Руководство по программированию модулей ядра Linux

1 The Linux Kernel Module Programming Guide **(Руководство по программированию модулей ядра Linux)**

Peter Jay Salzman, Michael Burian, Ori Pomerantz

2004-05-16 ver 2.6.0

Перевод: Андрей Киселёв (kis_an [at] linuxgazette [dot] ru), www.linuxcenter.ru

Руководство посвящено написанию модулей ядра для Linux 2.6. Рассматриваются такие вопросы, как взаимодействие с пользовательскими процессами через файлы устройств и файловую систему /proc, а также реализация новых системных вызовов. Текст богато проиллюстрирован примерами.

Оригинальная версия была опубликована на сайте проекта The Linux Documentation Project.

Содержание

Перечень примеров

Предисловие

1. Об авторах

Эта книга изначально была написана Ори Померанцем (Ori Pomerantz) для ядра Linux версии 2.2. К сожалению у Ори не хватает времени на продолжение работы над этой книгой. В конце концов ядро Linux продолжает быстро развиваться. Питер Зальцман (Peter Jay Salzman) взял на себя труд по адаптации документа для ядра версии 2.4. К сожалению и Питер не нашел достаточно свободного времени, чтобы продолжить работу над книгой и дополнить ее материалами, касающимися ядра версии 2.6. Таким образом, майкл Бариан (Michael Burian) взялся за адаптацию материала книги для ядра 2.6.

2. Нумерация версий и дополнительные примечания

Ядро Linux -- динамически развивающийся проект. И перед авторами книги всегда остро стоял вопрос -- удалять ли устаревшие сведения из книги или сохранять их, как историческую ценность. Майкл и я (Питер Зальцман) решили создать отдельную ветку документа для каждой стабильной серии ядер. Таким образом, LKMPG (Linux Kernel Module Programming Guide), имеющее версию 2.4.x относится к ядру 2.4.x, а LKMPG 2.6.x -- к ядру версии 2.6.x. Мы решили не сохранять устаревшие сведения в новых версиях документа. Желающие получить эту информацию должны обращаться к соответствующим версиям документа.

Исходный код и подаваемый материал не относятся к какой либо конкретной аппаратной архитектуре, но я ничего не могу гарантировать. Одно важное исключение -- "Обработка прерываний" (Глава 12), где весь обсуждаемый материал относится к архитектуре x86.

3. Благодарности

Ори Померанц выражает свою благодарность Yoav Weiss, за многочисленные предложения, обсуждение материала и внесение исправлений. Он так же хотел бы поблагодарить Фродо Лоойаарда из Нидердандов, Стефана Джадда из Новой Зеландии, Магнуса Альторпа из Швеции и Эммануэль Папиракис из Квебека, Канада.

Питер выражает свою благодарность Ори, за то что позволил ему принять участие в проекте LKMPG. А так же Джеффа Ньюмиллера, Ронду Франчес Бэйли (ныне Ронда Франчес Зальцман) и Марка Кима за науку и долготерпение. Он так же хотел бы поблагодарить Дэвида Портера, который оказал неоценимую помощь в переводе документа из формата LaTeX в формат docbook. Это была тяжелая и нудная работа, но ее необходимо было сделать.

Отдельное спасибо всем участникам проекта www.kernelnewbies.org, и особенно Марку Маклолину и Джону Левону, которые, я уверен, могли бы найти много более интересное занятие, нежели "зависать" на kernelnewbies.org и обучать новичков. Если это руководство научит вас чему либо, знайте -- в этом есть доля и их "вины"!

И Ори и я хотели бы сказать слова благодарности в адрес Ричарда М. Столлмана и Линаса Торвальдса не только за эту превосходную операционную систему, но и за то, что позволили исследовать ее исходный код, чтобы разобраться в том, как она работает.

Хочется выразить благодарность тем, кто внес свои исправления в документ или внес свои предложения по улучшению. Это Игнасио Мартин, Дэвид Портер, Дэниэл Паоло Скарпацца и Димо Велев.

Глава 1. Введение.

1.1. Что такое "Модуль Ядра"?

Итак, Вы хотите писать модули ядра. Вы знакомы с языком C и у вас есть опыт создания обычных программ, а теперь вы хотите забраться туда, где свершается великое таинство. Туда, где один ошибочный указатель может "снести" файловую систему или "подвесить" компьютер.

Так что же такое "модуль ядра"? Модуль -- это некий код, который может быть загружен или выгружен ядром по мере необходимости. Модули расширяют функциональные возможности ядра без необходимости перезагрузки системы. Например, одна из разновидностей модулей ядра, драйверы устройств, позволяют ядру взаимодействовать с аппаратурой компьютера. При отсутствии поддержки модулей нам пришлось бы писать монолитные ядра и добавлять новые возможности прямо в ядро. При этом, после добавления в ядро новых возможностей, пришлось бы перезагружать систему.

1.2. Как модули попадают в ядро?

Вы можете просмотреть список загруженных модулей командой **lsmod**, которая в свою очередь обращается за необходимыми сведениями к файлу `/proc/modules`.

Как же модули загружаются ядром? Когда ядро обнаруживает необходимость в тех или иных функциональных возможностях, еще не загруженных в память, то демон **kmod** [1] вызывает утилиту **modprobe**, передавая ей строку в виде:

- Название модуля, например `softdog` или `ppp`.
- Универсальный идентификатор, например `char-major-10-30`.

Если утилите **modprobe** передается *универсальный идентификатор*, то она сначала пытается отыскать имя соответствующего модуля в файле `/etc/modules.conf`, где каждому универсальному идентификатору поставлено в соответствие имя модуля, например:

```
alias char-major-10-30 softdog
```

Это соответствует утверждению: "Данному универсальному идентификатору соответствует файл модуля `softdog.ko`".

Затем **modprobe** отыскивает файл `/lib/modules/version/modules.dep`, чтобы проверить -- не нужно ли загружать еще какие-либо модули, от которых может зависеть заданный модуль. Этот файл создается командой **depmod -a** и описывает зависимости модулей. Например, модуль `msdos.ko` требует, чтобы предварительно был загружен модуль `fat.ko`. Если модуль **Б** экспортирует ряд имен (имена функций, переменных и т.п.), которые используются модулем **А**, то говорят, что "Модуль **А** зависит от модуля **Б**".

И наконец **modprobe** вызывает **insmod**, чтобы сначала загрузить необходимые, для удовлетворения зависимостей, модули, а затем и запрошенный модуль. Вызывая **insmod**, утилита **modprobe** указывает ей каталог, `/lib/modules/version/` [2] -- стандартный путь к модулям ядра. Утилита **insmod** ничего не "знает" о размещении модулей ядра, зато это "знает" утилита **modprobe**. Таким образом, если вам необходимо загрузить модуль `msdos`, то вам необходимо дать следующие команды:

```
insmod /lib/modules/2.6.0/kernel/fs/fat/fat.ko
insmod /lib/modules/2.6.0/kernel/fs/msdos/msdos.ko
```

или просто:

```
modprobe -a msdos
```

В большинстве дистрибутивов Linux, утилиты **modprobe, insmod, depmod** входят в состав пакета **modutils** или **mod-utils**.

Прежде чем закончить эту главу, я предлагаю вкратце ознакомиться с содержимым файла `/etc/modules.conf`:

```
### This file is automatically generated by modules-update
#
# Please do not edit this file directly. If you want to change or add
# anything please take a look at the files in /etc/modules.d and read
# the manpage for modules-update.
#
### modules-update: start processing /etc/modules.d/aliases
# Aliases to tell insmod/modprobe which modules to use
path[misc]=/lib/modules/2.6.?/local
keep
path[net]=~p/mymodules
options mydriver irq=10
alias eth0 eepro
```

Строки, начинающиеся с символа "#" являются комментариями. Пустые строки игнорируются.

Строка `path[misc]` сообщает **modprobe** о том, что модули ядра из категории misc следует искать в каталоге `/lib/modules/2.6.?/local`. Как видите, здесь вполне допустимы шаблонные символы.

Строка `path[net]` задает каталог размещения модулей категории net, однако, директива **keep**, стоящая выше, сообщает, что каталог `~p/mymodules` не замещает стандартный путь поиска модулей (как это происходит в случае с `path[misc]`), а лишь добавляется к нему.

Строка `alias` говорит о том, что если запрошена загрузка модуля по универсальному идентификатору `eth0`, то следует загружать модуль `eepro.ko`

Вы едва ли встретите в этом файле строки, подобные:

```
alias block-major-2 floppy
```

поскольку **modprobe** уже знает о существовании стандартных драйверов устройств, которые используются в большинстве систем.

1.2.1. Прежде, чем продолжить

Прежде, чем мы приступим к программированию, необходимо обсудить еще ряд моментов. Любая система имеет свои отличительные черты и каждый из нас имеет разный багаж знаний. Написать, скомпилировать и запустить свою первую программу "Hello World!" для многих может оказаться довольно сложной задачей. Однако, после преодоления этого начального препятствия, работа, как правило, продвигается без особых проблем.

2 1.2.1.1. Механизм контроля версий

Модули, скомпилированные с одним ядром, могут не загружаться другим ядром, если в ядре включен механизм проверки версий модулей. В большинстве дистрибутивов ядро собирается с такой поддержкой. Мы не собираемся обсуждать проблему контроля версий в этой книге, поэтому нам остается порекомендовать, в случае возникновения проблем, пересобрать ядро без поддержки механизма контроля версий.

3 1.2.1.2. Работа в XWindow

Мы настоятельно рекомендуем скачать и опробовать все примеры, обсуждаемые в книге. Кроме того, мы настаиваем на том, чтобы всю работу, связанную с редактированием исходных текстов, компиляцией и запуском модулей, вы выполняли из текстовой консоли. Поверьте нашему опыту, XWindow не подходит для выполнения подобных задач.

Модули не могут использовать функцию `printf()` для вывода не экран, но они могут регистрировать сообщения об ошибках, которые в конечном итоге попадают на экран, но только в текстовой консоли. Если же модуль загружается из окна терминала, например `xterm`, то эти сообщения будут попадать только в системный журнал и не будут выводиться на экран. Чтобы видеть выводимые сообщения на экране, работайте в текстовой консоли (***от переводчика:*** *при опробовании примеров из книги мне не удалось вывести ни одного сообщения на экран, так что ищите ваши сообщения в системном журнале, в моем случае это был файл* `/var/log/kern.log`).

4 1.2.1.3. Проблемы компиляции

Зачастую, дистрибутивостроители распространяют исходные тексты ядра, на которые уже наложены разные нестандартные заплаты. Это может породить определенные проблемы.

Не менее частый случай -- неполный набор заголовочных файлов ядра. Для сборки своих модулей вам потребуются многие заголовочные файлы ядра Linux. А закон Мэрфи гласит: "Отсутствовать будут как раз те файлы, в которых вы больше всего нуждаетесь".

Чтобы избежать этих двух проблем, мы рекомендуем собрать и установить наиболее свежее ядро. Скачать исходные тексты ядра вы сможете на любом из зеркал, распространяющих ядро Linux. За более подробной информацией обращайтесь к "Kernel HOWTO".

Как это ни покажется странным, но проблемы могут крыться и в компиляторе. По-умолчанию `gcc` может искать заголовочные файлы ядра совсем не там, куда вы их установили (как правило это каталог `/usr/src/`. Эта проблема легко преодолевается заданием ключа компиляции **-I**.

Глава 2. Hello World

2.1. "Hello, World" (часть 1): Простейший модуль ядра.

Когда первый пещерный программист высекал свою первую программу для каменного компьютера, это была программа, которая рисовала "Hello, World!" поперек изображения антилопы. Древнеримские учебники по программированию начинались с программы "Salut, Mundi". Я не знаю -- что случается с людьми, которые порывают с этой традицией, но думаю, что лучше этого и не знать. Мы начнем с серии программ "Hello world", на которых разберем различные базовые аспекты создания модулей ядра.

Ниже приводится исходный текст самого простого модуля ядра, какой только возможен. Пока только ознакомьтесь с его содержимым, а компиляцию и запуск модуля мы обсудим в следующем разделе.

Пример 2-1. hello-1.c

```
/*
 *  hello-1.c - Простейший модуль ядра.
 */
#include <linux/module.h>  /* Необходим для любого модуля ядра */
#include <linux/kernel.h>  /* Здесь находится определение KERN_ALERT */

int init_module(void)
{
        printk("<1>Hello world 1.\n");

  /*
   * Если вернуть ненулевое значение, то это
   * будет воспринято как признак ошибки,
   * возникшей в процессе работы init_module;
   * в результате модуль не будет загружен.
   */
        return 0;
}

void cleanup_module(void)
```

```
{
        printk(KERN_ALERT "Goodbye world 1.\n");
}
```

Любой модуль ядра должен иметь по меньшей мере хотя бы две функции: функцию инициализации модуля -- `init_module()`, которую вызывает **insmod** во время загрузки модуля, и функцию завершения работы модуля -- `cleanup_module()`, которую вызывает **rmmod**. Начиная с ядра, версии 2.3.13, требования к именованию начальной и конечной функций были сняты. Теперь вы можете давать им свои имена. Как это сделать будет описано в разделе Hello World (часть 2). Новый метод именования является более предпочтительным, однако многие по-прежнему продолжают использовать имена `init_module()` и `cleanup_module()`.

Обычно функция `init_module()` выполняет регистрацию обработчика какого-либо события или замещает какую-либо функцию в ядре своим кодом (который, как правило, выполнив некие специфические действия, вызывает оригинальную версию функции в ядре). Функция `cleanup_module()` является полной противоположностью, она производит "откат" изменений, сделаных функцией `init_module()`, что делает выгрузку модуля безопасной.

И наконец, любой модуль ядра должен подключать заголовочный файл `linux/module.h`. В нашем примере мы подключали еще один файл -- `linux/kernel.h`, но лишь для того, чтобы получить доступ к определению `KERN_ALERT`, которое более подробно будет обсуждаться в следующем разделе

2.1.1. Знакомство с `printk()`

Несмотря на столь красноречивое название, функция `printk()` вовсе не предназначена для вывода информации на экран, даже не смотря на то, что мы использовали ее в своем примере именно для этой цели! Основное назначение этой функции -- дать ядру механизм регистрации событий и предупреждений. Поэтому, каждый вызов `printk()` сопровождается указанием приоритета, в нашем примере это `<1>` и `KERN_ALERT`. Всего в ядре определено 8 различных уровней приоритета для функции `printk()` и каждый из них имеет свое макроопределение, таким образом нет необходимости писать числа, лишенные смысла (имена уровней приоритета и их числовые значения вы найдете в файле `linux/kernel.h`). Если уровень приоритета не указывается, то по-умолчанию он принимается равным `DEFAULT_MESSAGE_LOGLEVEL`.

Найдите время и просмотрите содержимое этого файла. Здесь вы найдете краткое описание значения каждого из уровней. На практике считается дурным тоном указание уровней приоритета числовым значением, например так: `<4>`. Для этих целей лучше пользоваться именами макроопределений, например: `KERN_WARNING`.

Если задан уровень ниже, чем `int console_loglevel`, то сообщение выводится на экран. Если запущены и **syslog**, и **klogd**, то сообщение попадет также и в системный журнал `/var/log/messages`, при этом оно может быть выведено на экран, а может и не выводиться. Мы использовали достаточно высокий уровень приоритета `KERN_ALERT` для того, чтобы гарантировать вывод сообщения на экран функцией `printk()`. Когда вы вплотную займетесь созданием модулей ядра, вы будете использовать уровни приоритета наиболее подходящие под конкретную ситуацию.

2.2. Сборка модулей ядра

Чтобы модуль был работоспособен, при компиляции необходимо передать **gcc** ряд опций. Кроме того, необходимо чтобы модули компилировались с предварительно определенными символами. Ранние версии ядра полностью полагались, в этом вопросе, на программиста и ему приходилось явно указывать требуемые определения в `Makefile`-ах. Несмотря на иерархическую организацию, в `Makefile`-ах, на вложенных уровнях, накапливалось такое огромное количество параметров настройки, что управление и сопровождение этих настроек стало довольно трудоемким делом. К счастью появился **kbuild**, в результате процесс сборки внешних загружаемых модулей теперь полностью интегрирован в механизм сборки ядра. Дополнительные сведения по сборке модулей, которые не являются частью официального ядра (как в нашем случае), вы найдете в файле `linux/Documentation/kbuild/modules.txt`.

А теперь попробуем собрать наш с вами модуль hello-1.c . Соответствующий `Makefile` содержит всего одну строку:

Пример 2-2. Makefile для модуля ядра

```
obj-m += hello-1.o
```

Для того, чтобы запустить процесс сборки модуля, дайте команду **make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules** (*от переводчика: если у вас в каталоге `/usr/src` присутствует символическая ссылка `linux` на каталог с исходными текстами ядра, то команда сборки может быть несколько упрощена:* ***make -C /usr/src/linux SUBDIRS=$PWD modules***). На экран должно быть выведено нечто подобное:

```
[root@pcsenonsrv test_module]# make -C /
/usr/src/linux-`uname -r` SUBDIRS=$PWD modules
make: Entering directory `/usr/src/linux-2.6.x
  CC [M]  /root/test_module/hello-1.o
  Building modules, stage 2.
  MODPOST
  CC      /root/test_module/hello-1.mod.o
  LD [M]  /root/test_module/hello-1.ko
make: Leaving directory `/usr/src/linux-2.6.x
```

Обратите внимание: в ядрах версии 2.6 введено новое соглашение по именованию объектных файлов модулей. Теперь, они имеют расширение `.ko` (взамен прежнего `.o`), что отличает их от обычных объектных файлов. Дополнительную информацию по оформлению `Makefile`-ов модулей вы найдете в `linux/Documentation/kbuild/makefiles.txt`. Обязательно прочтите этот документ прежде, чем начнете углубляться в изучение `Makefile`-ов.

Итак, настал торжественный момент -- теперь можно загрузить свежесобранный модуль! Дайте команду **insmod ./hello-1.ko** (появляющиеся сообщения о "загрязнении" ядра вы сейчас можете просто игнорировать, вскоре мы обсудим эту проблему).

Любой загруженный модуль ядра заносится в список `/proc/modules`, так что дружно идем туда и смотрим содержимое этого файла. как вы можете убедиться, наш модуль стал частью ядра. С чем вас и поздравляем, теперь вы стали одним из авторов кода ядра! Вдоволь насладившись ощущением новизны, выгрузите модуль командой **rmmod hello-1** и загляните в файл `/var/log/messages`, здесь вы увидите сообщения, которые сгенерировал ваш модуль. (*от переводчика: в моем случае на экран консоли сообщения не выводились, зато они появились в файле* `/var/log/kern.log`).

А теперь небольшое упражнение: Измените содержимое файла `hello-1.c` так, чтобы функция `init_module()` возвращала бы какое либо ненулевое значение и проверьте -- что получится?

2.3. Hello World (часть 2)

Как мы уже упоминали, начиная с ядра, версии 2.3.13, требования к именованию начальной и конечной функций модуля были сняты. Достигается это с помощью макроопределений `module_init()` и `module_exit()`. Они определены в файле `linux/init.h`. Единственное замечание: начальная и конечная функции должны быть определены выше строк, в которых вызываются эти макросы, в противном случае вы получите ошибку времени компиляции. Ниже приводится пример использования этих макроопределений:

Пример 2-3. hello-2.c

```
/*
 *  hello-2.c - Демонстрация использования
 *  макроопределений module_init() и module_exit().
 */
#include <linux/module.h>  /* Необходим для любого модуля ядра */
#include <linux/kernel.h>  /* Здесь находится определение KERN_ALERT */
#include <linux/init.h>    /* Здесь находятся определения макросов */

static int __init hello_2_init(void)
{
        printk(KERN_ALERT "Hello, world 2\n");
        return 0;
}

static void __exit hello_2_exit(void)
{
        printk(KERN_ALERT "Goodbye, world 2\n");
}

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

Теперь мы имеем в своем багаже два настоящих модуля ядра. Добавить сборку второго модуля очень просто:

Пример 2-4. Makefile для сборки обоих модулей

```
obj-m += hello-1.o
obj-m += hello-2.o
```

Теперь загляните в файл `linux/drivers/char/Makefile`. Он может рассматриваться как пример полноценного `Makefile` модуля ядра. Здесь видно, что ряд модулей жестко "зашиты" в ядро (obj-y), но нигде нет строки obj-m. Почему? Знакомые с языком сценариев командной оболочки легко найдут ответ. Все записи вида obj-$(CONFIG_FOO) будут заменены на obj-y или obj-m, в зависимости от значения переменных CONFIG_FOO. Эти переменные вы сможете найти в файле `.config`, который был создан во время конфигурирования ядра с помощью **make menuconfig** или что-то вроде этого.

2.4. Hello World (часть 3): Макроопределения __init и __exit

Это демонстрация особенностей ядра, появившихся, начиная с версии 2.2. Обратите внимание на то, как изменились определения функций инициализации и завершения работы модуля. Макроопределение `__init` вынуждает ядро, после выполнения инициализации модуля, освободить память, занимаемую функцией, правда относится это только к встроенным модулям и не имеет никакого эффекта для загружаемых модулей. Если вы мысленно представите себе весь процесс инициализации встроенного модуля, то все встанет на свои места.

То же относится и к макросу `__initdata`, но только для переменных.

Макроопределение `__exit` вынуждает ядро освободить память, занимаемую функцией, но только для встроенных модулей, на загружаемые модули это макроопределение не оказывает эффекта. Опять же, если вы представите себе -- когда вызывается функция завершения работы модуля, то станет понятно, что для встроенных модулей она не нужна, в то время как для загружаемых модулей -- просто необходима.

Оба этих макроса определены в файле `linux/init.h` и отвечают за освобождение неиспользуемой памяти в ядре. Вам наверняка приходилось видеть на экране, во аремя загрузки, сообщение примерно такого содержания: `Freeing unused kernel memory: 236k freed`. Это как раз и есть результат работы данных макроопределений.

Пример 2-5. hello-3.c

```
/*
 *  hello-3.c - Использование макроопределений __init, __initdata и __exit.
 */
#include <linux/module.h>  /* Необходим для любого модуля ядра */
#include <linux/kernel.h>  /* Здесь находится определение KERN_ALERT */
#include <linux/init.h>    /* Здесь находятся определения макросов */

static int hello3_data __initdata = 3;

static int __init hello_3_init(void)
{
        printk(KERN_ALERT "Hello, world %d\n", hello3_data);
        return 0;
}

static void __exit hello_3_exit(void)
{
        printk(KERN_ALERT "Goodbye, world 3\n");
}

module_init(hello_3_init);
module_exit(hello_3_exit);
```

2.5. Hello World (часть 4): Вопросы лицензирования и документирования модулей

Если у вас установлено ядро 2.4 или более позднее, то наверняка, во время запуска примеров модулей, вам пришлось столкнуться с сообщениями вида:

```
# insmod hello-3.o
Warning: loading hello-3.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted
  for information about tainted modules
Hello, world 3
Module hello-3 loaded, with warnings
```

В ядра версии 2.4 и выше был добавлен механизм контроля лицензий, чтобы иметь возможность предупреждать пользователя об использовании проприетарного (не свободного) кода. Задать условия лицензирования модуля можно с помощью макроопределения `MODULE_LICENSE()`. Ниже приводится выдержка из файла `linux/module.h` (*от переводчика: я взял на себя смелость перевести текст комментариев на русский язык*):

```
/*
 * В настоящее время, для обозначения свободных лицензий, приняты следующие
 * идентификаторы
 *
 *      "GPL"                           [GNU Public License v2 или выше]
 *      "GPL v2"                        [GNU Public License v2]
 *      "GPL and additional rights"     [GNU Public License v2
 *                                      с дополнительными правами]
 *      "Dual BSD/GPL"                  [GNU Public License v2
 *                                      или BSD license]
 *      "Dual MPL/GPL"                  [GNU Public License v2
 *                                      или Mozilla license]
 *
 * Кроме того, дополнительно имеются следующие идентификаторы
 *
 *      "Proprietary"                   [проприетарный, не свободный продукт]
 *
 * Здесь присутствуют компоненты, подразумевающие двойное лицензирование,
 * однако, по отношению к Linux они приобретают значение GPL, как наиболее
 * уместное, так что это не является проблемой.
 * Подобно тому, как LGPL связана с GPL
 *
 * На это есть несколько причин
 * 1. modinfo может показать сведения о лицензировании
 *    для тех пользователей, которые желают, чтобы их
 *    набор программных компонент был свободным
 * 2. Сообщество может игнорировать отчеты об ошибках
 *    (bug reports), относящиеся к проприетарным модулям
 * 3. Поставщики программных продуктов могут поступать
 *    аналогичным образом, основываясь на своих
 *    собственных правилах
 */
```

Точно так же, для описания модуля может использоваться макрос `MODULE_DESCRIPTION()`, для установления авторства -- `MODULE_AUTHOR()`, а для описания типов устройств, поддерживаемых модулем -- `MODULE_SUPPORTED_DEVICE()`.

Все эти макроопределения описаны в файле `linux/module.h`. Они не используются ядром и служат лишь для описания модуля, которое может быть просмотрено с помощью **objdump**. Попробуйте с помощью утилиты **grep** посмотреть, как авторы модулей используют эти макросы (в каталоге `linux/drivers`).

Пример 2-6. hello-4.c

```
/*
 *  hello-4.c - Демонстрация описания модуля.
 */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#define DRIVER_AUTHOR "Peter Jay Salzman <p@dirac.org>"
#define DRIVER_DESC   "A sample driver"

static int __init init_hello_4(void)
{
        printk(KERN_ALERT "Hello, world 4\n");
        return 0;
}

static void __exit cleanup_hello_4(void)
{
        printk(KERN_ALERT "Goodbye, world 4\n");
}

module_init(init_hello_4);
module_exit(cleanup_hello_4);

/*
 *  Вы можете передавать в макросы строки, как это показано ниже:
```

```
 */

/*
 * Запретить вывод предупреждения о "загрязнении" ядра,
 * объявив код под GPL.
 */
MODULE_LICENSE("GPL");

/*
 * или определения:
 */
MODULE_AUTHOR(DRIVER_AUTHOR);    /* Автор модуля */
MODULE_DESCRIPTION(DRIVER_DESC); /* Назначение модуля */

/*
 *  Этот модуль использует устройство /dev/testdevice.
 *  В будущих версиях ядра
 *  макрос MODULE_SUPPORTED_DEVICE может быть использован
 *  для автоматической настройки модуля, но пока
 *  он служит исключительно в описательных целях.
 */
MODULE_SUPPORTED_DEVICE("testdevice");
```

2.6. Передача модулю параметров командной строки

Имеется возможность передачи модулю дополнительных параметров командной строки, но делается это не с помощью `argc/argv`.

Для начала вам нужно объявить глобальные переменные, в которые будут записаны входные параметры, а затем вставить макрос `MODULE_PARAM()`, для запуска механизма приема внешних аргументов. Значения параметров могут быть переданы модулю с помощью команд insmod или modprobe. Например: **insmod mymodule.ko myvariable=5**. Для большей ясности, объявления переменных и вызовы макроопределений следует размещать в начале модуля. Пример кода прояснит мое, по общему признанию, довольно неудачное объяснение.

Макрос `MODULE_PARAM()` принимает 2 аргумента: имя переменной и ее тип. Поддерживаются следующие типы переменных

- "b" -- byte (байт);
- "h" -- short int (короткое целое);
- "i" -- integer (целое, как со знаком, так и без знака);
- "l" -- long int (длинное целое, как со знаком, так и без знака);
- "s" -- string (строка, должна объявляться как char*).

Для переменных типа `char *`, **insmod** будет сама выделять необходимую память. Вы всегда должны инициализировать переменные значениями по-умолчанию, не забывайте -- это код ядра, здесь лучше лишний раз перестраховаться. Например:

```
int myint = 3;
char *mystr;

MODULE_PARAM(myint, "i");
MODULE_PARAM(mystr, "s");
```

Параметры-массивы так же допустимы. Целое число, предшествующее символу типа аргумента, обозначает максимальный размер массива. Два числа, разделенные дефисом -- минимальное и максимальное количество значений. Например, массив целых, который должен иметь не менее 2-х и не более 4-х значений, может быть объявлен так:

```
int myintArray[4];
MODULE_PARAM(myintArray, "2-4i");
```

Желательно, чтобы все входные параметры модуля имели значения по-умолчанию, например адреса портов ввода-вывода. Модуль может выполнять проверку переменных на значения по-умолчанию и если такая проверка дает положительный результат, то переходить к автоматическому конфигурированию (вопрос автонастройки будет обсуждаться ниже).

И, наконец, еще одно макроопределение -- `MODULE_PARAM_DESC()`. Оно используется для описания входных аргументов модуля. Принимает два параметра: имя переменной и строку описания, в свободной форме.

Пример 2-7. hello-5.c

```
/*
 *  hello-5.c - Пример передачи модулю аргументов командной строки.
 */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";

/*
 * module_param(foo, int, 0000)
 * Первый параметр -- имя переменной,
 * Второй -- тип,
 * Последний -- биты прав доступа
 * для того, чтобы выставить в sysfs
 * (если ненулевое значение) на более поздней стадии.
 */

module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");

static int __init hello_5_init(void)
{
        printk(KERN_ALERT "Hello, world 5\n=============\n");
        printk(KERN_ALERT "myshort is a short integer: %hd\n", myshort);
        printk(KERN_ALERT "myint is an integer: %d\n", myint);
        printk(KERN_ALERT "mylong is a long integer: %ld\n", mylong);
        printk(KERN_ALERT "mystring is a string: %s\n", mystring);
        return 0;
}

static void __exit hello_5_exit(void)
{
        printk(KERN_ALERT "Goodbye, world 5\n");
}

module_init(hello_5_init);
module_exit(hello_5_exit);
```

Давайте немножко поэкспериментируем с этим модулем:

```
satan# insmod hello-5.o mystring="bebop" myshort=255
myshort is a short integer: 255
myint is an integer: 420
mylong is a long integer: 9999
mystring is a string: bebop

satan# rmmod hello-5
Goodbye, world 5

satan# insmod hello-5.o
    mystring="supercalifragilisticexpialidocious" myint=100
myshort is a short integer: 1
myint is an integer: 100
mylong is a long integer: 9999
mystring is a string: supercalifragilisticexpialidocious

satan# rmmod hello-5
Goodbye, world 5

satan# insmod hello-5.o mylong=hello
hello-5.o: `hello' invalid for parameter mylong
```

2.7. Модули, состоящие из нескольких файлов

Иногда возникает необходимость разместить исходные тексты модуля в нескольких файлах. В этом случае **kbuild** опять возьмет на себя всю "грязную" работу, а `Makefile` поможет сохранить наши руки чистыми, а голову светлой! Ниже приводится пример модуля, состоящего из двух файлов:

Пример 2-8. start.c

```
/*
 *  start.c - Пример модуля, исходный
 *  текст которого размещен в нескольких файлах
 */

#include <linux/kernel.h>       /* Все-таки мы пишем код ядра! */
#include <linux/module.h>       /* Необходим для любого модуля ядра */
```

```
int init_module(void)
{
        printk("Hello, world - this is the kernel speaking\n");
        return 0;
}
```

Пример 2-9. stop.c

```
/*
 *  stop.c - Пример модуля, исходный текст
 *  которого размещен в нескольких файлах
 */

#include <linux/kernel.h>       /* Все-таки мы пишем код ядра! */
#include <linux/module.h>       /* Необходим для любого модуля ядра */

void cleanup_module()
{
        printk("<1>Short is the life of a kernel module\n");
}
```

Пример 2-10. Makefile для сборки всех модулей

```
obj-m += hello-1.o
obj-m += hello-2.o
obj-m += hello-3.o
obj-m += hello-4.o
obj-m += hello-5.o
obj-m += startstop.o
startstop-objs := start.o stop.o
```

2.8. Сборка модулей под существующее ядро

Мы уже рекомендовали вам пересобрать свое ядро, включив некоторые полезные для отладки опции, например такие, как (MODULE_FORCE_UNLOAD) -- когда эта опция включена, то вы имеете возможность принудительной выгрузки модуля (посредством команды **rmmod -f module_name**), даже если ядро "считает" ваши действия небезопасными. Эта опция поможет вам сэкономить время на перезагрузках системы, в процессе отладки модуля.

Как бы то ни было, но ситуация может сложиться так, что вам потребуется загрузить модуль в ранее откомпилированное ядро, например, на другой системе, или в случае, когда вы не можете пересобрать ядро по каким либо соображениям. Если вы не предполагаете возникновение таких ситуаций, то можете просто пропустить эту часть главы.

Если вы лишь установили дерево с исходными текстами ядра и использовали их для сборки своего модуля, то в большинстве случаев, при попытке загрузить его в работающее ядро, вы получите следующее сообщение об ошибке:

```
insmod: error inserting 'your_module_name.ko': -1 Invalid module format
```

Более подробная информация будет помещена в файл `/var/log/messages`:

```
Jun  4 22:07:54 localhost kernel:
your_module_name: version magic '2.6.5-1.358custom 686
REGPARM 4KSTACKS gcc-3.3' should be
'2.6.5-1.358 686 REGPARM 4KSTACKS gcc-3.3'
```

Другими словами -- ваше ядро отказывается "принимать" ваш модуль из-за несоответствия версий (точнее -- из-за несоответствия сигнатур версий). Сигнатура версии сохраняется в объектном файле в виде статической строки, начинающейся со слова `vermagic:`. Эта строка вставляется во время компоновки модуля с файлом `init/vermagic.o`. Просмотреть сигнатуру версии (так же как и некоторые дополнительные сведения) можно посредством команды **modinfo module.ko**:

```
[root@pcsenonsrv 02-HelloWorld]# modinfo hello-4.ko
license:        GPL
author:         Peter Jay Salzman <p@dirac.org>
description:    A sample driver
vermagic:       2.6.5-1.358 686 REGPARM 4KSTACKS gcc-3.3
depends:
```

Для преодоления этого препятствия можно воспользоваться ключом **--force-vermagic** (команды **modprobe**, *прим. перев.*), но это решение потенциально опасно и совершенно неприменимо при распространении готовых модулей. Следовательно, вам придется пересобрать модуль в окружении идентичном тому, в котором было собрано целевое ядро. Вопрос: "Как это сделать?" и является темой для дальнейшего обсуждения в данной главе.

Прежде всего вам необходимо установить дерево с исходными текстами ядра той же версии, что и целевое ядро. Найдите файл конфигурации целевого ядра, как правило он располагается в каталоге `/boot`, под именем, что-то вроде `config-2.6.x`. Просто скопируйте его в каталог с исходными текстами ядра на своей машине.

Вернемся к сообщению об ошибке, которое было приведено выше, и еще раз внимательно прочитаем его. Как видите, версия ядра практически та же самая, но даже небольшого отличия хватило, чтобы ядро отказалось загружать модуль. Все различие заключается лишь в наличии слова `custom` в сигнатуре

версии модуля. Теперь откройте `Makefile` ядра и удостоверьтесь, что информация о версии в точности соответствует целевому ядру. Например, в данном конкретном случае `Makefile` должен содержать строки:

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 5
EXTRAVERSION = -1.358custom
...
```

Теперь запустите **make**, чтобы обновить информацию о версии:

```
[root@pcsenonsrv linux-2.6.x]# make
CHK     include/linux/version.h
UPD     include/linux/version.h
SYMLINK include/asm -> include/asm-i386
SPLIT   include/linux/autoconf.h -> include/config/*
HOSTCC  scripts/basic/fixdep
HOSTCC  scripts/basic/split-include
HOSTCC  scripts/basic/docproc
HOSTCC  scripts/conmakehash
HOSTCC  scripts/kallsyms
CC      scripts/empty.o
...
```

Если вы не желаете полностью пересобирать ядро, то можете прервать процесс сборки (CTRL_C) сразу же после появления строки, начинающейся со слова `SPLIT`, поскольку в этот момент все необходимые файлы уже будут готовы. Перейдем в каталог с исходными текстами модуля и скомпилируем его. Теперь сигнатура версии модуля будет в точности соответствовать версии целевого ядра и будет загружено им без каких либо проблем.

Глава 3. Дополнительные сведения

3.1. Модули ядра и прикладные программы

Работа программы обычно начинается с исполнения функции `main()`. После выполнения всей последовательность команд программа завершает свою работу. Модули исполняются иначе. Они всегда начинают работу с исполнения функции `init_module`, или с функции, которую вы определили через вызов `module_init`. Это функция запуска модуля, которая подготавливает его для последующих вызовов. После завершения исполнения функции `init_module` модуль больше ничего не делает, он просто "сидит и ждет", когда ядро обратится к нему для выполнения специфических действий.

Вторая точка входа в модуль -- cleanup_module, вызывается непосредственно перед его выгрузкой. Она производит "откат" изменений, выполненных функцией `init_module()` и, как бы говорит ядру: "Я ухожу! Больше не проси меня ни о чем!".

Любой модуль обязательно должен иметь функцию инициализации и функцию завершения. Так как существует более чем один способ определить функции инициализации и завершения, я буду стараться использовать термины "начальная" и "конечная" функции, если я собьюсь и укажу названия `init_module` и `cleanup_module`, то думаю, что вы поймете меня правильно.

3.2. Функции, которые доступны из модулей

Как программист, вы знаете, что приложение может вызывать функции, которые не определены в самой программе. На стадии связывания (линковки) разрешаются все внешние ссылки, уходящие во внешние библиотеки. Функция printf -- одна из таких функций, которая определена в библиотеке libc.

Модули ядра в этом плане сильно отличаются от прикладных программ. В примере "Hello World" мы использовали функцию `printk()`, но не подключали стандартную библиотеку ввода-вывода. Модули так же проходят стадию связывания, но только с ядром, и могут вызывать только те функции, которые экспортируются ядром. Разрешение ссылок на внешние символы производится утилитой **insmod**. Если у вас есть желание взглянуть на список имен, экспортируемых ядром, загляните в файл `/proc/kallsyms`.

Здесь я хочу заострить ваше внимание на различиях между библиотечными функциями и системными вызовами. Библиотечные функции -- это верхний уровень, который работает в пространстве пользователя и обеспечивает более удобный интерфейс к функциям, которые выполняют основную работу -- системным вызовам. Системные вызовы работают в привилегированном режиме от имени пользователя и предоставляются самим ядром. Библиотечная функция `printf()` на первый взгляд выглядит как основная функция вывода, но все, что она фактически делает -- это формирует строку, в соответствии с заданным форматом, и передает ее низкоуровневому системному вызову `write()`, который и выводит строку на устройство стандартного вывода.

Как в этом можно убедиться? Да очень просто! Скомпилируйте следующую программу:

```
#include <stdio.h>
int main(void)
{
  printf("hello");
  return 0;
}
```

с помощью команды **gcc -Wall -o hello hello.c** и запустите ее командой **strace hello**. Впечатляет? Каждая строка, выводимая на экран, соответствует системному вызову. **strace** -- незаменимый инструмент для того, чтобы выяснить -- куда программа, пытается обратиться, включая такие сведения, как имена системных вызовов, передаваемые им аргументы и возвращаемые значения. Здесь вы должны увидеть строку, которая выглядит примерно так:

`write(1, "hello", 5hello)`. Это и есть то, что мы ищем. Т.е. скрытая от нас сторона вызова функции `printf()`. Возможно вы не знакомы с вызовом `write()`, поскольку большинство программистов предпочитает пользоваться стандартными библиотечными функциями (такими как `fopen(), fputs(), fclose()`). Если это так, тогда загляните в **man 2 write**. Второй раздел справочного руководства содержит описания системных вызовов (таких как `kill(), read()` и т.п.). В третьем разделе описываются библиотечные вызовы (такие как `cosh(), random()` и пр.).

Вы можете даже написать модули, которые подменяют системные вызовы ядра, вскоре мы продемонстрируем это. Взломщики довольно часто используют эту возможность для создания "черного хода" в систему или "троянов", но вы можете использовать ее в менее вредоносных целях, например заставить ядро выводить строку "Tee hee, that tickles!" ("Хи-хи, щекотно!") каждый раз, когда кто нибудь пробует удалить файл.

3.3. Пространство пользователя и пространство ядра

За доступ к ресурсам системы отвечает ядро, будь то видеоплата, жесткий диск или даже память. Программы часто конкурируют между собой за доступ к тем или иным ресурсам. Например, при подготовке этого документа, я сохраняю файл с текстом на жесткий диск, тут же стартует **updatedb**, чтобы обновить локальную базу данных. В результате мой **vim** и **updatedb** начинают конкурировать за обладание жестким диском. Ядро должно обслужить конкурирующие запросы, и "выстроить" их в порядке очередности. К тому же сам центральный процессор может работать в различных режимах. Каждый из режимов имеет свою степень "свободы" действий. Микропроцессор Intel 80386 имеет четыре таких режима, которые часто называют "кольцами". Unix использует только два из них: наивысший (нулевое кольцо, известное так же под названием *привилегированный режим*) и низший (*пользовательский режим*).

Вернемся к обсуждению библиотечных функций и системных вызовов. Как правило, программа обращается к библиотечным функциям, находясь в *пользовательском режиме*. Затем библиотечные функции обращаются к системным вызовам. Системные вызовы выступают от имени библиотечных функций, но работают в *привилегированном режиме*, так как они являются непосредственной частью ядра. Как только системный вызов завершает свою работу, он возвращает управление библиотечной функции и происходит обратный переход в *пользовательский режим*.

Обычно, о режимах исполнения, мы говорим как о *пространстве ядра* и *пространстве пользователя*. Эти два понятия охватывают не только два режима исполнения, но так же и то, что каждый из режимов имеет свое собственное отображение памяти -- свое собственное адресное пространство.

Unix производит переключение из *пространства пользователя* в *пространство ядра* всякий раз, когда приложение делает системный вызов или приостанавливается аппаратным прерыванием. Код ядра, исполняющий системный вызов, работает в контексте процесса -- от имени вызвавшего процесса и имеет доступ к данным в адресном пространстве процесса. Код, который обрабатывает прерывание, наоборот, являясь асинхронным по своей природе, не относится ни к одному из процессов.

Основное назначение модулей -- расширение функциональности ядра. Код модуля исполняется в *пространстве ядра*. Обычно модуль реализует обе, рассмотренные выше задачи -- одни функции выполняются как часть системных вызовов, другие -- производят обработку прерываний.

3.4. Пространство имен

Когда вы пишете небольшую программку на C, вы именуете свои функции и переменные так, как вам это удобно. С другой стороны, когда вы разрабатываете некую программную единицу, входящую в состав большого программного пакета, любые глобальные переменные, которые вы вводите, становятся частью всего набора глобальных переменных пакета. В этой ситуации могут возникнуть конфликты имен. Внедрение большого количества имен функций и переменных, с глобальной областью видимости, значение которых не интутивно и трудноразличимо, приводит к "загрязнению" пространства имен. Программист, который работает с такими приложениями, тратит огромное количество умственных сил на то, чтобы запомнить "зарезервированные" имена и придумать свои уникальные названия.

Модули ядра компонуются с огромным программным пакетом -- ядром, поэтому проблема "загрязнения" пространства имен становится достаточно острой. Коллизии имен могут породить трудноуловимые ошибки, начиная от того, что модуль просто отказывается загружаться, и заканчивая весьма причудливыми сообщениями. Лучший способ избежать "загрязнения" пространства имен -- это объявлять все имена как `static` и использовать префиксы для придания уникальности именам с глобальной областью видимости. По соглашению об именовании, желательно, в качестве префиксов, использовать символы нижнего регистра. Если вы не можете какие-то имена объявить как `static`, то разрешить проблему можно посредством создания `symbol table` и регистрации ее в ядре. Эту тему мы обсудим ниже.

Файл `/proc/kallsyms` содержит все имена в ядре, с глобальной областью видимости, которые доступны для ваших модулей.

3.5. Адресное пространство

Управление памятью - очень сложная тема, она достаточно полно освещается в книге "Understanding The Linux Kernel", выпущенной издательством O'Reilly. Мы не собираемся делать из вас экспертов в области управления памятью, но вам действительно необходимо знать некоторые факты.

Если вы никогда не задумывались над тем, что означает слово segfault, то для вас скорее всего окажется сюрпризом тот факт, что указатели фактически не указывают на какой-то реальный участок физической памяти. В любом случе, эти адреса не являются реальными. Когда запускается процесс, ядро выделяет под него кусок физической памяти и передает его процессу. Эта память используется для размещения исполняемого кода, стека, переменных, динамической "кучи" и других вещей, о чем наверняка знают компьютерные гении. [3] Эта память начинается с логического адреса #0 и простирается до того адреса, который необходим. Поскольку области памяти, выделенные для разных процессов, не пересекаются, то каждый из процессов, обратившись к ячейке памяти с адресом, скажем `0xbffff978`, получит данные из различных областей физической памяти! В даннм случае число `0xbffff978` можно рассматривать как смещение относительно начала области памяти, выделенной процессу. Как правило программы, подобные нашей "Hello World", не могут обратиться к памяти, занимаемой другим процессом, хотя существуют обходные пути, позволяющие добиться этого, но оставим пока эту тему для более позднего обсуждения.

Ядро тоже имеет свое собственное адресное пространство. Поскольку модуль по сути является частью ядра, то он так же работает в адресном пространстве ядра. Если ошибка `segmentation fault`, возникающая в приложении может быть отслежена и устранена без особых проблем, то в модуле подобная ошибка может стать фатальной для всей системы. Из-за незначительной ошибки в модуле вы рискуете "затоптать" ядро. Результат может быть самым плачевным. Поэтому будьте предельно внимательны!

Хотелось бы заметить, что это справедливо для любой операционной системы, которая построена на монолитном ядре. [4] Есть операционные системы, в основе которых лежит микроядро. В таких ОС каждый модуль получает свое адресное пространство. Примерами могут служить GNU Hurd и QNX Neutrino.

3.6. Драйверы устройств

Драйверы устройств являются одной из разновидностей модулей ядра. Они играют особую роль. Это настоящие "черные ящики", которые полностью скрывают детали, касающиеся работы устройства, и предоставляют четкий программный интерфейс для работы с аппаратурой. В Unix каждое аппаратное устройство представлено псевдофайлом (файлом устройства) в каталоге `/dev`. Этот файл обеспечивает средства взаимодействия с аппаратурой. Так, например, драйвер звуковой платы `es1370.ko` связывает файл устройства `/dev/sound` со звуковой платой Ensoniq IS1370. Пользовательское приложение, например **mp3blaster** может использовать для своей работы `/dev/sound`, ничего не подозревая о типе установленной звуковой платы.

3.6.1. Старший и младший номер устройства

Давайте взглянем на некоторые файлы устройств. Ниже перечислены те из них, которые представляют первые три раздела на первичном жестком диске:

```
# ls -l /dev/hda[1-3]
brw-rw----  1 root  disk  3, 1 Jul  5  2000 /dev/hda1
brw-rw----  1 root  disk  3, 2 Jul  5  2000 /dev/hda2
brw-rw----  1 root  disk  3, 3 Jul  5  2000 /dev/hda3
```

Обратили внимание на столбец с числами, разделенными запятой? Первое число называют "Старшим номером" устройства. Второе -- "Младшим номером". Старший номер говорит о том, какой драйвер используется для обслуживания аппаратного обеспечения. Каждый драйвер имеет свой уникальный старший номер. Все файлы устройств с одинаковым старшим номером управляются одним и тем же драйвером. Все из выше перечисленных файлов устройств имеют старший номер, равный 3, потому что все они управляются одним и тем же драйвером.

Младший номер используется драйвером, для различения аппаратных средств, которыми он управляет. Возвращаясь к примеру выше, заметим, что хотя все три устройства обслуживаются одним и тем же драйвером, тем не менее каждое из них имеет уникальный младший номер, поэтому драйвер "видит" их как различные аппаратные устройства.

Устройства подразделяются на две большие группы -- *блочные* и *символьные*. Основное различие блочных и символьных устройств состоит в том, что обмен данными с блочным устройством производится порциями байт -- блоками. Они имеют внутренний буфер, благодаря чему повышается скорость обмена. В большинстве Unix-систем размер одного блока равен 1 килобайту или другому числу, являющемуся степенью числа 2. Символьные же устройства -- это лишь каналы передачи информации, по которым данные следуют последовательно, байт за байтом. Большинство устройств относятся к классу символьных, поскольку они не ограничены размером блока и не нуждаются в буферизации. Если первый символ в списке, полученном командой **ls -l /dev**, 'b', тогда это блочное устройство, если 'c', тогда -- символьное. Устройства, которые были приведены в примере выше -- блочные. Ниже приводится список некоторых символьных устройств (последовательные порты):

```
crw-rw----  1 root  dial 4, 64 Feb 18 23:34 /dev/ttyS0
crw-r-----  1 root  dial 4, 65 Nov 17 10:26 /dev/ttyS1
crw-rw----  1 root  dial 4, 66 Jul  5  2000 /dev/ttyS2
crw-rw----  1 root  dial 4, 67 Jul  5  2000 /dev/ttyS3
```

Если вам интересно узнать, как назначаются старшие номера устройств, загляните в файл `/usr/src/linux/documentation/devices.txt` .

Все файлы устройств создаются в процессе установки системы с помощью утилиты **mknod**. Чтобы создать новое устройство, например с именем "coffee", со старшим номером 12 и младшим номером 2, нужно выполнить команду **mknod /dev/coffee c 12 2**. Вас никто не обязывает размещать файлы устройств в каталоге `/dev`, тем не менее, делается это в соответствии с принятыми соглашениями. Однако, при разработке драйвера устройства, на период отладки, размещать файл устройства в своем домашнем каталоге -- наверное не такая уж и плохая идея. Единственное -- не забудьте исправить место для размещения файла устройства после того, как отладка будет закончена.

Еще несколько замечаний, которые явно не касаются обсуждаемой темы, но которые мне хотелось бы сделать. Когда происходит обращение к файлу устройства, ядро использует старший номер файла, для определения драйвера, который должен обработать это обращение. Это означает, что ядро в действительности не использует и даже ничего не знает о младшем номере. Единственный, кто обеспокоен этим -- это сам драйвер. Он использует младший номер, чтобы отличить разные физические устройства.

Между прочим, когда я говорю "устройства", я подразумеваю нечто более абстрактное чем, скажем, PCI плата, которую вы можете подержать в руке. Взгляните на эти два файла устройств:

```
% ls -l /dev/fd0 /dev/fd0u1680
brwxrwxrwx   1 root  floppy   2,  0 Jul  5  2000 /dev/fd0
brw-rw----   1 root  floppy   2, 44 Jul  5  2000 /dev/fd0u1680
```

К настоящему моменту вы можете сказать об этих файлах устройств, что оба они - блочные устройства, что обслуживаются одним и тем же драйвером (старший номер 2). Вы можете даже заявить, что они оба представляют ваш дисковод для гибких дисков, несмотря на то, что у вас стоит только один дисковод. Но почему два файла? А дело вот в чем, один из них представляет дисковод для дискет, емкостью 1.44 Мб. Другой -- тот же самый дисковод, но для дискет емкостью 1.68 Мб, и соответствует тому, что некоторые люди называют "суперотформатированным" диском ("superformatted" disk). Такие дискеты могут хранить больший объем данных, чем стандартно-отформатированная дискета. Вот тот случай, когда два файла устройства, с различным младшими номерами, фактически представляют одно и то же физическое устройство. Так что, слово "устройство", в нашем обсуждении, может означать нечто более абстрактное.

Глава 4. Файлы символьных устройств

4.1. Структура `file_operations`

Структура file_operations определена в файле `linux/fs.h` и содержит указатели на функции драйвера, которые отвечают за выполнение различных операций с устройством. Например, практически любой драйвер символьного устройства реализует функцию чтения данных из устройства. Адрес этой функции, среди всего прочего, хранится в структуре file_operations. Ниже приводится определение структуры, взятое из исходных текстов ядра 2.6.5:

```
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(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
ssize_t(*write) (struct file *,
        const char __user *, size_t, loff_t *);
ssize_t(*aio_write) (struct kiocb *, const char __user *, size_t,
                     loff_t);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int,
              unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t(*readv) (struct file *, const struct iovec *, unsigned long,
                 loff_t *);
ssize_t(*writev) (struct file *, const struct iovec *, unsigned long,
                  loff_t *);
ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
                    void __user *);
ssize_t(*sendpage) (struct file *, struct page *, int, size_t,
                    loff_t *, int);
unsigned long (*get_unmapped_area) (struct file *, unsigned long,
                unsigned long, unsigned long,
                unsigned long);
};
```

Драйвер зачастую реализует далеко не все функции, предусмотренные данной структурой. Например, драйвер, который обслуживает видеоплату, не обязан выполнять операцию чтения каталога (`readdir`). Поля структуры, соответствующие нереализованным функциям, заполняются "пустыми" указателями -- NULL.

Компилятор **gcc** предоставляет программисту довольно удобный способ заполнения полей структуры в исходном тексте. Поэтому, если вы встретите подобный прием в современных драйверах, пусть это вас не удивляет. Ниже приводится пример подобного заполнения:

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

Однако, существует еще один способ заполнения структур, который описывается стандартом C99. Причем этот способ более предпочтителен. gcc 2.95, который я использую, поддерживает синтаксис C99. Вам так же следует придерживаться этого синтаксиса, если вы желаете обеспечить переносимость своему драйверу:

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

На мой взгляд все выглядит достаточно понятным. И еще, вы должны знать, что в любое поле структуры, которое вы явно не инициализируете, компилятор **gcc** запишет "пустой" указатель -- NULL. Указатель на `struct file_operations` обычно именуют как `fops`.

4.2. Структура `file`

Каждое устройство представлено в ядре структурой `file`, которая определена в файле `linux/fs.h`. Эта структура используется исключительно ядром и никогда не используется прикладными программами, работающими в пространстве пользователя. Это совершенно не то же самое, что и `FILE`, определяемое библиотекой `glibc` и которое в свою очередь в ядре нигде не используется. Имя структуры может ввести в заблуждение, поскольку она представляет абстракцию открытого файла, а не файла на диске, который представляет структура `inode`.

Как правило указатель на структуру `file` называют `filp`.

Загляните в заголовочный файл и посмотрите определение структуры `file`. Большинство имеющихся полей структуры, например `struct dentry *f_dentry`, не используются драйверами устройств, и вы можете игнорировать их. Драйверы не заполняют структуру `file` непосредственно, они только используют структуры, содержащиеся в ней.

4.3. Регистрация устройства

Как уже говорилось ранее, доступ к символьным устройствам осуществляется посредством файлов устройств, которые как правило располагаются в каталоге `/dev`. [5] Старший номер устройства говорит о том, какой драйвер с каким файлом устройства связан. Младший номер используется самим драйвером для идентификации устройства, если он обслуживает несколько таких устройств.

Добавление драйвера в систему подразумевает его регистрацию в ядре. Это означает -- получение старшего номера в момент инициализации модуля. Получить его можно вызовом функции `register_chrdev()`, определенной в файле `linux/fs.h`:

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

где `unsigned int major` -- это запрашиваемый старший номер устройства, `const char *name` -- название устройства, которое будет отображаться в `/proc/devices` и `struct file_operations *fops` -- указатель на таблицу `file_operations` драйвера. В случае ошибки, функция `register_chrdev()` возвращает отрицательное число. Обратите внимание: функции регистрации драйвера не передается младший номер устройства. Все потому, что ядро не обслуживает его -- это прерогатива драйвера.

А теперь вопрос: Как получить старший номер для своего устройства, чтобы случайно не "занять" уже существующий? Самый простой способ -- заглянуть в файл `Documentation/devices.txt` и выбрать один из неиспользуемых. Но это не самый лучший выход, потому что вы никогда не будете уверены в том, что выбранный вами номер не будет позднее официально связан с каким-либо другим устройством. Правильный ответ -- "попросить" ядро выделить вам динамический номер устройства.

Если вы передадите функции `register_chrdev()`, в качестве старшего номера, число 0, то возвращаемое положительное значение будет представлять собой, динамически выделенный ядром, старший номер устройства. Один из неприятных моментов здесь состоит в том, что вы заранее не можете создать файл устройства, поскольку старший номер устройства вам заранее не известен. Тем не менее, можно предложить ряд способов решения этой проблемы.

1. Драйвер может выводить сообщение в системный журнал (как это делает модуль "Hello World"), а вы затем вручную создадите файл устройства.

2. Для вновь зарегистрированного устройства, в файле `/proc/devices` появится запись. Вы можете найти эту запись и вручную создать файл устройства или можно написать небольшой сценарий, который выполнит эту работу за вас.

3. Можно "заставить" сам драйвер создавать файл устройства, с помощью системного вызова `mknod`, после успешной регистрации. А внутри `cleanup_module()` предусмотреть возможность удаления файла устройства с помощью `rm`.

4.4. Отключение устройства

Мы не можем позволить выгружать модуль по прихоти суперпользователя. Если файл устройства удалить после того как он будет открыт процессом, то может возникнуть ситуация когда процесс попытается обратиться к выгруженному драйверу (в конце концов процесс даже не подозревает, что такое могло произойти). В результате произойдет попытка обращения к тому участку памяти, где ранее находилась функция обработки запроса. Если вам повезет, то этот участок памяти окажется не затертым ядром и вы получите сообщение об ошибке. Если не повезет -- то произойдет переход в середину "чужой" функции. Результат такого "вызова" трудно предугадать заранее

Обычно, если какая-то операция должна быть отвергнута, функция возвращает код ошибки (отрицательное число). В случае с функцией `cleanup_module()` это невозможно, поскольку она не имеет возвращаемого значения. Однако, для каждого модуля в системе имеется счетчик обращений, который хранит число процессов, использующих модуль. Вы можете увидеть это число в третьем поле, в файле `/proc/devices`. Если это поле не равно нулю, то **rmmod** не сможет выгрузить модуль. Обратите внимание: вам нет нужды следить за состоянием счетчика в `cleanup_module()`, это делает система, внутри системного вызова `sys_delete_module` (определение функции вы найдете в файле `linux/module.c`). Вы не должны изменять значение счетчика напрямую, тем не менее, ядро предоставляет в ваше распоряжение функции, которые увеличивают и уменьшают значение счетчика обращений:

- `try_module_get(THIS_MODULE)`: увеличивает счетчик обращений на 1.

  `try_module_put(THIS_MODULE)`: уменьшает счетчик обращений на 1.

Очень важно сохранять точное значение счетчика! Если Вы каким-либо образом потеряете действительное значение, то вы никогда не сможете выгрузить модуль. Тут, милые мои мальчики и девочки, поможет только перезагрузка! Это обязательно случиться с вами, рано или поздно, при разработке какого-либо модуля!

4.5. chardev.c

Следующий пример создает устройство с именем `chardev`. Вы можете читать содержимое файла устройства с помощью команды **cat** или открывать его на чтение из программы (функцией `open()`). Посредством этого файла драйвер будет извещать о количестве попыток обращения к нему. Модуль не поддерживает операцию записи (типа: **echo "hi" > /dev/chardev**), но определяет такую попытку и сообщает пользователю о том, что операция записи не поддерживается.

Пример 4-1. chardev.c

```
/*
 *  chardev.c: Создает символьное устройство,
 *  доступное только для чтения
 *  возвращает сообщение, с указанием количества произведенных
 *  попыток чтения из файла устройства
 */
```

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>        /* определение функции put_user */

/*
 *  Прототипы функций, обычно их выносят в заголовочный файл (.h)
 */
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
/* Имя устройства, будет отображаться в /proc/devices   */
#define DEVICE_NAME "chardev"
#define BUF_LEN 80     /* Максимальная длина сообщения */

/*
 * Глобальные переменные, объявлены как static,
 * воизбежание конфликтов имен.
 */

/* Старший номер устройства нашего драйвера */
static int Major;
/* Устройство открыто?
static int Device_Open = 0;
* используется для предотвращения одновременного
* обращения из нескольких процессов */
/* Здесь будет собираться текст сообщения */
static char msg[BUF_LEN];
static char *msg_Ptr;

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

/*
 * Функции
 */

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

  if (Major < 0) {
    printk("Registering the character device failed with %d\n",
           Major);
    return Major;
  }

  printk("<1>I was assigned major number %d.  To talk to\n", Major);
  printk("<1>the driver, create a dev file with\n");
  printk("'mknod /dev/chardev c %d 0'.\n", Major);
  printk("<1>Try various minor numbers.  Try to cat and echo to\n");
  printk("the device file.\n");
  printk("<1>Remove the device file and module when done.\n");

  return 0;
}

void cleanup_module(void)
{
  /*
   * Отключение устройства
   */
  int ret = unregister_chrdev(Major, DEVICE_NAME);
  if (ret < 0)
    printk("Error in unregister_chrdev: %d\n", ret);
}

/*
 * Обработчики
 */

/*
 * Вызывается, когда процесс пытается
 * открыть файл устройства, например командой
```

```
 * "cat /dev/chardev"
 */
static int device_open(struct inode *inode, struct file *file)
{
  static int counter = 0;
  if (Device_Open)
    return -EBUSY;
  Device_Open++;
  sprintf(msg, "I already told you %d times 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--;  /* Теперь мы готовы обслужить другой процесс */

  /*
   * Уменьшить счетчик обращений, иначе, после первой
   * же удачной попытки открыть файл устройства,
   * вы никогда не сможете выгрузить модуль.
   */
  module_put(THIS_MODULE);

  return 0;
}

/*
 * Вызывается, когда процесс пытается
 * прочитать уже открытый файл устройства
 */
/* см. include/linux/fs.h   */
static ssize_t device_read(struct file *filp,
/* буфер, куда надо положить данные */
        char *buffer,
/* размер буфера */
        size_t length,
        loff_t * offset)
{
  /*
   * Количество байт, фактически записанных в буфер
   */
  int bytes_read = 0;

  /*
   * Если достигли конца сообщения,
   * вернуть 0, как признак конца файла
   */
  if (*msg_Ptr == 0)
    return 0;

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

    /*
     * Буфер находится в пространстве
     * пользователя (в сегменте данных),
     * а не в пространстве ядра, поэтому
     * простое присваивание здесь недопустимо.
     * Для того, чтобы скопировать данные,
     * мы используем функцию put_user,
     * которая перенесет данные из пространства
     * ядра в пространство пользователя.
     */
    put_user(*(msg_Ptr++), buffer++);

    length--;
    bytes_read++;
        }

  /*
   * В большинстве своем, функции чтения
   * возвращают количество байт, записанных в буфер.
   */
  return bytes_read;
}

/*
```

```
 * Вызывается, когда процесс пытается записать в устройство,
 * например так: echo "hi" > /dev/chardev
 */
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
  printk("<1>Sorry, this operation isn't supported.\n");
  return -EINVAL;
}
```

Пример 4-2. Makefile

```
obj-m += chardev.o
```

4.6. Создание модулей для работы с разными версиями ядра

Системные вызовы, которые суть есть основной интерфейс с ядром, как правило не изменяют свой синтаксис вызова от версии к версии. В ядро могут быть добавлены новые системные вызовы, но старые, практически всегда, сохраняют свое поведение, независимо от версии ядра. Делается это с целью сохранения обратной совместимости, чтобы не нарушить корректную работу ранее выпущенных приложений. В большинстве случаев, файлы устройств также останутся теми же самыми. С другой стороны, внутренние интерфейсы ядра могут изменяться от версии к версии.

Версии ядра подразделяются на стабильные (n.<четное_число>.m) и нестабильные (n.<нечетное_число>.m). Нестабильные версии несут в себе самые новые наработки, включая те, которые будут считаться ошибкой и те, которые претерпят существенные изменения в следующей версии. В результате, вы не можете доверять тому или иному интерфейсу, поскольку он может еще измениться (по этой причине я не посчитал нужным описывать их в этой книге -- слишком много работы, к тому же изменения происходят слишком быстро). От стабильных версий мы можем ожидать, что интерфейсы останутся неизменными, независимо от версии релиза (последнее число в номере версии -- m).

Итак, мы уже поняли, что между разными версиями ядра могут существовать весьма существенные отличия. Если у вас появится необходимость в создании модуля, который мог бы работать с разными версиями ядра, то можете воспользоваться директивами условной компиляции, основываясь на сравнении макроопределений `LINUX_VERSION_CODE` и `KERNEL_VERSION`. Для версии a.b.c, макрос `KERNEL_VERSION` вернет код версии, вычисленный в соответствии с выражением: 2^{16}a+2^{8}b+c. Макрос `LINUX_VERSION_CODE` возвращает текущую версию ядра.

В предыдущих версиях данного руководства, довольно подробно описывалось, как писать обратно совместимый код, с использованием директив условной компиляции. Но, начиная с этой версии, мы решили порвать с устоявшейся традицией. Теперь, если вы желаете писать модули под определенные версии ядра, обращайтесь к соответствующей версии руководства (LKMPG). Мы решили выпускать этот документ под версиями (номер версии и номер подверсии), совпадающими с версиями обсуждаемого ядра. Таким образом, разработчики, работающие под ядро 2.4.x, должны обращаться к LKMPG версии 2.4.x, работающие под ядро 2.6.x -- к LKMPG версии 2.6.x и т.д.

Глава 5. Файловая система `/proc`

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

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.

Методика работы с файловой системой `/proc` очень похожа на работу драйверов с файлами устройств: вы создаете структуру 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 `cleanup_module` отменяет регистрацию.

Основная причина, по которой используется `proc_register_dynamic` [6] состоит в том, что номер inode, для нашего файла, заранее неизвестен, поэтому мы даем возможность ядру 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.

Поскольку мы не предусматриваем обработку операций открытия/закрытия файла в файловой системе `/proc`, то нам некуда вставлять вызовы функций `try_module_get` и `try_module_put`. Если вдруг случится так, что модуль был выгружен в то время как соответствующий файл в `/proc` оставался открытым, к сожалению у нас не будет возможности избежать возможных последствий. В следующем разделе мы расскажем о довольно сложном, но достаточно гибком способе защиты от подобных ситуаций.

Пример 5-1. procfs.c

```
/*
 *  procfs.c -  пример создания "файла" в /proc
 */

/* Необходимо для любого модуля */
#include <linux/module.h>
/* Все-таки мы работаем с ядром! */
#include <linux/kernel.h>
/* Необходимо для работы с файловой системой /proc */
#include <linux/proc_fs.h>

struct proc_dir_entry *Our_Proc_File;

/* Обработчик чтения из файла в /proc.
```

```
 *
 * Аргументы
 * =========
 * 1. Буфер с данными. Как его заполнить -- вы решаете сами
 * 2. Указатель на указатель на строку символов.
 *    Если вы не желаете использовать буфер
 *    размещенный ядром.
 * 3. Текущая позиция в файле
 * 4. Размер буфера.
 * 5. Признак конца файла, "1" == EOF.
 * 6. Указатель на данные (необходим в случае единственного
 *    обработчика на несколько файлов в /proc)
 *
 * Порядок использования и возвращаемое значение
 * =============================================
 * Нулевое значение == "буфер пуст", т.е. "Конец файла".
 * Отрицательное значение == код ошибки.
 *
 * Дополнительные сведения
 * =======================
 * Основные принципы реализации этой функции
 * я почерпнул не из документации, а из исходных текстов
 * модулей, выполняющих подобные действия.
 * Меня интересовало использование
 * поля get_info в структуре proc_dir_entry (Если вам это интересно
 * то для поиска я пользовался утилитами find и grep),
 * Интересующий меня пример я нашел в <kernel source
 * directory>/fs/proc/array.c.
 *
 * Когда вам что-то непонятно, то лучше всего
 * поискать примеры в исходных текстах ядра. В этом состоит
 * огромное преимущество Linux перед другими ОС,
 * так как нам доступны все исходные тексты, так что --
 * пользуйтесь этим преимуществом!
 */
ssize_t
procfile_read(char *buffer,
              char **buffer_location,
              off_t offset, int buffer_length, int *eof, void *data)
{
  printk(KERN_INFO "inside /proc/test : procfile_read\n");

  int len = 0;          /* Фактическое число байт */
  static int count = 1;

  /*
   * Мы всегда должны выдавать имеющуюся информацию,
   * если пользователь спрашивает -- мы должны ответить.
   *
   * Это очень важно, поскольку библиотечная функция read
   * будет продолжать обращаться к системному вызову
   * read до тех пор, пока ядро не ответит, что сведений больше нет
   * или пока буфер не будет заполнен.
   */
  if (offset > 0) {
    printk(KERN_INFO "offset %d : /proc/test : procfile_read, \
           wrote %d Bytes\n", (int)(offset), len);
    *eof = 1;
    return len;
  }

  /*
   * Заполнить буфер и получить его размер
   */
  len = sprintf(buffer,
          "For the %d%s time, go away!\n", count,
          (count % 100 > 10 && count % 100 < 14) ? "th" :
          (count % 10 == 1) ? "st" :
          (count % 10 == 2) ? "nd" :
          (count % 10 == 3) ? "rd" : "th");
  count++;

 /*
  * Вернуть размер буфера
  */
  printk(KERN_INFO
         "leaving /proc/test : procfile_read, wrote %d Bytes\n", len);
  return len;
}

int init_module()
{
  int rv = 0;
  Our_Proc_File = create_proc_entry("test", 0644, NULL);
  Our_Proc_File->read_proc = procfile_read;
```

```
  Our_Proc_File->owner = THIS_MODULE;
  Our_Proc_File->mode = S_IFREG | S_IRUGO;
  Our_Proc_File->uid = 0;
  Our_Proc_File->gid = 0;
  Our_Proc_File->size = 37;

  printk(KERN_INFO "Trying to create /proc/test:\n");

  if (Our_Proc_File == NULL) {
    rv = -ENOMEM;
    remove_proc_entry("test", &proc_root);
    printk(KERN_INFO "Error: Could not initialize /proc/test\n");
  } else {
    printk(KERN_INFO "Success!\n");
  }

  return rv;
}

void cleanup_module()
{
  remove_proc_entry("test", &proc_root);
  printk(KERN_INFO "/proc/test removed\n");
}
```

Пример 5-2. Makefile

```
obj-m += procfs.o
```

5.2. Файловая система `/proc`: создание файлов, доступных для записи

Пока мы знаем о двух способах получения информации от драйвера устройства: можно зарегистрировать драйвер и создать файл устройства, и создать файл в файловой системе `/proc`. Единственная проблема -- мы пока ничего не можем передать модулю ядра. Для начала попробуем организовать передачу данных модулю ядра посредством файловой системы `/proc`.

Поскольку файловая система `/proc` была написана, главным образом, для того чтобы получать данные от ядра, она не предусматривает специальных средств для записи данных в файлы. Структура proc_dir_entry не содержит указатель на функцию-обработчик записи. Поэтому, вместо того, чтобы писать в `/proc` напрямую, мы вынуждены будем использовать стандартный, для файловой системы, механизм.

Linux предусматривает возможность регистрации файловой системы. Так как каждая файловая система должна иметь собственные функции, для обработки inode и выполнять файловые операции, [7] то имеется специальная структура, которая хранит указатели на все необходимые функции-обработчики -- `struct inode_operations`, которая включает указатель на `struct file_operations`. Файловая система `/proc`, всякий раз, когда мы регистрируем новый файл, позволяет указать -- какая `struct inode_operations` будет использоваться для доступа к нему. В свою очередь, в этой структуре имеется указатель `struct file_operations`, а в ней уже находятся указатели на наши функции-обработчики.

Обратите внимание: стандартные понятия "чтение" и "запись", в ядре имеют противоположный смысл. Функции чтения используются для записи в файл, в то время как функции записи используются для чтения из файла. Причина в том, что понятия "чтение" и "запись" рассматриваются здесь с точки зрения пользователя: если процесс читает что-то из ядра -- ядро должно записать эти данные, если процесс пишет -- ядро должно прочитать то, что записано.

Еще один интересный момент -- функция `module_permission`. Она вызывается всякий раз, когда процесс пытается обратиться к файлу в файловой системе `/proc`, и принимает решение -- разрешить доступ к файлу или нет. На сегодняшний день, решение принимается только на основе выполняемой операции и UID процесса, но в принципе возможна и иная организация принятия решения, например, разрешать ли одновременный доступ к файлу нескольким процессам и пр..

Причина, по которой для копирования данных используются функции `put_user` и `get_user`, состоит в том, что процессы в Linux (по крайней мере в архитектуре Intel) исполняются в изолированных адресных пространствах, не пересекающихся с адресным пространством ядра. Это означает, что указатель, не содержит уникальный адрес физической памяти -- он хранит логический адрес в адресном пространстве процесса.

Единственное адресное пространство, доступное процессу -- это его собственное адресное пространство. Практически любой модуль ядра, должен иметь возможность обмена информацией с пользовательскими процессами. Однако, когда модуль ядра получает указатель на некий буфер, то адрес этого буфера находится в адресном пространстве процесса. Макрокоманды `put_user` и `get_user` позволяют обращаться к памяти процесса по указанному им адресу.

Пример 5-3. procfs.c

```
/*
 *  procfs.c -  Пример создания файла в /proc,
 *  который доступен как на чтение, так и на запись.
 */
/* Необходимо для любого модуля */
#include <linux/module.h>
/* Все-таки мы работаем с ядром! */
#include <linux/kernel.h>
/* Необходимо для работы с файловой системой /proc */
#include <linux/proc_fs.h>
/* определения функций get_user и put_user */
#include <asm/uaccess.h>
```

```
/*
 * Место хранения последнего принятого сообщения,
 * которое будет выводиться в файл, чтобы показать, что
 * модуль действительно может получать ввод от пользователя
 */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];
static struct proc_dir_entry *Our_Proc_File;

#define PROC_ENTRY_FILENAME "rw_test"

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

  /*
   * Для индикации признака конца файла возвращается 0.
   * Если этого не сделать, процесс будет продолжать
   * пытаться читать из файла,
   * угодив в бесконечный цикл.
   */
  if (finished) {
    finished = 0;
    return 0;
  }

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

  /*
   * Обратите внимание: в данной ситуации мы исходим из предположения,
   * что размер сообщения меньше, чем len,
   * в противном случае сообщение будт обрезано.
   * В реальной ситуации, если длина сообщения больше чем
   * len, то возвращается len, а остаток сообщения возвращается
   * на последующих вызовах.
   */
  finished = 1;

  return i;             /* Вернуть количество "прочитанных" байт */
}

static ssize_t
module_input(struct file *filp,
   const char *buff, size_t len, loff_t * off)
{
  int i;
  /*
   * Переместить данные, полученные от пользователя в буфер,
   * который позднее будет выведен функцией module_output.
   */
  for (i = 0; i < MESSAGE_LENGTH - 1 && i < len; i++)
    get_user(Message[i], buff + i);

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

/*
 * Эта функция принимает решение о праве на выполнение операций с файлом
 * 0 -- разрешено, ненулеое значение -- запрещено.
 *
 * Операции с файлом могут быть:
 * 0 - Исполнениее (не имеет смысла в нашей ситуации)
 * 2 - Запись (передача от пользователя к модулю ядра)
 * 4 - Чтение (передача от модуля ядра к пользователю)
 *
 * Эта функция проверяет права доступа к файлу
 * Права, выводимые командой ls -l
 * могут быть проигнорированы здесь.
 */
```

```
static int module_permission(struct inode *inode,
   int op, struct nameidata *foo)
{
  /*
   * Позволим любому читать файл, но
   * писать -- только root-у (uid 0)
   */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;

  /*
   * Если что-то иное -- запретить доступ
   */
        return -EACCES;
}

/*
 * Файл открыт -- пока нам нет нужды беспокоиться о чем-то
 * единственное, что нужно сделать -- это нарастить
 * счетчик обращений к модулю.
 */
int module_open(struct inode *inode, struct file *file)
{
  try_module_get(THIS_MODULE);
  return 0;
}

/*
 * Файл закрыт -- уменьшить счетчик обращений.
 */
int module_close(struct inode *inode, struct file *file)
{
  module_put(THIS_MODULE);
  return 0;             /* все нормально! */
}

static struct file_operations File_Ops_4_Our_Proc_File = {
        .read = module_output,
        .write = module_input,
        .open = module_open,
        .release = module_close,
};

/*
 * Операции над индексной записью нашего файла. Необходима
 * для того, чтобы указать местоположение структуры
 * file_operations нашего файла, а так же, чтобы задать адрес
 * функции определения прав доступа к файлу. Здесь можно указать адреса
 * других функций-обработчиков, но нас они не интересуют.
 */

static struct inode_operations Inode_Ops_4_Our_Proc_File = {
  .permission = module_permission,      /*  проверка прав доступа */
};

/*
 * Начальная и конечная функции модуля
 */
int init_module()
{
  int rv = 0;
  Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);
  Our_Proc_File->owner = THIS_MODULE;
  Our_Proc_File->proc_iops = &Inode_Ops_4_Our_Proc_File;
  Our_Proc_File->proc_fops = &File_Ops_4_Our_Proc_File;
  Our_Proc_File->mode = S_IFREG | S_IRUGO | S_IWUSR;
  Our_Proc_File->uid = 0;
  Our_Proc_File->gid = 0;
  Our_Proc_File->size = 80;

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

  return rv;
}

void cleanup_module()
{
  remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
}
```

Хотите еще примеры работы с файловой системой `/proc`? Хорошо, но имейте ввиду, ходят слухи, что `/proc` уходит в небытие и вместо нее следует использовать sysfs. Дополнительные сведения о файловой системе `/proc` вы найдете в `linux/Documentation/DocBook/` . Дайте команду **make help**, она выведет инструкции по созданию документации в различных форматах, например: **make htmldocs**.

Глава 6. Работа с файлами устройств

Файлы устройств представляют физические устройства. В большинстве своем, физические устройства используются как для вывода, так и для ввода, таким образом необходимо иметь некий механизм для передачи данных от процесса (через модуль ядра) к устройству. Один из вариантов -- открыть файл устройства и записать в него данные, точно так же, как в обычный файл. В следующем примере, операция записи реализуется функцией `device_write`.

Однако, этого не всегда бывает достаточно. Допустим, что у вас есть модем, подключенный к компьютеру через последовательный порт (это может быть и внутренний модем, с точки зрения CPU он "выглядит" как модем, связанный с последовательным портом). Естественное решение -- использовать файл устройства для передачи данных модему (это могут быть команды модема или данные, которые будут посланы в телефонную линию) и для чтения данных из модема (ответы модема на команды или данные, полученные из телефонной линии). Однако, это оставляет открытым вопрос о том, как взаимодействовать непосредственно с последовательным портом, например, как настроить скорость обмена.

Ответ: в Unix следует использовать специальную функцию с именем `ioctl` (сокращенно от Input Output ConTroL). Любое устройство может иметь свои команды `ioctl`, которые могут читать (для передачи данных от процесса ядру), писать (для передачи данных от ядра к процессу), и писать и читать, и ни то ни другое, [8] Функция `ioctl` вызывается с тремя параметрами: дескриптор файла устройства, номер `ioctl` и третий параметр, который имеет тип long, используется для передачи дополнительных аргументов. [9]

Номер ioctl содержит комбинацию бит, составляющих старший номер устройства, тип команды и тип дополнительного параметра. Обычно номер ioctl создается макроопределением (_IO, _IOR, _IOW или _IOWR, в зависимости от типа) в файле заголовка. Этот заголовочный должен подключаться директивой #include, к исходным файлам программы, которая использует ioctl для обмена данными с модулем. В примере, приводимом ниже, представлены файл заголовка chardev.h и программа, которая взаимодействует с модулем ioctl.c.

Если вы предполагаете использовать ioctl в ваших собственных модулях, то вам надлежит обратиться к файлу `Documentation/ioctl-number.txt` с тем, чтобы не "занять" зарегистрированные номера ioctl.

Пример 6-1. chardev.c

```
/*
 *  chardev.c - Пример создания символьного устройства
 *              доступного на запись/чтение
 */

#include <linux/module.h> /* Необходимо для любого модуля */
#include <linux/kernel.h> /* Все-таки мы работаем с ядром! */
#include <linux/fs.h>
#include <asm/uaccess.h>  /* определения функций get_user и put_user */

#include "chardev.h"
#define SUCCESS 0
#define DEVICE_NAME "char_dev"
#define BUF_LEN 80

/*
 * Устройство уже открыто? Используется для
 * предотвращения конкурирующих запросов к устройству
 */
static int Device_Open = 0;

/*
 * Ответ устройства на запрос
 */
static char Message[BUF_LEN];

/*
 * Позиция в буфере.
 * Используется в том случае, если сообщение оказывется длиннее
 * чем размер буфера.
 */
static char *Message_Ptr;

/*
 * Вызывается когда процесс пытается открыть файл устройства
 */
static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
  printk("device_open(%p)\n", file);
#endif

  /*
   * В каждый конкретный момент времени только
   * один процесс может открыть файл устройства
   */
  if (Device_Open)
    return -EBUSY;
```

```
  Device_Open++;
  /*
   * Инициализация сообщения
   */
  Message_Ptr = Message;
  try_module_get(THIS_MODULE);
  return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
  printk("device_release(%p,%p)\n", inode, file);
#endif

  /*
   * Теперь мы готовы принять запрос от другого процесса
   */
  Device_Open--;

  module_put(THIS_MODULE);
  return SUCCESS;
}

/*
 * Вызывается когда процесс, открывший файл устройства
 * пытается считать из него данные.
 */
static ssize_t device_read(struct file *file, /* см. include/linux/fs.h*/
            char __user * buffer,             /* буфер для сообщения */
            size_t length,                    /* размер буфера       */
            loff_t * offset)
{
  /*
   * Количество байт, фактически записанных в буфер
   */
  int bytes_read = 0;

#ifdef DEBUG
  printk("device_read(%p,%p,%d)\n", file, buffer, length);
#endif

  /*
   * Если достигнут конец сообщения -- вернуть 0
   * (признак конца файла)
   */
  if (*Message_Ptr == 0)
    return 0;

  /*
   * Собственно запись данных в буфер
   */
  while (length && *Message_Ptr) {

    /*
     * Поскольку буфер располагается в пространстве пользователя,
     * обычное присвоение не сработает. Поэтому
     * для записи данных используется put_user,
     * которая копирует данные из пространства ядра
     * в пространство пользователя.
     */
    put_user(*(Message_Ptr++), buffer++);
    length--;
    bytes_read++;
  }

#ifdef DEBUG
  printk("Read %d bytes, %d left\n", bytes_read, length);
#endif

  /*
   * Вернуть количество байт, помещенных в буфер.
   */
  return bytes_read;
}

/*
 * Вызывается при попытке записи в файл устройства
 */
static ssize_t
device_write(struct file *file,
            const char __user * buffer, size_t length, loff_t * offset)
{
  int i;
```

```
#ifdef DEBUG
  printk("device_write(%p,%s,%d)", file, buffer, length);
#endif

  for (i = 0; i < length && i < BUF_LEN; i++)
    get_user(Message[i], buffer + i);

  Message_Ptr = Message;

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

/*
 * Вызывается, когда процесс пытается
 * выполнить операцию ioctl над файлом устройства.
 * Кроме inode и структуры file функция получает
 * два дополнительных параметра:
 * номер ioctl и дополнительные аргументы.
 *
 */
int device_ioctl(struct inode *inode, /* см. include/linux/fs.h */
      struct file *file,              /* то же самое */
      unsigned int ioctl_num,         /* номер и аргументы ioctl */
      unsigned long ioctl_param)
{
  int i;
  char *temp;
  char ch;

  /*
   * Реакция на различные команды ioctl
   */
  switch (ioctl_num) {
  case IOCTL_SET_MSG:
    /*
     * Принять указатель на сообщение (в пространстве пользователя)
     * и переписать в буфер.
     * Адрес которого задан в дополнительно аргументе.
     */
    temp = (char *)ioctl_param;

    /*
     * Найти длину сообщения
     */
    get_user(ch, temp);
    for (i = 0; ch && i < BUF_LEN; i++, temp++)
      get_user(ch, temp);

    device_write(file, (char *)ioctl_param, i, 0);
    break;

  case IOCTL_GET_MSG:
    /*
     * Передать текущее сообщение вызывающему процессу -
     * записать по указанному адресу.
     */
    i = device_read(file, (char *)ioctl_param, 99, 0);

    /*
     * Вставить в буфер завершающий символ \0
     */
    put_user('\0', (char *)ioctl_param + i);
    break;

  case IOCTL_GET_NTH_BYTE:
    /*
     * Этот вызов является вводом (ioctl_param) и
     * выводом (возвращаемое значение функции) одновременно
     */
    return Message[ioctl_param];
    break;
  }

  return SUCCESS;
}

/* Объявлнеия */

/*
 * В этой структуре хранятся адреса функций-обработчиков
 * операций, производимых процессом над устройством.
 * Поскольку указатель на эту структуру хранится в таблице устройств,
```

```
 * она не может быть локальной для init_module.
 * Отсутствующие указатели в структуре забиваются значением NULL.
 */
struct file_operations Fops = {
  .read = device_read,
  .write = device_write,
  .ioctl = device_ioctl,
  .open = device_open,
  .release = device_release,    /* оно же close */
};

/*
 * Инициализация модуля - Регистрация символьного устройства
 */
int init_module()
{
  int ret_val;
  /*
   * Регистрация символьного устройства
   * (по крайней мере - попытка регистрации)
   */
  ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &Fops);

  /*
   * Отрицательное значение означает ошибку
   */
  if (ret_val < 0) {
    printk("%s failed with %d\n",
           "Sorry, registering the character device ", ret_val);
    return ret_val;
  }

  printk("%s The major device number is %d.\n",
         "Registeration is a success", MAJOR_NUM);
  printk("If you want to talk to the device driver,\n");
  printk("you'll have to create a device file. \n");
  printk("We suggest you use:\n");
  printk("mknod %s c %d 0\n", DEVICE_FILE_NAME, MAJOR_NUM);
  printk("The device file name is important, because\n");
  printk("the ioctl program assumes that's the\n");
  printk("file you'll use.\n");

  return 0;
}

/*
 * Завершение работы модуля - дерегистрация файла в /proc
 */
void cleanup_module()
{
  int ret;

  /*
   * Дерегистрация устройства
   */
  ret = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

  /*
   * Если обнаружена ошибка -- вывести сообщение
   */
  if (ret < 0)
    printk("Error in module_unregister_chrdev: %d\n", ret);
}
```

Пример 6-2. chardev.h

```
/*
 *  chardev.h - определения ioctl.
 *
 *  Определения, которые здесь находятся,
 *  должны помещаться в заголовочный файл потому,
 *  что они потребуются как модулю ядра (chardev.c), так и
 *  вызывающему процессу (ioctl.c)
 */

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/*
 * Старший номер устройства. В случае использования ioctl,
 * мы уже лишены возможности воспользоваться динамическим номером,
```

```
 * поскольку он должен быть известен заранее.
 */
#define MAJOR_NUM 100

/*
 * Операция передачи сообщения драйверу устройства
 */
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
/*
 * _IOR означает, что команда передает данные
 * от пользовательского процесса к модулю ядра
 *
 * Первый аргумент, MAJOR_NUM -- старший номер устройства.
 *
 * Второй аргумент -- код команды
 * (можно указать иное значение).
 *
 * Третий аргумент -- тип данных, передаваемых в ядро
 */

/*
 * Операция получения сообщения от драйвера устройства
 */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
/*
 * Эта команда IOCTL используется для вывода данных.
 * Нам по прежнему нужен буфер, размещенный в адресном пространстве
 * вызывающего процесса, куда это сообщение должно быть переписано.
 */

/*
 * Команда получения n-ного байта сообщения
 */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
/*
 * Здесь команда IOCTL работает как на ввод, так и на вывод.
 * Она принимает от пользователя номер байта (n),
 * и возвращает n-ный байт сообщения (Message[n]).
 */

/*
 * Имя файла устройства
 */
#define DEVICE_FILE_NAME "char_dev"

#endif
```

Пример 6-3. ioctl.c

```
/*
 *  ioctl.c - Пример программы, использующей
 *            ioctl для управления модулем ядра
 *
 *  До сих пор мы ползовались командой cat,
 *  для передачи данных в/из модуля.
 *  Теперь же мы должны написать свою программу,
 *  которая использовала бы ioctl.
 */

/*
 * Определения старшего номера устройства и коды операций ioctl
 */
#include "chardev.h"

#include <fcntl.h>      /* open */
#include <unistd.h>     /* exit */
#include <sys/ioctl.h>  /* ioctl */

/*
 * Функции работы с драйвером через ioctl
 */

ioctl_set_msg(int file_desc, char *message)
{
  int ret_val;

  ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

  if (ret_val < 0) {
    printf("Ошибка при вызове ioctl_set_msg: %d\n", ret_val);
    exit(-1);
  }
}
```

```
ioctl_get_msg(int file_desc)
{
  int ret_val;
  char message[100];

  /*
   * Внимание - ядро понятия не имеет -- какой длины буфер мы используем
   * поэтому возможна ошибка, связанная с переполнением буфера.
   * В реальных проектах вам необходимо предусмотреть
   * передачу в ioctl двух дополнительных параметров:
   * собственно буфера сообщения и его длину
   */
  ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

  if (ret_val < 0) {
    printf("Ошибка при вызове ioctl_get_msg: %d\n", ret_val);
    exit(-1);
  }

  printf("Получено сообщение (get_msg): %s\n", message);
}

ioctl_get_nth_byte(int file_desc)
{
  int i;
  char c;

  printf("N-ный байт в сообщении (get_nth_byte): ");

  i = 0;
  while (c != 0) {
    c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);

    if (c < 0) {
      printf
        ("Ошибка при вызове ioctl_get_nth_byte на %d-м байте.\n", i);
      exit(-1);
    }

    putchar(c);
  }
  putchar('\n');
}

/*
 * Main - Проверка работоспособности функции ioctl
 */
main()
{
  int file_desc, ret_val;
  char *msg = "Это сообщение передается через ioctl\n";

  file_desc = open(DEVICE_FILE_NAME, 0);
  if (file_desc < 0) {
    printf("Невозможно открыть файл устройства: %s\n", DEVICE_FILE_NAME);
    exit(-1);
  }

  ioctl_get_nth_byte(file_desc);
  ioctl_get_msg(file_desc);
  ioctl_set_msg(file_desc, msg);

  close(file_desc);
}
```

Пример 6-4. Makefile

```
obj-m += chardev.o
```

Для облегчения сборки примера, предлагается скрипт, который выполнит эту работу за вас:

Пример 6-5. build.sh

```
#/bin/sh

# сборка пользовательского приложения
gcc -o ioctl ioctl.c

# создание файла устройства
mknod char_dev c 100 0
```

Глава 7. Системные вызовы

До сих пор, все что мы делали, это использовали ранее определенные механизмы ядра для регистрации файлов в файловой системе `/proc` и файлов устройств. Это все замечательно, но это годится только для создания драйверов устройств. А что если вы хотите сделать действительно что-то необычное, например изменить реакцию системы на какое либо событие?

Мы как раз вступаем в ту область, где программирование действительно становится опасным. При разработке примера, приведенного ниже, я "уничтожил" системный вызов `open`. В результате система потеряла возможность открывать любые файлы, что равносильно отказу системы выполнять любые программы. Я не мог даже остановить систему командой **shutdown**. Из-за этого пришлось прибегнуть к "помощи" кнопки выключения питания. К счастью, ни один файл не был уничтожен. Чтобы обезопасить себя от потери данных, в аналогичных ситуациях, перед загрузкой подобных модулей всегда выполняйте резервное копирование.

Забудьте про `/proc`, забудьте и про файлы устройств. *Реальный* механизм взаимодействия процессов с ядром -- это системные вызовы. Когда процесс запрашивает какую-либо услугу ядра (например, открытие файла, запуск нового процесса или выделение дополнительной памяти), используется механизм системных вызовов. Если вы хотите изменить поведение ядра, то системные вызовы -- это как раз то место, куда можно приложить свои знания и умения. Между прочим, если вы захотите увидеть -- какие системные вызовы используются той или иной программой, запустите: **strace <command> <arguments>**.

Строго говоря, процесс не имеет доступа в пространство ядра. Он не может обращаться к памяти ядра и не может вызывать функции в ядре. Микропроцессор ограничивает такого рода доступ на аппаратном уровне (вот почему режим исполнения ядра называется *защищенным*, или *привилегированным*).

Системные вызовы являются исключением из этого правила. Чтобы исполнить системный вызов, процесс заполняет регистры микропроцессора соответствующими значениями и выполняет специальную инструкцию, которая производит переход в предопределенное место в пространстве ядра (разумеется, точка перехода доступна пользовательским процессам на чтение). Для платформы Intel -- это инструкция прерывания с вектором 0x80. Микропроцессор воспринимает это как переход из ограниченного пользовательского режима в защищенный режим ядра, где позволено делать все, что вам заблагорассудится.

Точка перехода, в ядре, называется `system_call`. Процедура, которая там находится, проверяет номер системного вызова, который сообщает ядру -- какую именно услугу запрашивает процесс. Затем, она просматривает таблицу системных вызовов (`sys_call_table`), отыскивает адрес функции ядра, которую следует вызвать, после чего вызывается нужная функция. По окончании работы системного вызова, выполняется ряд дополнительных проверок и лишь после этого управление возвращается вызывающему процессу (или другому процессу, если вызывающий процесс исчерпал свой квант времени). Код, выполняющий все вышеперечисленные действия, вы найдете в файле `arch/<architecture>/kernel/entry.S`, после строки `ENTRY(system_call)`.

Итак, если вас одолевает желание изменить поведение некоторого системного вызова, то первое, что необходимо сделать -- это написать вашу собственную функцию, которая выполняла бы требуемые действия (обычно, после выполнения своих действий, в подобных случаях, вызывается первоначальная функция, реализующая системный вызов), затем -- изменить указатель в `sys_call_table` так, чтобы он указывал на вашу функцию. Поскольку ваш модуль впоследствии может быть выгружен, то следует предусмотреть восстановление системы в ее первоначальное состояние, чтобы не оставлять ее в нестабильном состоянии. Это делается в пределах функции `cleanup_module`.

Ниже приводится исходный текст такого модуля. Он "шпионит" за выбранным пользователем, и посылать через printk сообщение всякий раз, когда данный пользователь открывает какой-либо файл. Для этого, системный вызов `open()`, подменяется функцией с именем `our_sys_open`. Она проверяет UID (User ID) текущего процесса, и если он равен заданному, то вызывает printk, чтобы сообщить имя открываемого файла, и в заключение вызывает оригинальную функцию `open()` с теми же параметрами, которая открывает требуемый файл.

Функция `init_module` изменяет соответствующий указатель в `sys_call_table` и сохраняет его первоначальное значение в переменной. Функция `cleanup_module` восстанавливает указатель в `sys_call_table`, используя эту переменную. В данном подходе кроются свои "подводные камни" из-за возможности существования двух модулей, перекрывающих один и тот же системный вызов. Представьте себе: имеется два модуля, A и B. Пусть модуль A перекрывает системный вызов `open`, своей функцией `A_open`, а модуль B -- функцией `B_open`. Первым загружается модуль A, он заменяет системный вызов `open` на `A_open`. Затем загружается модуль B, который заменит системный вызов `A_open` на `B_open`. Модуль B полагает, что он подменил оригинальный системный вызов, хотя на самом деле был подменен вызов `A_open`.

Теперь, если модуль B выгрузить первым, то ничего страшного не произойдет -- он просто восстановит запись в таблице `sys_call_table` в значение `A_open`, который в свою очередь вызывает оригинальную функцию `sys_open`. Однако, если первым будет выгружен модуль A, а затем B, то система "рухнет". Модуль A восстановит адрес в `sys_call_table`, указывающий на оригинальную функцию `sys_open`, "отсекая" таким образом модуль B от обработки действий по открытию файлов. Затем, когда будет выгружен модуль B, он восстановит адрес в `sys_call_table` на тот, который запомнил сам, потому что он считает его оригинальным. Т.е. вызовы будут направлены в функцию `A_open`, которой уже нет в памяти. На первый взгляд, проблему можно решить, проверкой -- совпадает ли адрес в `sys_call_table` с адресом нашей функции `open` и если не совпадает, то не восстанавливать значение этого вызова (таким образом B не будет "восстанавливать" системный вызов), но это порождает другую проблему. Когда выгружается модуль A, он "видит", что системный вызов был изменен на `B_open` и "отказывается" от восстановления указателя на `sys_open`. Теперь, функция `B_open` будет по прежнему пытаться вызывать `A_open`, которой больше не существует в памяти, так что система "рухнет" еще раньше -- до удаления модуля B.

Обратите внимание: подобные проблемы делают такую "подмену" системных вызовов неприменимой для широкого распространения.С целью предотвращения потенциальной опасности, связанной с подменой адресов системных вызовов, ядро более не экспортирует `sys_call_table`. Поэтому, если вы желаете сделать нечто большее, чем просто пробежать глазами по тексту данного примера, вам надлежит наложить "заплату" на ядро. В каталоге с примерами вы найдете файл `README` и "заплату". Как вы наверняка понимаете, подобные модификации сопряжены с определенными трудностями, поэтому я не рекомендую производить их на системах, владельцем которых вы не являетесь или не в состоянии быстро восстановить. Если вас одолевают сомнения, то лучшим выбором будет отказ от прогона этого примера.

Пример 7-1. syscall.c

```
/*
 *  syscall.c
 *
 *  Пример "перехвата" системного вызова.
 */
/*
 * Copyright (C) 2001 by Peter Jay Salzman
 */

/*
```

```
 * Необходимые заголовочные файлы
 */

#include <linux/kernel.h>      /* Все-таки мы работаем с ядром! */
#include <linux/module.h>      /* Необходимо для любого модуля */
#include <linux/moduleparam.h> /* для передачи параметров модулю */
#include <linux/unistd.h>      /* Список системных вызовов */

/*
 * Необходимо, чтобы уметь определять
 * user id вызвавшего процесса.
 */
#include <linux/sched.h>
#include <asm/uaccess.h>

/*
 * Таблица системных вызовов (таблица адресов функций).
 * Просто определим ее как ссылку на внешнюю таблицу.
 *
 * sys_call_table больше не экспортируется ядрами 2.6.x.
 * Если вы намереваетесь опробовать этот ОПАСНЫЙ модуль,
 * вам следует наложить "заплату" на ядро и пересобрать его
 */
extern void *sys_call_table[];

/*
 * UID пользователя, за которым "шпионим",
 * принимается из командной строки
 */
static int uid;
module_param(uid, int, 0644);

/*
 * Указатель на оригинальную функцию, выполняющую системный вызов.
 * Мы сохраняем ее, вместо того, чтобы напрямую вызывать оригинальную
 * функцию, для того, чтобы имелась возможность вызова
 * обработчиков, вставленных до нас
 * Это не гарантирует 100% безопасность, поскольку другой модуль,
 * замещающий sys_open может быть выгружен раньше нашего модуля.
 *
 * Другая причина -- мы не можем получить адрес оригинальной sys_open.
 * Этот адрес не экспортируется ядром.
 */
asmlinkage int (*original_call) (const char *, int, int);

/*
 * Функция, замещающая sys_open (вызывается
 * всякий раз, когда делается обращение к системному вызову open).
 * Прототип функции, количество аргументов и их тип
 * вы найдете в fs/open.c.
 *
 * Теоретически - мы "привязаны" к данной конкретной
 * версии ядра. Практически -- системные вызовы
 * очень редко подвергаются кардинальному изменению,
 * поскольку это сделало бы огромное количество программного обеспечения
 * несовместимым с ядром и потребовало бы их пересборки.
 */
asmlinkage int our_sys_open(const char *filename, int flags, int mode)
{
  int i = 0;
  char ch;

  /*
   * Проверить -- это искомый пользователь?
   */
  if (uid == current->uid) {
    /*
     * Зафиксировать
     */
    printk("Opened file by %d: ", uid);
    do {
      get_user(ch, filename + i);
      i++;
      printk("%c", ch);
    } while (ch != 0);
    printk("\n");
  }

  /*
   * Вызвать оригинальную версию системного вызова sys_open - иначе
   * система потеряет возможность открывать файлы
   */
  return original_call(filename, flags, mode);
}
```

```
/*
 * Инициализация модуля - подмена системного вызова
 */
int init_module()
{
  /*
   * Внимание - предупреждение запоздало, но
   * может быть в следующий раз...
   */
  printk("I'm dangerous. I hope you did a ");
  printk("sync before you insmod'ed me.\n");
  printk("My counterpart, cleanup_module(), is even");
  printk("more dangerous. If\n");
  printk("you value your file system, it will ");
  printk("be \"sync; rmmod\" \n");
  printk("when you remove this module.\n");

  /*
   * Сохранить указатель на оригинальную функцию
   * в переменной original_call, и затем заменить указатель
   * в таблице системных вызовов
   */
  original_call = sys_call_table[__NR_open];
  sys_call_table[__NR_open] = our_sys_open;

  /*
   * Чтобы получить адрес любого системного вызова
   * с именем foo, обращайтесь к записи sys_call_table[__NR_foo].
   */

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

  return 0;
}

/*
 * Завершение работы модуля - восстановление
 * указателя на оригинальный системный вызов
 */
void cleanup_module()
{
  /*
   * Восстановить адрес системного вызова
   */
  if (sys_call_table[__NR_open] != our_sys_open) {
    printk("Somebody else also played with the ");
    printk("open system call\n");
    printk("The system may be left in ");
    printk("an unstable state.\n");
  }

  sys_call_table[__NR_open] = original_call;
}
```

Пример 7-2. "Заплата" на ядро (export_sys_call_table_patch_for_linux_2.6.x)

```
--- kernel/kallsyms.c.orig     2003-12-30 07:07:17.000000000 +0000
+++ kernel/kallsyms.c   2003-12-30 07:43:43.000000000 +0000
@@ -184,7 +184,7 @@
                iter->pos = pos;
                return get_ksymbol_mod(iter);
        }
-
+
        /* If we're past the desired position, reset to start. */
        if (pos < iter->pos)
                reset_iter(iter);
@@ -291,3 +291,11 @@

 EXPORT_SYMBOL(kallsyms_lookup);
 EXPORT_SYMBOL(__print_symbol);
+/* START OF DIRTY HACK:
+ * Purpose: enable interception of syscalls as shown in the
+ * Linux Kernel Module Programming Guide. */
+extern void *sys_call_table;
+EXPORT_SYMBOL(sys_call_table);
+ /* see http://marc.free.net.ph/message/20030505.081945.fa640369.html
+  * for discussion why this is a BAD THING(tm) and no
     longer supported by 2.6.0
+  * END OF DIRTY HACK: USE AT YOUR OWN RISK */
```

Пример 7-3. Makefile

```
obj-m += syscall.o
```

Пример 7-4. README.txt

```
Основная проблема, связанная с данным примером,
состоит в невозможности определить адрес sys_call_table,
поскольку он более не экспортируется ядрами 2.6.x.
Возможность "перекрытия" системных вызовов через
sys_call_table  потенциально опасна поэтому, начиная
с версии 2.5.41, она больше не поддерживается

Обсуждение проблемы вы найдете на:

  http://www.ussg.iu.edu/hypermail/linux/kernel/0305.0/0711.html
  http://marc.free.net.ph/message/20030505.081945.fa640369.html
  http://marc.theaimsgroup.com/?l=linux-kernel&m=105212296015799&w=2


Чтобы иметь возможность опробовать данный
пример на ядрах версии 2.5.41 и выше
вам необходимо наложить заплату на ядро.

ВНИМАНИЕ:
        НЕ ИСПОЛЬЗУЙТЕ ЭТУ ЗАПЛАТУ НА ПРОМЫШЛЕННЫХ ИЛИ ИНЫХ СИСТЕМАХ,
        КОТОРЫЕ СОДЕРЖАТ ЦЕННУЮ ИНФОРМАЦИЮ.


Если бы я писал встроенную справку к этой заплате в Configure.help
то я бы пометил ее как <dangerous> и дал бы следующее описание:

#############################################################################

Эта опция экспортирует sys_call_table, что делает возможным
"перекрытие" (подмену) системных вызовов.
Подмена системных вызовов потенциально опасна
и может стать причиной потери даных или еще хуже.

Скажите Y, если желаете опробовать прилагаемый пример
и вас не беспокоит возможная потеря данных.

Практически любой должен здесь сказать N.

#############################################################################

Если ваш старенький PC используется только как игрушка
можете наложить эту заплату и опробовать пример.

Предполагается, что исходные тексты ядра 2.6.x
находятся в каталоге /usr/src/linux/
(http://www.linuxmafia.com/faq/Kernel/usr-src-linux-symlink.html )

Ниже приводится текст сценария, выполняющий наложение заплаты.


Эта заплата протестирована с ядрами 2.6.[0123],
и может накладываться или не накладываться
на другие версии.



#!/bin/sh
cp export_sys_call_table_patch_for_linux_2.6.x /usr/src/linux/
cd /usr/src/linux/
patch -p0 < export_sys_call_table_patch_for_linux_2.6.x
```

Глава 8. Блокировка процессов

Что вы делаете, когда кто-то просит вас о чем-то, а вы не можете сделать это немедленно? Пожалуй единственное, что вы можете ответить: "Пожалуйста, не сейчас, я пока занят.". А что должен делать модуль ядра? У него есть другая возможность. Он можете приостановить работу процесса до тех пор, пока не сможет обслужить его. В конечном итоге, ядро постоянно то приостанавливает, то вновь возобновляет работу процессов. Именно так обеспечивается возможность одновременного исполнения нескольких процессов на единственном процессоре.

Пример ниже демонстрирует такую возможность. Модуль создает файл `/proc/sleep`, который может быть открыт только одним процессом, в каждый конкретный момент времени. Если файл уже был открыт кем-нибудь, то модуль вызывает `wait_event_interruptible`. [10] Эта функция изменяет состояние "задачи" (здесь, под термином "задача" понимается структура данных в ядре, которая хранит информацию о процессе), присваивая ему значение `TASK_INTERRUPTIBLE`, это означает, что задача не будет выполняться до тех пор, пока не будет "разбужена" каким либо образом, и добавляет процесс в очередь ожидания `WaitQ`, куда помещаются все процессы, желающие открыть файл `/proc/sleep`. Затем функция передает управление планировщику, который в свою очередь предоставляет возможность поработать другому процессу.

Когда процесс закрывает файл, это приводит к вызову `module_close`. Она запускает все процессы, которые "сидят" в очереди `WaitQ` (к сожалению нет механизма, который позволил бы "разбудить" только один процесс). Затем управление возвращается процессу, который только что закрыл файл и он продолжает свою работу. После того, как данный процесс исчерпает свой квант времени, планировщик передаст управление другому процессу. Таким образом, один из процессов, ожидавших своей очереди доступа к файлу, в конечном итоге получит управление и продолжит исполнение с точки, следующей за вызовом `wait_event_interruptible`. [11] Он установит глобальную переменную, извещающую остальные процессы о том, что файл открыт и займется обработкой открытого файла. Когда другие процессы получат свой квант времени, они обнаружат, что файл все еще открыт и опять приостановят свою работу.

Чтобы как-то оживить повествование замечу, что `module_close` не обладает монопольным правом на возобновление работы ожидающих процессов. Сигнал Ctrl-C (SIGINT) также может "разбудить" процесс. [12] В этом случае процессу немедленно возвращается `-EINTR`. Таким образом пользователи могут, например, прервать процесс прежде, чем он получит доступ к файлу.

Тут есть еще один момент, о котором хотелось бы упомянуть. Некоторые процессы не желают быть заблокированными, такие процессы должны либо получить в свое распоряжение открытый файл немедленно, либо извещение о том, что их запрос не может быть удовлетворен в настоящий момент. Такие процессы используют флаг `O_NONBLOCK` при открытии файла. Если ядро не в состоянии немедленно удовлетворить запрос, оно отвечает кодом ошибки `-EAGAIN`.

Пример 8-1. sleep.c

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

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

/*
 * Место хранения последнего принятого сообщения,
 * которое будет выводиться в файл, чтобы показать, что
 * модуль действительно может получать ввод от пользователя
 */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];

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

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

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

  /*
   * Для передачи данных из пространства ядра в пространство пользователя
   * следует использовать put_user.
   * В обратном направлении -- get_user.
```

```
   */
  sprintf(message, "Last input:%s\n", Message);
  for (i = 0; i < len && message[i]; i++)
    put_user(message[i], buf + i);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  /*
   * Возобновить работу процессов из WaitQ.
   */
  wake_up(&WaitQ);

  module_put(THIS_MODULE);

  return 0;
}

/*
 * Эта функция принимает решение о праве на выполнение операций с файлом
 * 0 -- разрешено, ненулеое значение -- запрещено.
 *
 * Операции с файлом могут быть:
 * 0 - Исполнениеe (не имеет смысла в нашей ситуации)
 * 2 - Запись (передача от пользователя к модулю ядра)
 * 4 - Чтение (передача от модуля ядра к пользователю)
 *
 * Эта функция проверяет права доступа к файлу
 * Права, выводимые командой ls -l
 * могут быть проигнорированы здесь.
 */
static int module_permission(struct inode *inode,
   int op, struct nameidata *nd)
{
  /*
   * Позволим любому читать файл, но
   * писать -- только root-у (uid 0)
   */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;

  /*
   * Если что-то иное -- запретить доступ
   */
  return -EACCES;
}

/*
 * Указатели на функции-обработчики для нашего файла.
 */
static struct file_operations File_Ops_4_Our_Proc_File = {
  .read = module_output,    /* чтение из файла */
  .write = module_input,    /* запись в файл */
```

```
  .open = module_open,      /* открытие файла */
  .release = module_close,  /* закрытие файла */
};


/*
 * Операции над индексной записью нашего файла. Необходима
 * для того, чтобы указать местоположение структуры
 * file_operations нашего файла, а так же, чтобы задать
 * функцию определения прав доступа к файлу. Здесь можно указать адреса
 * других функций-обработчиков, но нас они не интересуют.
 */
static struct inode_operations Inode_Ops_4_Our_Proc_File = {
  .permission = module_permission,      /* check for permissions */
};

/*
 * Начальная и конечная функции модуля
 */

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

int init_module()
{
  int rv = 0;
  Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);
  Our_Proc_File->owner = THIS_MODULE;
  Our_Proc_File->proc_iops = &Inode_Ops_4_Our_Proc_File;
  Our_Proc_File->proc_fops = &File_Ops_4_Our_Proc_File;
  Our_Proc_File->mode = S_IFREG | S_IRUGO | S_IWUSR;
  Our_Proc_File->uid = 0;
  Our_Proc_File->gid = 0;
  Our_Proc_File->size = 80;

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

  return rv;
}

/*
 * Завершение работы модуля - дерегистрация файла в /proc.
 * Чревато последствиями
 * если в WaitQ остаются процессы, ожидающие своей очереди,
 * поскольку точка их исполнения
 * практически находится в функции open, которая
 * будет выгружена при удалении модуля.
 * Позднее, в 9 главе, я опишу как воспрепятствовать
 * удалению модуля в таких случаях
 */
void cleanup_module()
{
  remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
}
```

Глава 9. Замена printk

9.1. Замена printk

Ранее я уже говорил о том, что XWindow и разработка модулей ядра есть вещи несовместимые. Это все так, но иногда возникает необходимость выдачи сообщений от модуля на любой tty. [13]

В качестве одного из вариантов можно предложить следующее: используя указатель на текущий исполняемый процесс -- current, получить структуру tty. Затем извлечь из этой структуры указатель на функцию вывода строки и использовать ее для выдачи сообщений.

Пример 9-1. print_string.c

```
/*
 *  print_string.c - отправляет вывод на
 *  tty терминала, независимо от того
 *  X11 это, или telnet, или что-то еще.
 *  Делается это путем вывода строки на tty,
 *  ассоциированный с текущим процессом.
 */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/sched.h>   /* определение current */
#include <linux/tty.h>     /* определение tty */
#include <linux/version.h> /* макрос LINUX_VERSION_CODE */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

static void print_string(char *str)
{
  struct tty_struct *my_tty;

  /*
   * Начиная с версии 2.6.6, структура tty перекочевала в структуру signal
   */
#if ( LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,5) )
  /*
   * tty текущего процесса
   */
  my_tty = current->tty;
#else
  /*
   * tty текущего процесса, для ядер 2.6.6 и выше
   */
  my_tty = current->signal->tty;
#endif

  /*
   * если my_tty == NULL, то текущий процесс не имеет tty на который можно
   * было бы что нибудь вывести (например, демон).
   * В этом случае нам ничего не нужно делать.
   */
  if (my_tty != NULL) {

    /*
     * my_tty->driver -- структура, которая
     * хранит указатели на функции-обработчики,
     * одна из которых (write) используется
     * для вывода строк на tty.
     *
     * Первый параметр функции -- tty, на
     * который осуществляется вывод,
     * поскольку эта функция обычно
     * используется для вывода на все
     * tty одного и того же типа.
     * Второй параметр -- флаг расположения строки
     * если строка находится в пространстве ядра,
     * флаг равен false (0)
     * если в пространстве пользователя, то true (не ноль).
     * Третий параметр -- указатель на строку.
     * Четвертый параметр -- длина строки.
     */
    ((my_tty->driver)->write) (my_tty,  /* Собственно tty  */
            0,                          /* Строка в пространстве ядра */
            str,                        /* Сама строка */
            strlen(str));               /* Длина строки */

    /*
```

```
     * tty изначально был аппаратным устройством, который (обычно)
     * ограничивался стандартом ASCII, в котором перевод строки
     * включал в себя два символа -- "возврат каретки" и "перевод строки".
     * В Unix, символ ASCII -- "перевод строки" заменял оба этих символа,
     * поэтому нам придется использовать для перевода строки
     * оба символа.
     *
     * Это одна из причин различий между текстовыми файлами Unix и
     * MS Windows. CP/M и ее "наследницы", например MS-DOS и
     * MS Windows, строго придерживались стандарта ASCII.
     */
    ((my_tty->driver)->write) (my_tty, 0, "\015\012", 2);
  }
}

static int __init print_string_init(void)
{
  print_string("The module has been inserted.  Hello world!");
  return 0;
}

static void __exit print_string_exit(void)
{
  print_string("The module has been removed.  Farewell world!");
}

module_init(print_string_init);
module_exit(print_string_exit);
```

9.2. Управление индикаторами на клавиатуре

При определенных условиях у вас может возникнуть желание дать вашему модулю более простой и более прямолинейный способ взаимодействия с внешним миром. Изменение состояния светодиодных индикаторов клавиатуры может быть одним из вариантов привлечения внимания пользователя или отображения некоторого состояния. Светодиодные индикаторы присутствуют на любой клавиатуре, они всегда находятся в поле зрения, они не нуждаются в установке, и их "подмаргивание" достаточно ненавязчиво, по сравнению с выводом на tty или в файл.

Следующий исходный код иллюстрирует модуль ядра, который после загрузки начинает мигать индикаторами клавиатуры.

Пример 9-2. kbleds.c

```
/*
 *  kbleds.c - Мигание индикаторами на клавиатуре.
 */

#include <linux/module.h>
#include <linux/config.h>
#include <linux/init.h>
#include <linux/tty.h>    /* определение fg_console, MAX_NR_CONSOLES */
#include <linux/kd.h>     /* определение KDSETLED */
#include <linux/console_struct.h> /* определение  vc_cons */

MODULE_DESCRIPTION("Пример module illustrating the use of Keyboard LEDs.");
MODULE_AUTHOR("Daniele Paolo Scarpazza");
MODULE_LICENSE("GPL");

struct timer_list my_timer;
struct tty_driver *my_driver;
char kbledstatus = 0;

#define BLINK_DELAY   HZ/5
#define ALL_LEDS_ON   0x07
#define RESTORE_LEDS  0xFF

/*
 * Функция my_timer_func мигает индикаторами
 * на клавиатуре периодически вызывая
 * ioctl() драйвера клавиатуры с командой KDSETLED.
 * Дополнительную информацию,
 * по командам ioctl виртуального терминала, вы найдете в:
 * /usr/src/linux/drivers/char/vt_ioctl.c, function vt_ioctl().
 *
 * Дополнительный аргумент команды KDSETLED -- значение 7
 * (перевод в режим LED_SHOW_IOCTL -- управление
 * индикаторами через ioctl), значение 0xFF --
 * (любое значение, большее 7, перевод в режим
 * LED_SHOW_FLAGS --
 * отображение фактического состояния клавиатуры).
 * Дополнительная информация:
 * /usr/src/linux/drivers/char/keyboard.c,
```

```
 * function setledstate().
 *
 */

static void my_timer_func(unsigned long ptr)
{
  int *pstatus = (int *)ptr;

  if (*pstatus == ALL_LEDS_ON)
    *pstatus = RESTORE_LEDS;
  else
    *pstatus = ALL_LEDS_ON;

  (my_driver->ioctl) (vc_cons[fg_console].d->vc_tty, NULL, KDSETLED,
         *pstatus);

  my_timer.expires = jiffies + BLINK_DELAY;
  add_timer(&my_timer);
}

static int __init kbleds_init(void)
{
  int i;

  printk(KERN_INFO "kbleds: loading\n");
  printk(KERN_INFO "kbleds: fgconsole is %x\n", fg_console);
  for (i = 0; i < MAX_NR_CONSOLES; i++) {
    if (!vc_cons[i].d)
      break;
    printk(KERN_INFO "poet_atkm: console[%i/%i] #%i, tty %lx\n", i,
           MAX_NR_CONSOLES, vc_cons[i].d->vc_num,
           (unsigned long)vc_cons[i].d->vc_tty);
  }
  printk(KERN_INFO "kbleds: finished scanning consoles\n");

  my_driver = vc_cons[fg_console].d->vc_tty->driver;
  printk(KERN_INFO "kbleds: tty driver magic %x\n", my_driver->magic);

  /*
   * Инициировать таймер
   */
  init_timer(&my_timer);
  my_timer.function = my_timer_func;
  my_timer.data = (unsigned long)&kbledstatus;
  my_timer.expires = jiffies + BLINK_DELAY;
  add_timer(&my_timer);

  return 0;
}

static void __exit kbleds_cleanup(void)
{
  printk(KERN_INFO "kbleds: unloading...\n");
  del_timer(&my_timer);
  (my_driver->ioctl) (vc_cons[fg_console].d->vc_tty, NULL, KDSETLED,
         RESTORE_LEDS);
}

module_init(kbleds_init);
module_exit(kbleds_cleanup);
```

Если ни один из примеров данной главы вас не устраивает, можно попробовать другие хитрости, скрытые в ядре. Может быть вам подойдет опция CONFIG_LL_DEBUG в **make menuconfig**? Включив ее, вы получите низкоуровневый доступ к последовательному порту. Как бы страшно это ни прозвучало, но можете попробовать изменить реализацию `kernel/printk.c` или какого нибудь другого системного вызова для вывода ascii-строк, чтобы иметь возможность отслеживать действия вашего модуля через последовательную линию связи.

Несмотря на то, что вы уже встретили в этой книге намало наглядных приемов отладки, существует еще ряд моментов, которые вам необходимо знать. Отладка -- это всегда очень утомительный процесс и практически всегда он сопровождается внедрением значительного количества отладочного кода. Может сложиться так, что отладочный код не дает проявляться некоторым ошибкам. Поэтому, при выпуске вашего модуля, старайтесь свести отладочный код к минимуму и "прогнать" модуль еще раз, пытаясь обнаружить какие либо ошибки.

Глава 10. Планирование задач

Очень часто возникает необходимость запуска вспомогательных задач по расписанию. Если запускаемая задача -- обычный процесс, то помещаем ее в файл crontab. Если же задача является модулем ядра, то у нас есть две возможности. Первая состоит в том, чтобы поместить некую задачу в файл crontab, которая будет "будить" модуль системным вызовом в заданный момент времени, например, открывая файл. Это очень неэффективно, т.к. при запуске нового процесса из crontab приходится загружать программу в память и всем это только для того, чтобы "разбудить" модуль ядра, который уже находится в памяти.

Вместо этого мы попробуем создать функцию, которая будет вызываться по прерываниям от таймера. Для этого, создадим задачу `struct work_struct`. Эта структура будет хранить указатель на функцию, срабатывающую по таймеру. Затем, с помощью `queue_delayed_work`, поместим задачу в очередь `tq_timer`, где должны располагаться задачи, срабатывающие по таймеру. А так как предполагается срабатывание функции каждый раз, по истечении заданного интервала времени, мы должны всякий раз опять вставлять ее в очередь `tq_timer`.

И еще один немаловажный момент. Когда модуль выгружается командой **rmmod**, сначала проверяется счетчик обращений к модулю. Если он равен нулю, то вызывается `module_cleanup`. После чего модуль удаляется из памяти со всеми его функциями. И никто не проверяет -- содержит ли очередь задач таймера указатель на одну из удаляемых функций. По прошествии некоторого времени (с точки зрения человека -- практически мгновенно), ядро получит прерывание от таймера и попробует вызывать удаленную из очереди задачу. Но функции-то больше нет! В большинстве случаев страница памяти, где она была, будет рассматриваться как неиспользуемая и вы получите сообщение об ошибке. Но может случиться так, что на этом месте окажется некоторый другой код и тогда ваше дело -- табак. К сожалению, у нас нет достаточно простого способа удаления задачи из очереди таймера.

Так как `cleanup_module` не может вернуть код ошибки (она не имеет возвращаемого значения), то напрашивается решение -- приостановить процедуру завершения работы модуля. Вместо того, чтобы немедленно завершить работу функции `cleanup_module`, мы можем приостановить работу команды **rmmod**. Затем, установив глобальную переменную, сообщить функции, вызываемой по прерыванию таймера, чтобы она убрала себя из очереди (точнее -- чтобы она опять не вставляла себя в очередь). На ближайшем прерывании таймера, процесс **rmmod** будет "разбужен", когда функция удалит себя из очереди таймера и удаление модуля станет безопасным.

Пример 10-1. sched.c

```
/*
 *  sched.c - реализация срабатывания по таймеру.
 *
 *  Copyright (C) 2001 by Peter Jay Salzman
 */

/*
 * Необходимые заголовочные файлы
 */

/*
 * Обычные, для модулей ядра
 */
#include <linux/kernel.h>    /* Все-таки мы работаем с ядром! */
#include <linux/module.h>    /* Необходимо для любого модуля */
#include <linux/proc_fs.h>   /* Необходимо для работы с /proc */
#include <linux/workqueue.h> /* очереди задач */
#include <linux/sched.h>     /* Взаимодействие с планировщиком */
#include <linux/init.h>      /* макросы  __init и  __exit */
#include <linux/interrupt.h> /* определение irqreturn_t */

struct proc_dir_entry *Our_Proc_File;
#define PROC_ENTRY_FILENAME "sched"
#define MY_WORK_QUEUE_NAME "WQsched.c"

/*
 * Счетчик срабатываний по таймеру
 */
static int TimerIntrpt = 0;

static void intrpt_routine(void *);

static int die = 0;  /* 1 -- завершить работу */

/*
 * очередь задач, создается для того,
 * чтобы поместить в очередь таймера (workqueue.h)
 */
static struct workqueue_struct *my_workqueue;

static struct work_struct Task;
static DECLARE_WORK(Task, intrpt_routine, NULL);

/*
 * Функция-обработчик прерывания от таймера.
 * Обратите внимание на аргумент типа void*
 * функция может получать дополнительные
 * аргументы посредством этого указателя.
 */
static void intrpt_routine(void *irrelevant)
{
  /*
   * Нарастить счетчик
   */
```

```
  TimerIntrpt++;

  /*
   * Если признак завершения сброшен,
   * то опять вставить себя в очередь таймера
   */
  if (die == 0)
    queue_delayed_work(my_workqueue, &Task, 100);
}

/*
 * Запись данных в файл /proc.
 */
ssize_t
procfile_read(char *buffer,
        char **buffer_location,
        off_t offset, int buffer_length, int *eof, void *data)
{
  int len;              /* Фактическое число записанных байт */

  /*
   * Переменные объявлены как static, поэтому они располагаются не на стеке
   * функции, а в памяти модуля
   */
  static char my_buffer[80];

  static int count = 1;

  /*
   * Все сведения выдаются за один присест,
   * поэтому, если смещение != 0, то значит
   * нам нечего больше сказать, поэтому возвращается
   * 0, в качестве признака конца файла.
   */
  if (offset > 0)
    return 0;

  /*
   * Заполнить буфер и получить его длину
   */
  len = sprintf(my_buffer, "Timer called %d times so far\n", TimerIntrpt);
  count++;

  /*
   * Указать адрес буфера
   */
  *buffer_location = my_buffer;

  /*
   * Вернуть длину буфера
   */
  return len;
}

/*
 * Функция инициализации - зарегистрировать файл в /proc
 */
int __init init_module()
{
  int rv = 0;
  /*
   * Создать очередь задач с нашей задачей и поместить ее в очередь таймера
   */
  my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);
  queue_delayed_work(my_workqueue, &Task, 100);

  Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);
  Our_Proc_File->read_proc = procfile_read;
  Our_Proc_File->owner = THIS_MODULE;
  Our_Proc_File->mode = S_IFREG | S_IRUGO;
  Our_Proc_File->uid = 0;
  Our_Proc_File->gid = 0;
  Our_Proc_File->size = 80;

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

  return rv;
}

/*
```

```
 * Завершение работы
 */
void __exit cleanup_module()
{
  /*
   * Удалить файл из /proc
   */
  remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
  printk(KERN_INFO "/proc/%s removed\n", PROC_ENTRY_FILENAME);
  /* Известить функцию обработки прерываний о завершении работы */
  die = 1;
  cancel_delayed_work(&Task);
  flush_workqueue(my_workqueue); /* ждать пока отработает таймер */
  destroy_workqueue(my_workqueue);

  /*
   * Приостановить работу, пока intrpt_routine не
   * отработает в последний раз.
   * Это необходимо, поскольку мы освобождаем память,
   * занимаемую этой функцией.
   */

}

/*
 * некоторые функции, относящиеся к work_queue
 * доступны только если модуль лицензирован под GPL
 */
MODULE_LICENSE("GPL");
```

Глава 11. Обработка прерываний

11.1. Обработка прерываний

Везде, кроме предыдущей главы, все наши действия в ядре сводилось к ответам на разные запросы от процессов, к работе со специальными файлом, посылке команд ioctl или запуску системных вызовов. Однако работа ядра не может сводится только к обработке запросов. Еще одна немаловажная задача -- это работа с аппаратурой компьютера.

Существует два типа взаимодействий между CPU и остальной аппаратной частью компьютера. Первый -- передача команд аппаратным средствам, второй -- прием ответов от аппаратуры. Второй тип взаимодействия -- прерывания, является наиболее тяжелым в обработке, потому что прерывания возникают тогда, когда это удобно устройству, а не CPU. Аппаратные устройства обычно имеют весьма ограниченный объем ОЗУ, и если не считать поставляемую ими информацию немедленно, то она может потеряться.

В Linux аппаратные прерывания называются **IRQ** (сокращенно от Interrupt ReQuests -- Запросы на Прерывание). [14] Имеется два типа IRQ: "короткие" и "длинные". "Короткие" IRQ занимают очень короткий период времени, в течение которого работа операционной системы будет заблокирована, а так же будет невозможна обработка других прерываний. "Длинные" IRQ могут занять довольно продолжительное время, в течение которого могут обрабатываться и другие прерывания (но не прерывания из того же самого устройства). Поэтому, иногда бывает благоразумным разбить выполнение работы на исполняемую внутри обработчика прерываний (т.е. подтверждение прерывания, изменение состояния и пр.) и работу, которая может быть отложена на некоторое время (например постобработка данных, активизация процессов, ожидающих эти данные и т.п.). Если это возможно, лучше объявлять обработчики прерывания "длинными".

Когда CPU получает прерывание, он останавливает любые процессы (если это не более приоритетное прерывание, тогда обработка пришедшего прерывания произойдет только тогда, когда более приоритетное будет завершено), сохраняет некоторые параметры в стеке и вызывает обработчик прерывания. Это означает, что не все действия допустимы внутри обработчика прерывания, потому что система находится в неизвестном состоянии. Решение проблемы: обработчик прерывания определяет -- что должно быть сделано немедленно (обычно что-то прочитать из устройства или что-то послать ему), а затем запланировать обработку поступившей информации на более позднее время (это называется "bottom halves" -- "нижние половины") и вернуть управление. Ядро гарантирует вызов "нижней половины" так быстро, насколько это возможно. Когда это произойдет, то наш обработчик -- "нижняя половина", уже не будет стеснен какими-то рамками и ему будет доступно все то, что доступно обычным модулям ядра.

Устанавливается обработчик прерывания вызовом `request_irq`. Ей передаются номер IRQ, имя функции-обработчика, флаги, имя для `/proc/interrupts` и дополнительный параметр для обработчика прерываний. Флаги могут включать `SA_SHIRQ`, чтобы указать, что прерывание может обслуживаться несколькими обработчиками (обычно, по той простой причине, что на одном IRQ может "сидеть" несколько устройств) и `SA_INTERRUPT`, чтобы указать, что это "короткое" прерывание. Эта функция установит обработчик только в том случае, если на заданном IRQ еще нет обработчика прерывания, или если существующий обработчик зарегистрировал совместную обработку прерывания флагом `SA_SHIRQ`.

Во время обработки прерывания, из функции-обработчика прерывания, мы можем получить данные от устройства и затем, с помощью `queue_task_irq`, `tq_immediate` и `mark_bh(BH_IMMEDIATE)`, запланировать "нижнюю половину". В ранних версиях Linux имелся массив только из 32 "нижних половин", теперь же, одна из них (а именно BH_IMMEDIATE) используется для обслуживания целого списка "нижних половин" драйверов. Вызов `mark_bh(BH_IMMEDIATE)` как раз и вставляет "нижнюю половину" драйвера в этот список, планируя таким образом ее исполнение.

11.2. Клавиатура на архитектуре Intel

Материал, рассматриваемый в оставшейся части этой главы, может быть применим исключительно к архитектуре Intel. На других платформах код примера работать не будет.

Было очень трудно выбрать тип драйвера, который можно было бы использовать в качестве примера в этой главе. С одной стороны пример должен быть достаточно полезным, он должен работать на любом компьютере и быть достаточно выразительным. С другой стороны, в ядро уже включено огромное количество драйверов практически для всех общеизвестных и широкораспространенных устройств. Эти драйверы не смогли бы совместно работать с тем, что я собирался написать. Наконец я принял решение представить в качестве примера -- обработчик прерываний от клавиатуры, но для демонстрации работоспособности кода сначала придется отключить стандартный обработчик прерываний от клавиатуры, а так как этот символ объявлен как `static` (в файле `drivers/char/keyboard.c`), то нет никакого способа восстановить обработчик. Поэтому, прежде чем вы дадите команду **insmod**, перейдите в другую консоль и дайте команду **sleep 120; reboot**, если ваша файловая система представляет для вас хоть какую-нибудь ценность.

Этот пример захватывает обработку IRQ 1 -- прерывание от клавиатуры на архитектуре Intel. При получении прерывания обработчик читает состояние клавиатуры (inb(0x64)) и скан-код нажатой клавиши. Затем, как только ядро сочтет возможным, оно вызывает `got_char` (она играет роль "нижней половины"), которая выводит, через `printk`, код клавиши (младшие семь бит скан-кода) и признак "нажата/отпущена" (8-й бит скан-кода -- 0 или 1 соответственно).

Пример 11-1. intrpt.c

```
/*
 *  intrpt.c - Обработчик прерываний.
 *
 *  Copyright (C) 2001 by Peter Jay Salzman
 */


/*
 * Standard in kernel modules
 */
#include <linux/kernel.h>    /* Все-таки мы работаем с ядром! */
#include <linux/module.h>    /* Необходимо для любого модуля */
#include <linux/workqueue.h> /* очереди задач */
#include <linux/sched.h>     /* Взаимодействие с планировщиком */
#include <linux/interrupt.h> /* определение irqreturn_t */
#include <asm/io.h>

#define MY_WORK_QUEUE_NAME "WQsched.c"
```

```
static struct workqueue_struct *my_workqueue;

/*
 * Эта функция вызывается ядром, поэтому в ней будут безопасны все действия
 * которые допустимы в модулях ядра.
 */
static void got_char(void *scancode)
{
  printk("Scan Code %x %s.\n",
         (int)*((char *)scancode) & 0x7F,
         *((char *)scancode) & 0x80 ? "Released" : "Pressed");
}

/*
 * Обработчик прерываний от клавиатуры.
 * Он считывает информацию с клавиатуры
 * и передает ее менее критичной по
 * времени исполнения части,
 * которая будет запущена сразу же,
 * как только ядро сочтет это возможным.
 */

irqreturn_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
  /*
   * Эти переменные объявлены статическими, чтобы имелась возможность
   * доступа к ним (посредством указателей) из "нижней половины".
   */
  static int initialised = 0;
  static unsigned char scancode;
  static struct work_struct task;
  unsigned char status;

  /*
   * Прочитать состояние клавиатуры
   */
  status = inb(0x64);
  scancode = inb(0x60);

  if (initialised == 0) {
    INIT_WORK(&task, got_char, &scancode);
    initialised = 1;
  } else {
    PREPARE_WORK(&task, got_char, &scancode);
  }

  queue_work(my_workqueue, &task);

  return IRQ_HANDLED;
}

/*
 * Инициализация модуля - регистрация обработчика прерывания
 */
int init_module()
{
  my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);

  /*
   * Поскольку стандартный обработчик прерываний от клавиатуры не может
   * сосуществовать с таким как наш, то придется запретить его
   * (освободить IRQ) прежде, чем что либо сделать.
   * Но поскольку мы не знаем где он находится в ядре, то мы лишены
   * возможности переустановить его - поэтому
   * компьютер придется перезагрузить
   * после опробования этого примера.
   */
        free_irq(1, NULL);

  /*
   * Подставить свой обработчик (irq_handler) на IRQ 1.
   * SA_SHIRQ означает, что мы допускаем возможность совместного
   * обслуживания этого IRQ другими обработчиками.
   */
  return request_irq(1,     /* Номер IRQ */
         irq_handler,       /* наш обработчик */
         SA_SHIRQ,
         "test_keyboard_irq_handler",
         (void *)(irq_handler));
}

/*
 * Завершение работы
 */
```

```
void cleanup_module()
{
  /*
   * Эта функция добавлена лишь для полноты изложения.
   * Она вообще бессмысленна, поскольку я не вижу способа
   * восстановить стандартный обработчик прерываний от клавиатуры
   * поэтому необходимо выполнить перезагрузку системы.
   */
  free_irq(1, NULL);
}

/*
 * некоторые функции, относящиеся к work_queue
 * доступны только если модуль лицензирован под GPL
 */
MODULE_LICENSE("GPL");
```

Глава 12. Симметричная многопроцессорность

Один из самых простых и самых дешевых способов увеличения производительности -- это разместить на материнской плате несколько микропроцессоров. Каждый из процессоров может играть свою собственную роль (асимметричная многопроцессорная обработка -- ASMP) или же они все работают параллельно, организуя вычисления таким образом, при котором и операционная система, и приложения могут использовать любой доступный процессор (симметричная многопроцессорная обработка -- SMP). Примером асимметричной многопроцессорной ОС может служить NetWare SFT III, использующая зеркальное отображение серверов, в двухпроцессорном сервере первый процессор занимается предоставлением услуг, а второй - операциями ввода/вывода. Асимметричная многопроцессорноя обработка более эффективна, но она требует точного распределения ролей между процессорами, что практически невозможно на универсальных ОС, подобных Linux. С другой стороны, симметричная многопроцессорная обработка менее эффективна, но относительно проста в реализации (здесь, под словами "относительно проста", вовсе не подразумевается, что это *действительно* просто).

В симметричной многопроцессорной среде микропроцессоры совместно используют одну и ту же память, в результате код, работающий на одном CPU может изменять содержимое памяти, используемой другим CPU. Здесь уже нет уверенности, что переменная, которой вы присвоили некоторое значение, в предыдущей строке программы, все еще имеет то же самое значение -- код, исполняемый на другом CPU может его поменять. Очевидно, что я привел невероятный пример.

На самом деле, в случае обычных процессов, такой проблемы не существует. Как правило, в каждый конкретный момент времени, процесс может исполняться только на одном CPU. [15] Но ядро может выполнять одновременно разные процессы на разных CPU.

В ядрах версии 2.0.x это не было большой проблемой, поскольку ядро работало под защитой одной большой спин-блокировки (spinlock). Это означает, что когда один процессор работает в привилегированном режиме, а другой собирается войти в этот режим (например в случае системного вызова), то он вынужден ждать, пока первый процессор не выйдет в пользовательский режим. Это делает SMP в Linux безопасной, но малоэффективной.

Начиная с ядра версии 2.2.x стал возможным одновременный выход нескольких процессоров в привилегированный режим. Это обстоятельство вы должны знать и помнить.

Глава 13. Заключение

В заключение я хотел бы предупредить вас о некоторых общих ошибках, допускаемых программистами модулей ядра. Если я что-то упустил из виду, и из-за этого у вас произошло нечто ужасное -- напишите мне и я верну вам ту долю авторского гонорара, которую я получил за ваш экземпляр книги.

Стандартные библиотеки. Вы не должны использовать функции из стандартных библиотек языка C. Используйте только те функции, которые предоставляются ядром (большинство из них вы найдете в `/proc/kallsyms`).

Запрет прерываний. Если вы запретите прерывания на короткий срок, то ничего страшного не произойдет. Но если вы забудете их разрешить, то система "зависнет" и перезагрузить ее можно будет только кнопкой выключения питания.

Не суйте голову в пасть тигру! Вероятно это предупреждение излишне, но тем не менее...

5 Примечания

[1] В ранних версиях ядра он назывался **kerneld**.

[2] Если вы предполагаете проводить эксперименты с ядром, то чтобы избежать перезаписи существующих файлов модулей вы можете изменить значение переменной `EXTRAVERSION` в `Makefile` ядра. В результате все модули ядра будут записываться в отдельный каталог.

[3] Я не компьютерный гений, я простой физик!

[4] Это - далеко не то же самое, что и "встраивание всех модулей в ядро", хотя идея та же.

[5] Делается это в соответствии с принятыми соглашениями. Однако, при разработке драйвера устройства, на период отладки, размещать файл устройства в своем домашнем каталоге -- наверное не такая уж и плохая идея. Единственное -- не забудьте исправить место для размещения файла устройства после того, как отладка будет закончена.

[6] В версиях ядра 2.0 и 2.2 это делалось автоматически, если в качестве номера inode передавалось нулевое значение.

[7] Различие здесь состоит в том, что файловые операции работают с файлом непосредственно, а функции, работающие с inode, предоставляют способ сослаться на файл, например создание ссылок на файл.

[8] Обратите внимание, здесь смысл терминов "чтение" и "запись" также имеют обратный смысл. Так операция чтения передает данные от процесса ядру, а операция записи -- в обратном направлении, от ядра к процессу.

[9] Не совсем верно. Вы не сможете передать функции `ioctl`, например структуру, но передать указатель на структуру -- безусловно возможно.

[10] Самый простой способ оставить файл открытым -- это дать команду **tail -f**

[11] Это означает, что процесс продолжает свою работу в привилегированном режиме -- поскольку процесс был приостановлен во время работы системного вызова, который еще не закончил свою работу. Процесс даже не "подозревает" о том, что кто-то еще, кроме него, использовал процессор в промежутке между обращением к системному вызову и возвратом из него.

[12] Именно по этой причине использовался вызов функции `wait_event_interruptible`. Можно было бы использовать `wait_event`, но тогда задачу невозможно будет прервать по Ctrl-C во время ожидания.

[13] Tty (от англ. TeleTYpe - телетайп) -- первоначально обозначал комбинацию клавиатура-печатающее устройство предназначенное для взаимодействия с Unix-системой, на сегодняшний день -- это абстракция текстового потока, используемого программами Unix, независимо от того, является ли он физическим терминалом, окном xterm на дисплее, сетевым подключением или чем то иным.

[14] Это стандартное понятие для архитектуры Intel, на которой начинал разрабатываться Linux.

[15] За исключением многопоточных процессов. В этом случае разные потоки одного процесса могут исполняться одновременно на разных процессорах.