode *, struct dentry *);
   ...
   }
```

Детали использования этих структур и правила загрузки/выгрузки драйвера символьного устройства будут рассмотрены в подразделе 1.9.

1.8.2 Драйвер блочного устройства

Интерфейс блочного устройства во многом похож на интерфейс *символьного устройства*.
Он реализуется через **таблицу файловых операций**, которая является структурой типа *block_device_operations*, определенную в файле ***<linux/blkdev.h>*.**
Эта структура содержт указатели на функции ядра ОС, которые отвечают за выполнение всех операций с устройством.
Пример такой структуры представлен на листинге 1.10.

Листинг 1.10 — Пример структуры типа block_device_operations

```
struct block_device_operations {
       int (*open) ( struct block_device *, fmode_t );
       int (*release) ( struct gendisk *, fmode_t );
       int (*locked_ioctl) ( struct block_device *, fmode_t,
                             unsigned, unsigned long );
       int (*ioctl) ( struct block_device *, fmode_t, unsigned, unsigned long );
       int (*compat_ioctl) ( struct block_device *, fmode_t,
                             unsigned, unsigned long );
       int (*direct_access) ( struct block_device *, sector_t,
                              void **, unsigned long * );
       int (*media_changed) ( struct gendisk * );
       unsigned long long (*set_capacity) ( struct gendisk *, unsigned long long );
       int (*revalidate_disk) ( struct gendisk * );
       int (*getgeo)( struct block_device *, struct hd_geometry * );
       struct module *owner;
};
```

Для регистрации и освобождения блочных устройств используются функции внутреннего интерфейса ядра:

```
        int register_blkdev( unsigned major, const char* );
        void unregister_blkdev( unsigned major, const char* );
```

где *major* — старший номер устройства;
<*второй аргумент*> - указатель на имя устройства.

Замечание

В силу большого объема информации, необходимой для полного изучения интерфейсов устройств ЭВМ, обучение по данной теме ограничено только элементами интерфейса символьных устройств.
Студент желающий более подробно изучить работу с драйверами может обратиться за информацией к сайтам:

- *http://tldp.org/LDP/lkmpg/2.6/html/index.html*;
- *http://www.ibm.com/developerworks/ru/library/l-linux_kernel_01/index.html*.

1.9 Динамические устройства

В данном подразделе, как объявленно выше, более подробно рассматриваются вопросы создания *символьных устройств*.
Основное внимание уделяется технологии создания узлов (*inods*) устройств, как основной проблематике любой ОС.

Как рассмотрено в подразделе 1.8, основные функциональные возможности символьного драйвера сосредоточены в ссылках структуры типа *file_operations*, на внутренние функции ядра, реализующие эту функциональность.
Современные дистрибутивы ОС Linux поддерживают язык ***C*** в стандарте синтаксиса ***C99***, поэтому обеспечивают описание функциональных изменений в формате, который демонстрируется листингом 1.11:

Листинг 1.11 — Пример объяления изменений в структуре типа file_operations

```
        struct file_operations fops = {
                .read    = device_read,
                .write   = device_write,
                .open    = device_open,
                .release = device_release
        };
```

Здесь объявлена структура **fops** типа *file_operations*, которая заменяет стандартные ссылки на функции *read(), write(), open()* и *release()* ссылками на локальные функции модуля *device_read(), device_write(), device_open()* и *device_release()*.
Что касается **проблематики создания самих устройств**, то ограничимся только двумя технологиями, озаглавленными как:

- *современный класический подход*, представленный модифицированным примером дравера: *http://tldp.org/LDP/lkmpg/2.6/html/x569.html*;
- *альтернативный динамический подход*, демонстрирующий идеи публикации сайта: *http://www.ibm.com/developerworks/ru/library/l-linux_kernel_20/*.

1.9.1 Современный классический подход

Основная проблематика создания устройства - определение его старшего номера *major*.
Стандартные средства современных ОС Linux, для регистрации и удаления устройств из пространства ядра ОС, используют API, определенный в заголовочном файле <*linux/fs.h*>:

```
int register_chrdev(unsigned int major, const char *name,  struct
file_operations *fops);
int unregister_chrdev(unsigned int major, const char  *name);
```

где *major* — старший номер устройства;
name — имя устройства;
fops — ссылка на структуру, которая демонстрируется листингом 1.11;

Такой подход **позволяет**:

- *создавать* символьное устройство по его имени, относящемуся к старшему номеру устройства *major*;
- *запрашивать* старший номер устройства *major*, когда первый аргумент фнукции *register_chrdev()* равен ***нулю***;
- *обеспечить* работу с устройством по его имени;
- *безопасно удалить* устройство, после выгрузки драйвера.

Пример реализации такого драйвера приведен на листинге 1.12.

Листинг 1.12 — Пример символьного драйвера классического варианта

```
/*
 *  chardev-1.c: Открывает символьное устройство
 *  Модифицировано: Reznik, 18.08.2016
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>        /* для put_user */

/*
 *  Прототипы
 */
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode     *, struct file *);
static int device_release(struct inode  *, struct file *);
static ssize_t device_read(struct file  *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev-1" /* Имя устройства добаляемое в /proc/devices   */
#define BUF_LEN 1024            /* Максимальная длина сообщения в устройстве   */

/*
 * Глобальные переменные, декларированные как static, глобальные внутри файла.
 */

static int Major;           /* Major-номер, для нашего устройства */
static int Device_Open = 0; /* Is device open?  Используется, чтобы ограничить
                             * множественный доступ к устройству */
static char msg[BUF_LEN];   /* The msg the device will give when asked */
static char *msg_Ptr;

static struct file_operations fops = {
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release
};

/*
 * Эта функция вызывается, когда модуль загружается.
 */
```

```
int init_module(void)
{
        Major = register_chrdev(0, DEVICE_NAME, &fops);

        if (Major < 0) {
          printk(KERN_ALERT "Регистрация симв. устройства failed для %d\n", Major);
          return Major;
        }

        printk(KERN_INFO "Был присвоен major-номер %d.\n", Major);
        printk(KERN_INFO "To talk to the driver, create a dev file with\n");
        printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
        printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
        printk(KERN_INFO "the device file.\n");
        printk(KERN_INFO "Remove the device file and module when done.\n");

        return SUCCESS;
}

/*
 * Эта функция вызывается, когда модуль выгружается.
 */
void cleanup_module(void)
{
        /*
         * Unregister the device
         */
        unregister_chrdev(Major, DEVICE_NAME);
}

/*
 * Методы:
 *
 * Вызывается, когда процесс открывает файл устройства, подобно:
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
        static int counter = 0;

        if (Device_Open)
                return -EBUSY;

        Device_Open++;
        sprintf(msg, "Я уже сообщил тебе %d раз Hello world!\n", counter++);
        msg_Ptr = msg;
        try_module_get(THIS_MODULE);

        return SUCCESS;
}

/*
 * Вызывается, когда процесс закрывает файл устройства.
 */
static int device_release(struct inode *inode, struct file *file)
{
        Device_Open--;          /* Мы теперь готовы для следующего вызова */

        /*
         * Decrement the usage count, or else once you opened the file, you'll
         * never get get rid of the module.
         */
        module_put(THIS_MODULE);

        return 0;
```

```
}

/*
 * Вызывается, когда процесс, который уже открыл файл устройства,
 * пытается читать из него.
 */
static ssize_t device_read(struct file *filp,   /* смотри include/linux/fs.h   */
                           char *buffer,        /* буфер для данных */
                           size_t length,       /* длина буфера     */
                           loff_t * offset)     /* смещение     */
{
        /*
         * Число байт, реально записанных в буфер
         */
        int bytes_read = 0;

        /*
         * Если конец сообщения, то
         * return 0, обозначая конец файла
         */
        if (*msg_Ptr == 0)
                return 0;

        /*
         * Фактическая посылка данных в буфер
         */
        while (length && *msg_Ptr) {

                /*
                 * Буфер находится в user data segment, not the kernel
                 * segment so "*" assignment won't work.  Мы можем использовать
                 * put_user, которая копирует данные из сегмента данных ядра в
                 * сегмент данных пользователя.
                 */
                put_user(*(msg_Ptr++), buffer++);

                length--;
                bytes_read++;
        }

        /*
         * Большинство функций чтения возвращают число байт, посланных в буфер
         */
        return bytes_read;
}

/*
 * Вызывается, когда процесс записывает в файл устройства, например:
 *  echo "hi" > /dev/chardev-1
 */
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
        printk(KERN_ALERT "Извините, эта операция не поддерживается.\n");
        return -EINVAL;
}
```

Замечание

Операция выгрузки драйвера может завершиться с ошибкой, если устройство занято каким-либо обращением к нему.

Чтобы контролировать использование устройства (драйвера), следует пользоваться следующими функциями API ядра:

- `try_module_get(THIS_MODULE);` - увеличивает количесво использований драйвера;
- `module_put(THIS_MODULE);` - уменьшает количесво использований драйвера.

1.9.2 Альтернативный динамический подход

Широкое использование практики псевдодрайверов, которые имитируют работу только одного устройства, стала применяться техника регистрации ***misc drivers***.

Misc drivers (*miscellaneous*, интерфейс смешанных устройств) — основан на идее:

- *использования одного старшего номера* устройства *major=10*;
- *динамическое выделение младшего номера* устройства *minor* ядром ОС, во время регистрации драйвера.

Для примера, на рисунке 1.14 показан текущий список таких драйверов.

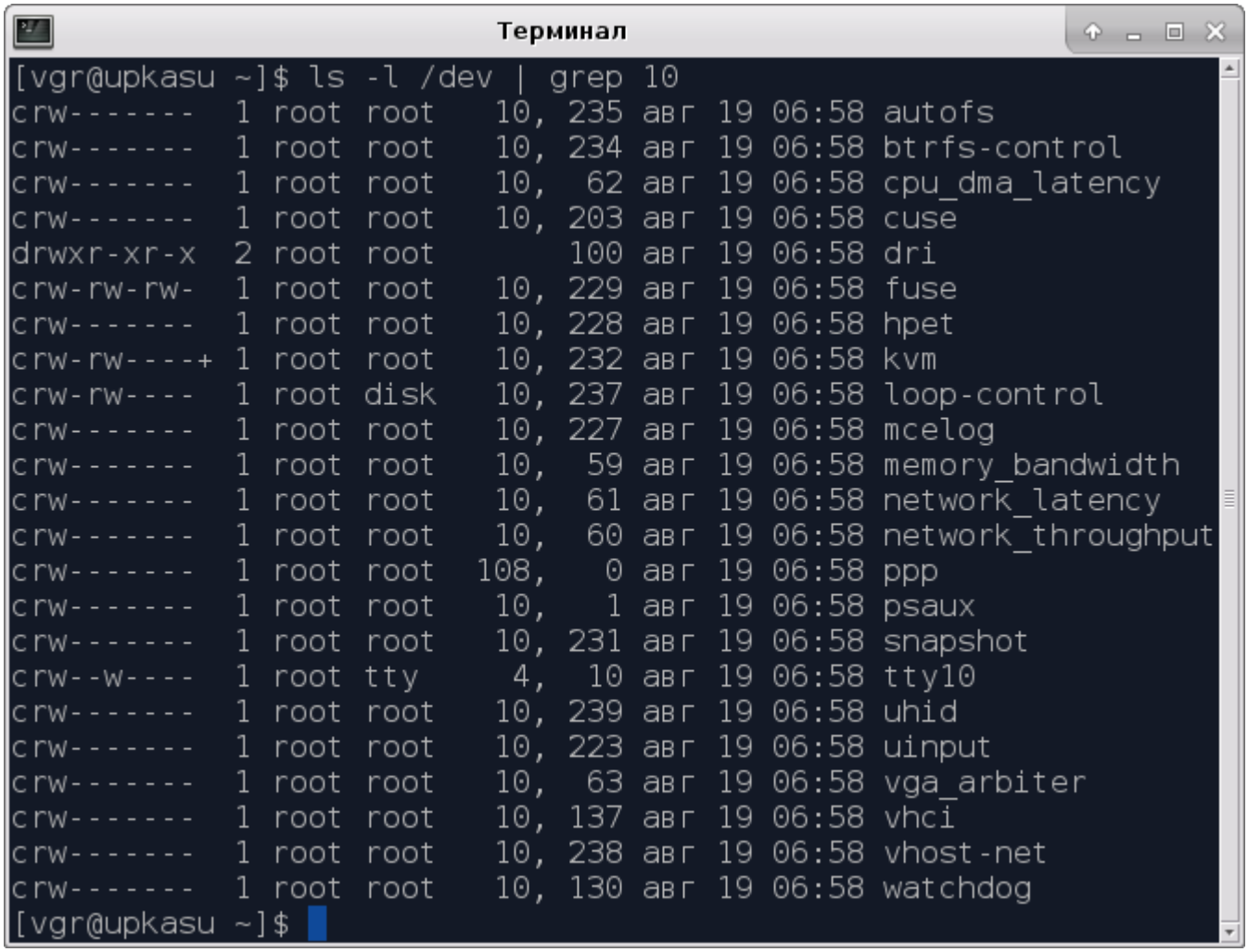

Рисунок 1.14 — Список динамически созданных устройств

Для упрощения реализации таких устройств применяется специальный интерфейс, определенный в файле <***linux/miscdevice.h***>, который использует только две функции API ядра ОС:

```
        extern int  misc_register  (struct miscdevice *misc);
        extern void misc_deregister(struct miscdevice *misc);
```

где *misc* — указатель на структуру вида:

```
        struct miscdevice  {
          int    minor;
          const  char *name;
          const  struct file_operations *fops;
          struct list_head list;
          struct device *parent;
          struct device *this_device;
          const  struct attribute_group **groups;
          const  char *nodename;
          umode_t mode;
        };
```

Минимальный код для регистрации такого драйвера представлен на листинге 1.13.

Листинг 1.13 — Пример минимального кода регистрации драйвера

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>       /* для put_user */
#include <linux/miscdevice.h>  /* API регистрации и удаления устройства */

static int minor = 0;          /* для сохранения minor */

/*
 * Переопределение функций: device_read(), ..., device_release()
 */
static const struct file_operations misc_fops = {
        .owner   = THIS_MODULE,
        .read    = device_read,
        .write   = device_write,
        .open    = device_open,
        .release = device_release
};

/*
 * Итоговая структура для регистрации
 */
static struct miscdevice misc_dev = {
   MISC_DYNAMIC_MINOR,    // Автоматически выбираемое устройство
   "own_misc_dev",        // Имя устройства
   &misc_fops             // Ссылка на структуру типа file_operations
};

/*
 * Функция инициализации модуля, обеспечивающая регистрацию символьного драйвера
 */
static int __init dev_init( void ) {
   int ret;
```

```
   if( minor != 0 ) misc_dev.minor = minor;
   ret = misc_register( &misc_dev );
   if( ret ) printk( KERN_ERR "=== Unable to register misc device\n" );
   return ret;
}

/*
 * Выгрузка модуля с удалением символьного драйвера
 */
static void __exit dev_exit( void ) {
   misc_deregister( &misc_dev );
}

/*
 * Реализация функций: device_read(), ..., device_release()
 */
...
...
```

Реализовав и загрузив этот модуль, мы увидим его среди списка драйверов, показанных на рисунке 1.14. Кроме того:

- *имя* его появится в псевдофайле */proc/devices*;
- *имя* драйвера будет зарегистрировано в едином классе *misc* файловой системы *sysfs*, по адресу */sys/class/misc/own_misc_dev*;
- *локальные и глобальные имена* модуля появятся в псевдофайле */proc/kallsyms*.

Замечание

Данный подход обеспечивает регистрацию только *символьных* драйверов.
Имеются и другие, более изощренные, подходы регистрации символьных устройств, о которых будет упомянуто в следующей теме.

1.10 Управление устройствами

Долгое время операции *ioctl()* были основной техникой выполнения нестандартных действий над устройством, не только в операционной системе UNIX/Linux, но и в других системах, например, в той же MS Windows.
Важно, что эти операции являются *опасными*: при их выполнении отклоняется контроль типизации параметров, и в качестве параметра может быть передана любая структура.
Серьёзной альтернативой управляющих операций *ioctl()* является ***чтение/запись*** информации через псевдоимена файловой системы */proc* и */sys.*

Непосредственная реализация драйвера, использующего управляющие операции *ioctl()*, не вызывает затруднений. Для этого необходимо лишь реализовать свой обработчик этого API ядра, например, как показано на листинге 1.14.

Листинг 1.14 — Пример реализации кода обработчика ioctl()

```
...
// Структура переназначения обработчиков драйвера
static const struct file_operations hello_fops = {
   .owner = THIS_MODULE,
   .open = dev_open,
   .release = dev_release,
   .read  = dev_read,
   .ioctl = dev_ioctl
};

// Текст обработчика ioctl()
static int dev_ioctl( struct inode *n, struct file *f,
                      unsigned int cmd, unsigned long arg ) {
   if( ( _IOC_TYPE( cmd ) != IOC_MAGIC ) )
      return -ENOTTY;
   switch( cmd ) {
      case IOCTL_GET_STRING:
         if( copy_to_user( (void*)arg, hello_str, _IOC_SIZE( cmd ) ) )
            return -EFAULT;
         break;
      default:
         return -ENOTTY;
   }
   return 0;
}
...
```

Замечание

В современных ядрах ОС Linux, стандартный обработчик *ioctl()*, указанный в структуре *file_operations*, замен двумя другими:

```
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl)   (struct file *, unsigned int, unsigned long);
```

2 Лабораторная работа №2

Тема: Разработка модулей ядра ОС

Выполнение данной лабораторной работы предполагает, что студент изучил теоретический материал, изложенный в разделе 1 данного учебного пособия.

Общая цель лабораторной работы — получение начальных практических навыков самостоятельной разработки загружаемых модулей ядра и симовольных драйверов, на примере среды ОС Linux, а также закрепление теоретического материала по данной тематике.

Общие организационные вопросы, связанные с выполнением лабораторной работы, предполагают:

- успешную загрузку студентом ОС УПК АСУ, подключение личного архива на flshUSB по теме *sos*, а также переход в среду пользователя *upk*;
- создание в домашней директории каталога *~/lab2_sos*, необходимого для локализации результатов выполненых работ;
- следование заданиям, изложенным в подразделах 2.1 и 2.2;
- проведение учебных работ, с обязательным отражением их результатов в индивидуальных отчетах.

2.1 Управление загружаемыми модулями

Для начала, следует кратко познакомиться с утилитой *make*, которая является основным инструментом разработчика, используя команды:

```
man  make
info make
```

Далее, следует перейти к выполнению конкретных заданий.

2.1.1 Простейший загружаемый модуль ядра

Перейти в *подраздел 1.7* данного руководства, повторно изучить его учебный материал, а затем выполнить разработку и исследование простейшего модуля, следуя изложенным там инструкциям.

Результаты исследования изложить в личном отчете.

Предполагается, что выполнив это задание, студент освоил основы работы с утилитой *make*.

2.1.2 Макросы и документация модулей

Цель задания:

- *освоить технологию макросов*, обеспечивающих нужные имена модуля в пространсве ядра ОС;
- *освоить технику документирования* информации о макросе.

Подготовительные действия:

- повторяем *учебный материал подраздела 1.1* данного учебного пособия;
- запускаем терминал и выполняем команду: `sudo mc` ;
- создаем каталог *~/lab2_sos/2* и используем его как рабочую директорию для выполнения текущего задания;
- с помощью редактора *mousepad* создаем файл *hello-2.c* и переносим в него текст листинга 1.3 (стр. 9);
- в рабочей директории: создаем файл *hello-3-4.c* и переносим в него текст листинга 1.4 (стр. 10);
- в рабочей директории: создаем файл *Makefile*, содержание которого показано на рисунке 2.1.

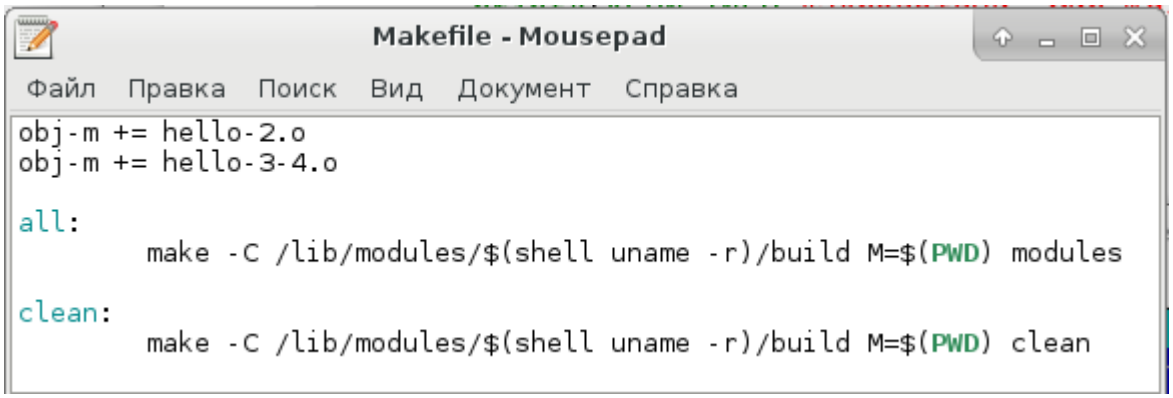

Рисунок 2.1 — Makefile для проекта №2

Создание модулей выполняется командой:

```
make
```

В результате, рабочая директория должна содержать модули *hello-2.ko* и *hello-3-4.ko*.

Тест 1, для модуля *hello-2.ko* выполняется также, как и для первого проекта (как показано на рисунке 2.2):

- загрузка и выгрузка модуля;
- просмотр файла журнала системных сообщений.

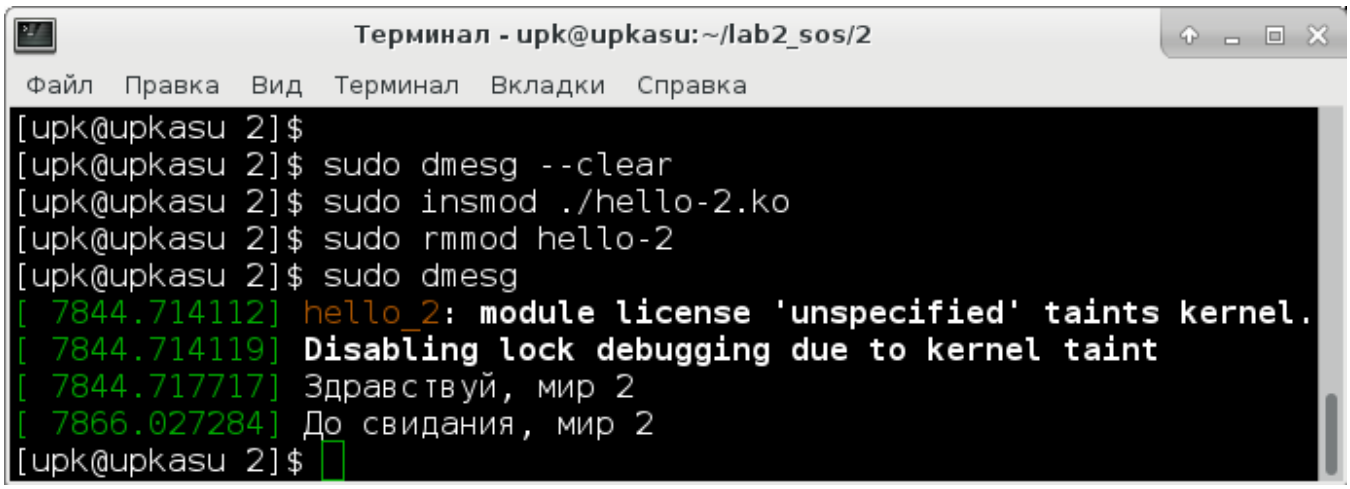

Рисунок 2.2 — Тест 1 для модуля hello-2.ko

Тест 2, для модуля *hello-2.ko*, демонстрирует объекты ядра ОС, как показано на рисунке 2.3.

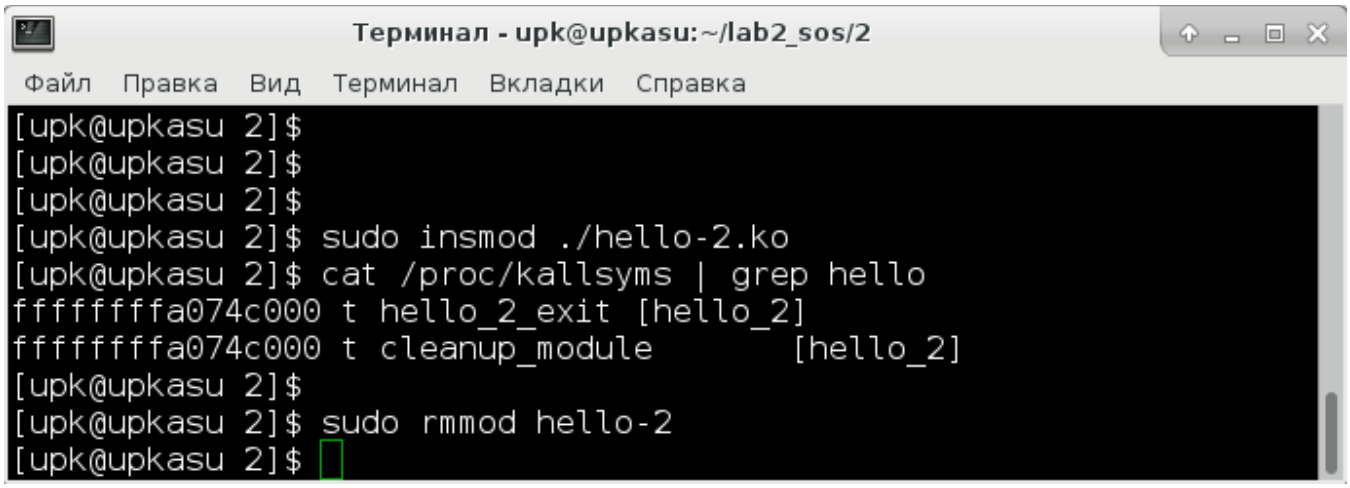

Рисунок 2.3 — Тест 2 для модуля hello-2.ko

Для модуля *hello-3-4.ko* можно проводить тесты, как и для модуля *hello-2.ko*, но его назначение — демострировать технику документирования, определенную его макросами.

Тест 3, для модуля *hello-3-4.ko*, демонстрирует возможности макросов документирования, как показано на рисунке 2.4:

- загружаем модуль;
- проверяем наличие его в списке загруженных;
- выводим информацию о модуле;
- выгружаем модуль.

Для проверки информации о модуле — *его не обязательно загружать!*

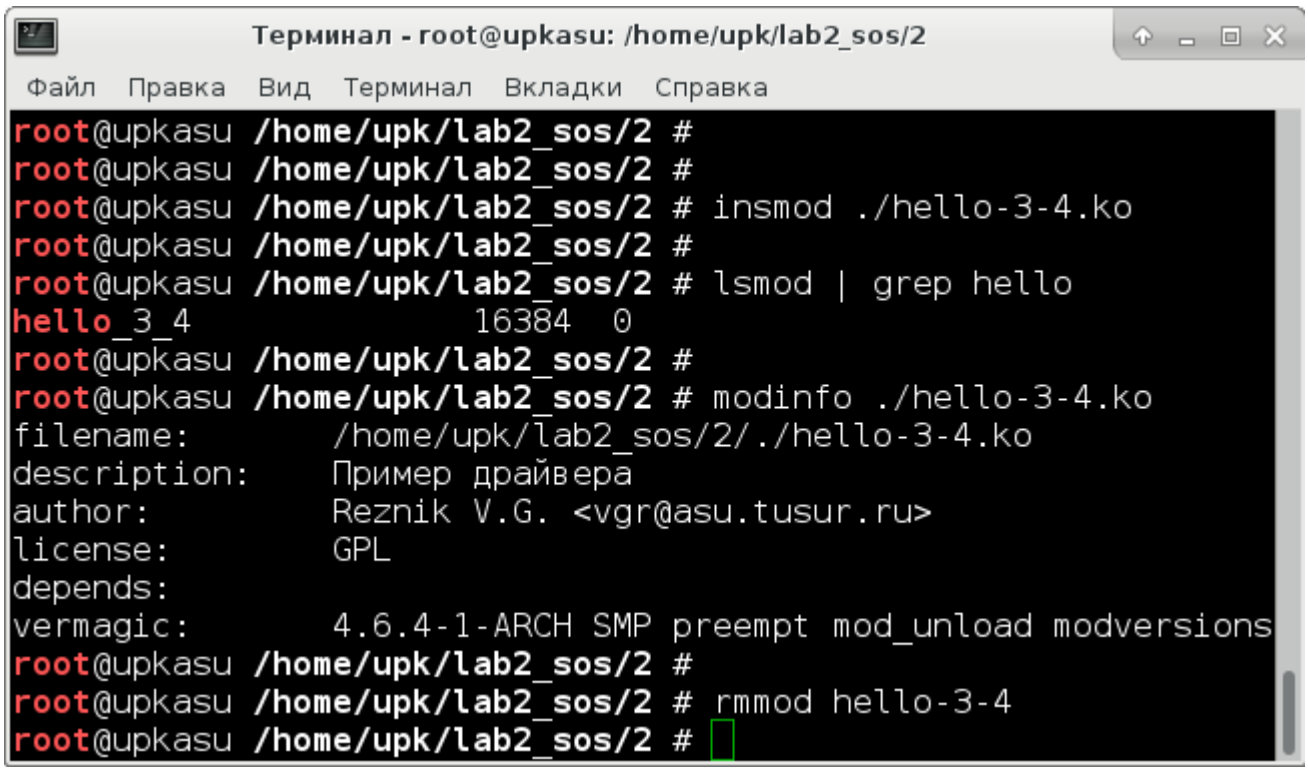

Рисунок 2.4 — Тест 3 для модуля hello-3-4.ko

2.1.3 Модули, читающие параметры загрузки

Цель задания: освоить технологию передачи параметров модулю.

Подготовительные действия:

- повторяем *учебный материал подраздела 1.6* данного учебного пособия;
- запускаем терминал и выполняем команду: `sudo mc` ;
- создаем каталог *~/lab2_sos/3* и используем его как рабочую директорию для выполнения текущего задания;
- с помощью редактора *mousepad* создаем файл *mod-params.c* и переносим в него текст листинга 1.7 (стр. 27);
- в рабочей директории: создаем файл *Makefile*, содержание которого показано на рисунке 2.5.

Замечание

Обратите внимание, что текст модуля *не содержит функции выгрузки модуля*!
Это возможно потому, что модуль никогда не сможет загрузиться, поскольку всегда возвращает отрицательное значение, но он обязательно выведет в системный журнал свои сообщения, необходимые нам для тестирования.

Создаем модуль командой: *make*
В результате, рабочая директория должна содержать модуль *mod-params.ko*.

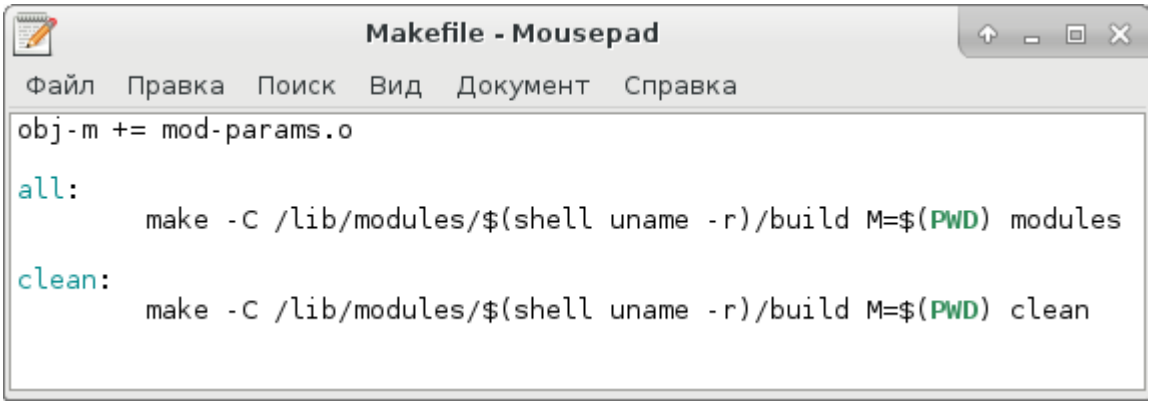

Рисунок 2.5 — Makefile для модуля mod-params.ko: проект №3

Тест 1, для модуля *mod-params.ko* проведем, просто запустив модуль без параметров (как показано на рисунке 2.6):

- очистка системного журнала;
- загрузка модуля;
- просмотр файла журнала системных сообщений.

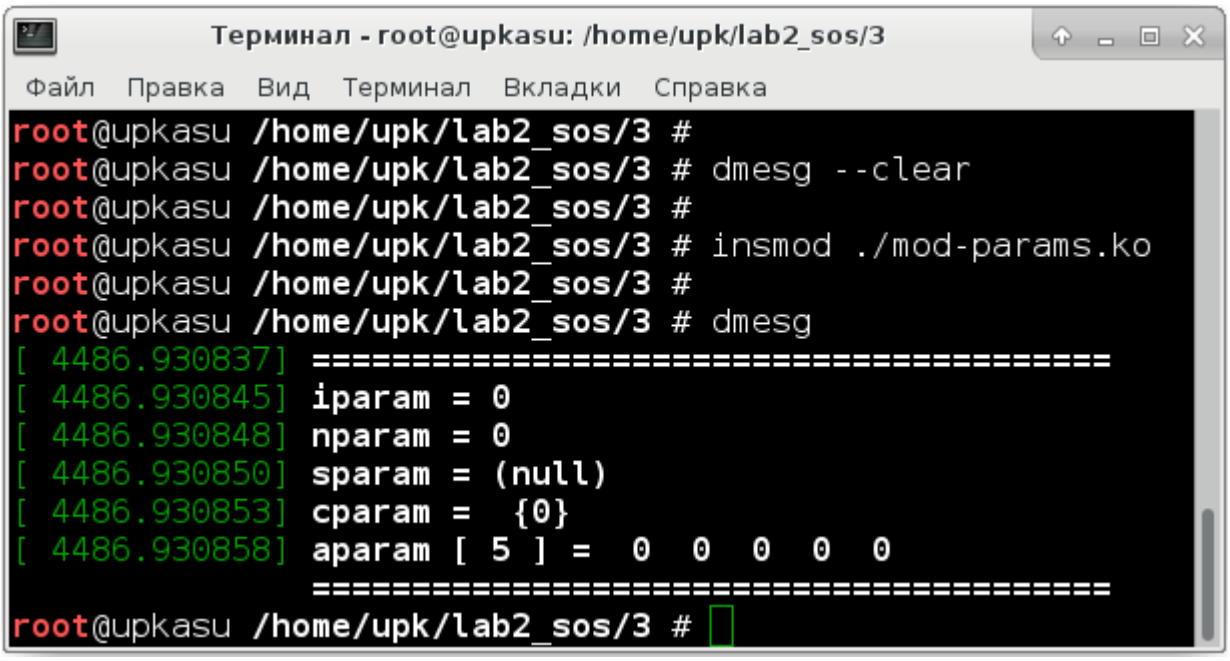

Рисунок 2.6 — Тест 1 для модуля mod-params.ko

Из рисунка видно, что все параметры модуля имеют значения, установленные в самом модуле по умолчанию.

Замечание

Тестирование модулей может оказаться достаточно сложным занятием, особенно если необходимо задавать значения многим параметрам, поэтому следует создавать специальные сценарии для различных вариантов теста.

Подготовим сценарий *Test2.sh*, показанный на рисунке 2.7 и предназначенный для тестирования модуля *mod-params.ko*, который разместим в рабочей директории проекта, а затем запустим, как показано на рисунке 2.8.

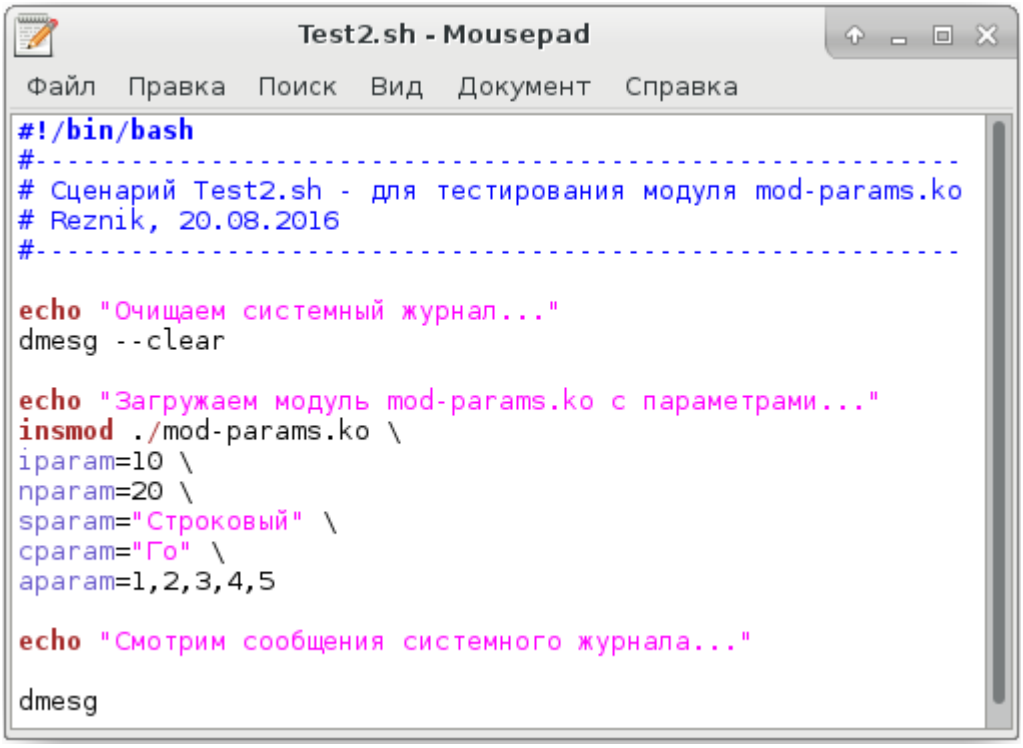

Рисунок 2.7 — Текст тестового сценария Test2.sh

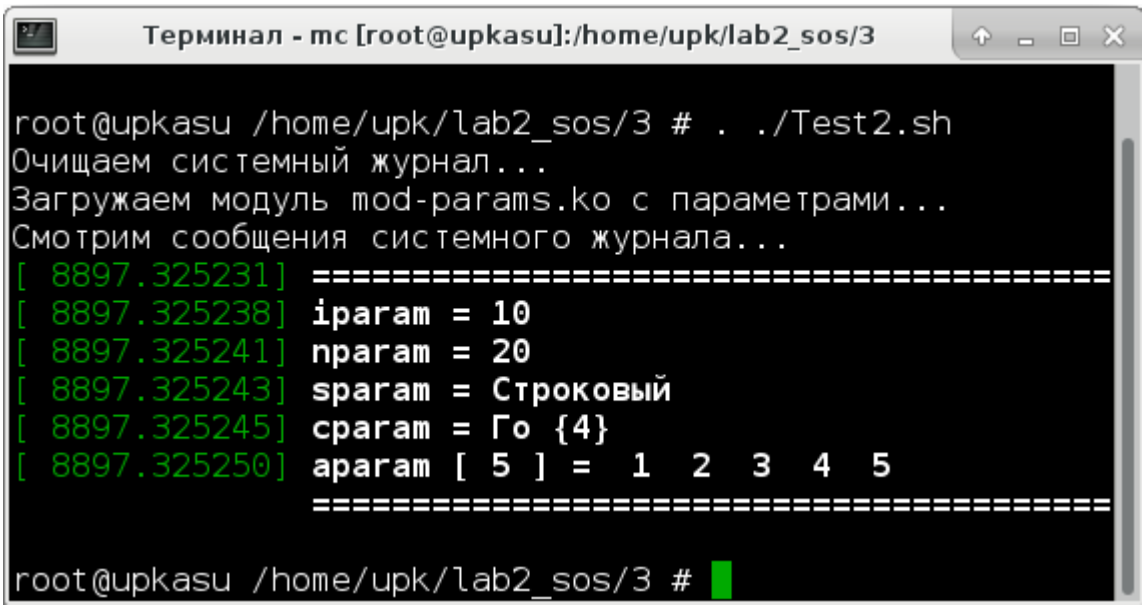

Рисунок 2.8 — Результат вывода сценария Test2.sh

Студенту следует, хорошо разобраться с исходным текстом модуля, а затем - провести эксперименты, изменяя параметры запуска модуля.
Результаты исследования отразить в отчете.

2.2 Символьные драйверы

Эта часть лабораторной работы посвящена символьным драйверам.
Учебная цель — освоить современные методы регистрации драйверов ОС.
Лабораторная часть обучения ограничена исследованием двух примеров:

- регистрация классического символьного устройства;
- регистрация динамически определяемого устройства.

Предварительно, следует внимательно изучить учебный материал подразделов 1.8 и 1.9 данного учебного пособия.

2.2.1 Тест классического символьного устройства

Учебная работа ведется в рамках проекта №4.
Подготовительные действия:

- повторяем *учебный материал части 1.9.1* данного учебного пособия;
- создаем каталог *~/lab2_sos/4* и используем его как рабочую директорию для выполнения текущего задания;
- с помощью редактора *mousepad* создаем файл *chardev-1.c* и переносим в него текст листинга 1.7 (стр. 38);
- в рабочей директории: создаем файл *Makefile*, содержание которого показано на рисунке 2.9.

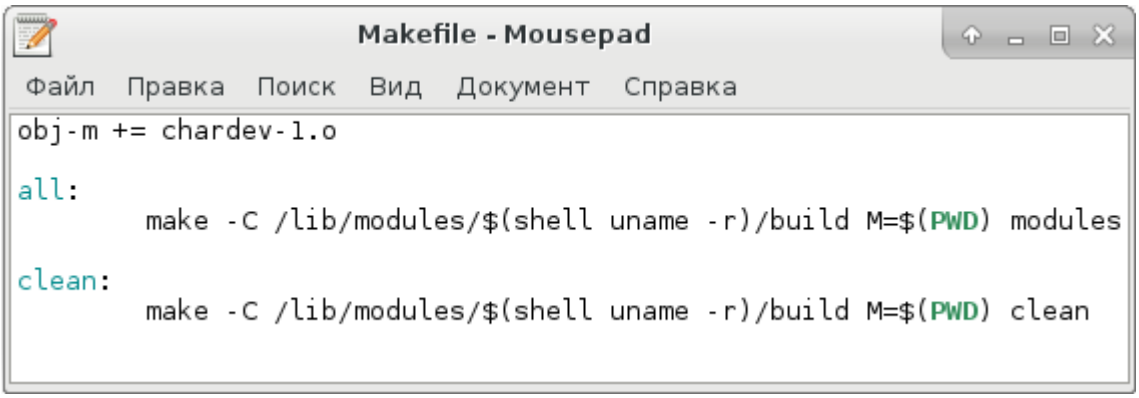

Рисунок 2.9 — Makefile для модуля chardev-1.ko: проект №4

Создаем модуль командой: *make*
В результате, рабочая директория должна содержать модуль *chardev-1.ko*.

Тест 1, для модуля *chardev-1.ko* проведем по классической схеме, как показано на рисунке 2.10:

- очистка системного журнала;
- загрузка модуля;
- просмотр файла журнала системных сообщений.

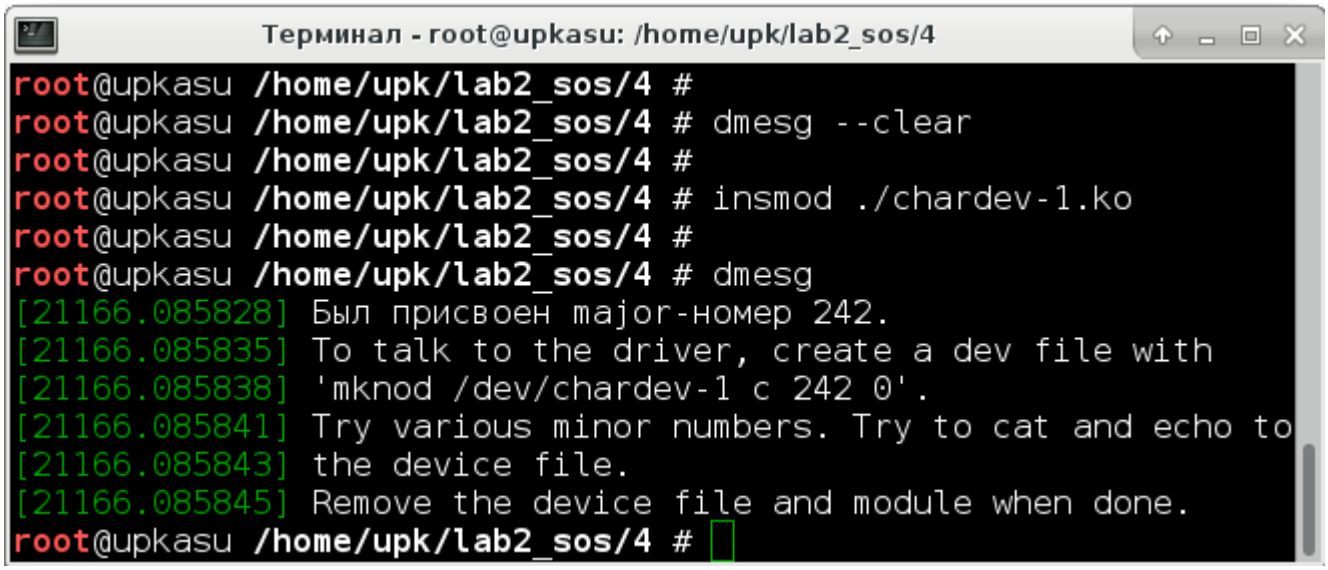

Рисунок 2.10 — Результат выполнения теста №1 для модуля chardev-1.ko

Нам предлагается создать узел устройства командой:

```
mknod /dev/chardev-1 c 242 0
```

после чего:

- в директории */dev* появится символное устройство с именем *chardev-1*;
- мы можем работать с этим устройством, например, с помощью команд *cat* и *echo*.

Создадим такое устройство и применим команды, как показано на рисунке 2.11.

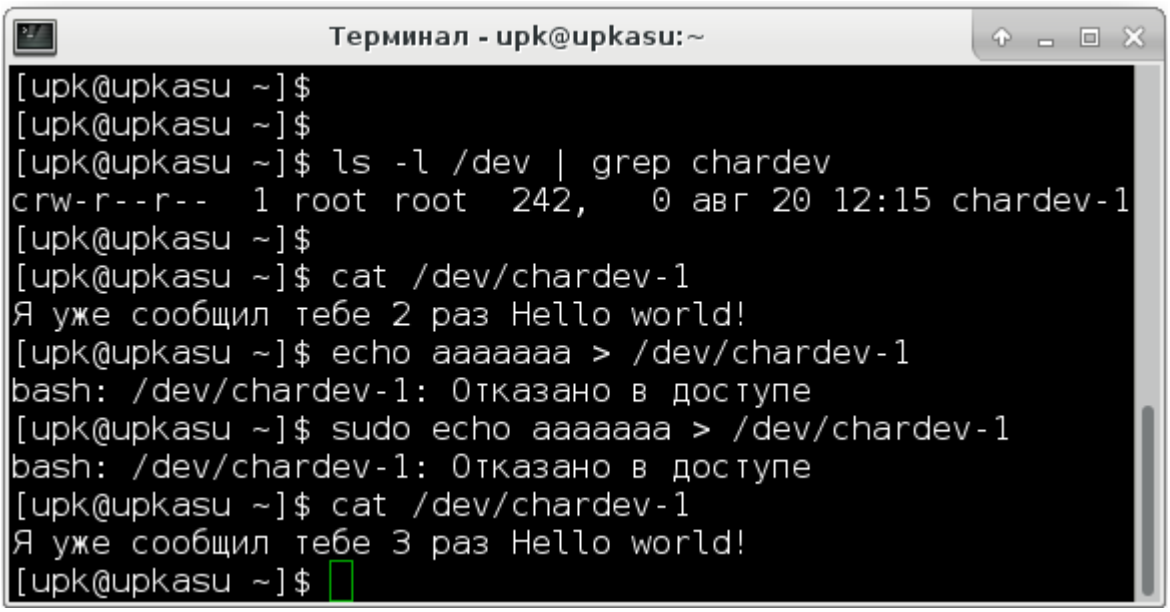

Рисунок 2.11 — Результат работы с устройством chardev-1

2.2.2 Тест динамического устройства

Учебная работа ведется в рамках проекта №5.
Подготовительные действия:

- повторяем *учебный материал части 1.9.2* данного учебного пособия;
- создаем каталог *~/lab2_sos/5* и используем его как рабочую директорию для выполнения текущего задания;
- изучаем исходный текст символьного драйвера, преставленного далее на листинге 2.1;
- с помощью редактора *mousepad* создаем файл *chardev-2.c* и переносим в него текст листинга 2.1;
- в рабочей директории: самостоятельно создаем *Makefile* для нового драйвера *chardev-2.ko*, а затем — создаем сам драйвер.

Листинг 2.1 — Исходный текст драйвера динамического устройства

```
/*
 *  chardev-2.c: Драйвер динамического символьного устройства
 *  Модифицировано: Reznik, 18.08.2016
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>        /* для put_user */
#include <linux/miscdevice.h>  /* API регистрации и удаления устройства */

/*
 *  Прототипы
 */
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode     *, struct file *);
static int device_release(struct inode  *, struct file *);
static ssize_t device_read(struct file  *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev-2" /* Имя устройства добаляемое в /proc/devices   */
#define BUF_LEN 1024            /* Максимальная длина сообщения в устройстве   */

/*
 * Глобальные переменные, декларированные как static, глобальные внутри файла.
 */

static int minor = 0;       /* Для сохранения minor */
static int Device_Open = 0; /* Is device open?  Используется, чтобы ограничить
                             * множественный доступ к устройству */
static char msg[BUF_LEN];   /* The msg the device will give when asked */
static char *msg_rPtr;      /* Указатель для чтения */
static char *msg_wPtr;      /* Указатель для записи */

/*
 * Переоаределение функций: device_read(), ..., device_release()
 */
static struct file_operations misc_fops = {
      .owner   = THIS_MODULE,
        .read    = device_read,
```

```
        .write   = device_write,
        .open    = device_open,
        .release = device_release
};

/*
 * Итоговая структура для регистрации
 */
static struct miscdevice misc_dev = {
   MISC_DYNAMIC_MINOR,    // Автоматически выбираемое устройство
   DEVICE_NAME,           // Имя устройства
   &misc_fops             // Ссылка на структуру типа file_operations
};

/*
 * Эта функция вызывается, когда драйвер загружается.
 */
int init_module(void)
{
   int ret;
   ret = misc_register( &misc_dev );
   minor = misc_dev.minor;
   if( ret ) printk( KERN_ERR "=== Не могу зарегистрировать misc device\n" );
   printk(KERN_INFO "Драйвер chardev-2.ko - стартовал!\n");
   sprintf(msg, "Начальная запись в буфере!\n");
   return ret;
}

/*
 * Эта функция вызывается, когда драйвер выгружается.
 */
void cleanup_module(void)
{
        /*
         * Unregister the device
         */
      printk(KERN_INFO "Драйвер chardev-2.ko - завершил работу!\n");
      misc_deregister( &misc_dev );
}

/*
 * Методы:
 *
 * Вызывается, когда процесс открывает файл устройства, подобно:
 * "cat /dev/chardev-2"
 */
static int device_open(struct inode *inode, struct file *file)
{
        static int counter = 0;

        if (Device_Open)
                return -EBUSY;

        Device_Open++;
        printk("Устройство %s открывается %d-й раз!\n",
            DEVICE_NAME, ++counter);
        msg_rPtr = msg;
        msg_wPtr = msg;
        try_module_get(THIS_MODULE);

        return SUCCESS;
}

/*
 * Вызывается, когда процесс закрывает файл устройства.
 */
```

```
static int device_release(struct inode *inode, struct file *file)
{
        Device_Open--;          /* Мы теперь готовы для следующего вызова */

        /*
         * Decrement the usage count, or else once you opened the file, you'll
         * never get get rid of the module.
         */
        module_put(THIS_MODULE);

        return 0;
}

/*
 * Вызывается, когда процесс, который уже открыл файл устройства,
 * пытается читать из него.
 */
static ssize_t device_read(struct file *filp,   /* смотри include/linux/fs.h   */
                           char *buffer,        /* буфер для данных */
                           size_t length,       /* длина буфера     */
                           loff_t * offset)     /* смещение     */
{
        /*
         * Число байт, реально записанных в буфер
         */
        int bytes_read = 0;

        /*
         * Если конец сообщения, то
         * return 0, обозначая конец файла
         */
        if (*msg_rPtr == 0)
                return 0;

        /*
         * Фактическая посылка данных в буфер
         */
        while (length && *msg_rPtr) {

                /*
                 * Буфер находится в user data segment, not the kernel
                 * segment so "*" assignment won't work.  Мы можем использовать
                 * put_user, которая копирует данные из сегмента данных ядра в
                 * сегмент данных пользователя.
                 */
                put_user(*(msg_rPtr++), buffer++);

                length--;
                bytes_read++;
        }

        /*
         * Большинство функций чтения возвращают число байт, посланных в буфер
         */
        return bytes_read;
}

/*
 * Вызывается, когда процесс записывает в файл устройства, например:
 *  echo "hi" > /dev/chardev-1
 */
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
        /*
         * Число байт, реально записанных в буфер msg
```

```
         */
        int bytes_write = 0;
      msg_wPtr = msg;
      if (len >= BUF_LEN) len = BUF_LEN -1;

        /*
         * Фактическая посылка данных в буфер
         */
        while (len && *buff) {

                get_user(*(msg_wPtr++), buff++);

                len--;
                bytes_write++;
        }
      *(msg_wPtr) = 0;
        /*
         * Большинство функций записи возвращают число байт, записанных в буфер
         */
        return bytes_write;
}
```

Исследование работы драйвера студент проводит *самостоятельно*, учитывая следующие его свойства:

- драйвер сообщает о начале и завершении свой работы, записывая соответствующее сообщение в системный журнал;
- после загрузки драйвера, создается файл устройства */dev/chardev-2*, что можно проверить кмандой: `ls -l /dev | grep chardev ;`
- после выгрузки драйвера, устройство автоматически удаляется;
- драйвер обеспечивает как чтение, так и запись в устройство, но только для пользователя *root*;
- чтобы сделать работу с устройством доступной для всех пользователей, нужно выполнить команду: sudo chmod 0666 /dev/chardev-2 .

Завершая работу по данной лабораторной работе студент должен:

- оформить все результаты исследований в личном отчете;
- сохранить копию отчета в области отличной от рабочей среды пользователя *upk*;
- корректно отключить тему обучения, а затем и личный flashUSB.

Успехов в учебе!

Список использованных источников

1 Резник В.Г. Операционные системы. Самостоятельная и индивидуальная работа студента. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 15 с.

2 Гордеев А.В. Операционные системы: учебное пособие для вузов. ─ СПб.: Питер, 2004. ─ 415с.

3 Таненбаум Э., Бос Х. Современные операционные системы. - СПб.: Питер, 2015. - 1120с.

4 Резник В.Г. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 38 с.

5 Резник В.Г. Современные операционные системы. Тема 1. Состояние и современные тенденции развития ОС. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 50 с.

Учебное издание

Резник Виталий Григорьевич

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебно-методическое пособие предназначено для изучения темы №2 по дисциплине «Современные операционные системы» для студентов уровня основной образовательной программы магистратура направления подготовки: 09.04.01 «Информатика и вычислительная техника».

Учебно-методическое пособие

Усл. печ. л. . Тираж . Заказ .
Томский государственный университет
систем управления и радиоэлектроники
634050, г. Томск, пр. Ленина, 40