

Модуль в иерархии программных систем (02-30)

Все мы умеем писать программы для Linux и имеем более или менее приличный опыт написания таких программ, которые при всем их многообразии имеют абсолютно идентичную единую структуру:

```
int main(int argc, char *argv[]) {  
    // и здесь далее следует любой программный код, вплоть до вот  
    такого:  
    printf("Hello, world!\n");  
    // ... и далее, далее, далее ...  
    exit(EXIT_SUCCESS);  
}
```

Такую структуру в коде будут неизменно иметь все приложения-программы, будь то тривиальная программа «Hello, world!», показанная только что, или самая навороченная среда разработки IDE или CAD. Это встречающийся в подавляющем большинстве случаев: пользовательское приложение, начинающееся с некоторого *main()*, и завершающее выполнение по *exit()*. Мы говорим о языке программирования C, но примерно то же самое будет и в сотне других используемых языков программирования.

Еще один встречающийся (но гораздо реже) в UNIX случай — демоны: программы, стартующие с *main()*, но никогда не завершающие своей работы. Чаще всего они представляют собой серверы различных служб. Так, в Linux — это сервисы, находящиеся под управлением подсистемы *systemd*. В этом случае для того, чтобы стать сервером-сервисом, все тот же пользовательский процесс должен выполнить некоторую фиксированную последовательность действий, называемую демонизацией.

Но даже тогда процесс выполняется в пользовательском адресном пространстве (отдельном для каждого процесса) со всеми ограничениями пользовательского режима: запрет на использование привилегированных команд, невозможность

обработки прерываний, запрет (без особых ухищрений) операций ввода/вывода и многих других тонких деталей.

Возникает вопрос: а может ли пользователь (потребитель) написать и выполнить собственный код, выполняющийся в режиме супервизора, а значит, имеющий все полномочия расширять (или даже изменять) функциональность ядра Linux?

Да, может! И эта техника программирования называется ***программированием модулей ядра*** (**программирование в ядре**). И именно она позволяет, в частности, создавать драйверы любого нестандартного оборудования.

Запуск модуля (называемый загрузкой) выполняется посредством специальных команд установки (*insmod*, *modprobe* ... и *remmod* для удаления). Вот, например, команда:

```
# insmod <имя-файла-модуля>.ko
```

После выполнения такой команды в модуле начинает выполняться функция инициализации, и модуль включается в состав ядра Linux.

Что же конкретно такое модуль ядра?

Модули – это элементы кода, которые по необходимости можно загружать в ядро и выгружать. Они расширяют его функциональность, не требуя перезагрузки системы. К примеру, одним из типов модулей является драйвер устройств, который позволяет ядру обращаться к подключённому аппаратному обеспечению.

Не имея модулей, нам бы пришлось строить монолитные ядра и добавлять новую функциональность непосредственно в их образы. И мало того что это привело бы к увеличению размеров ядра, но ещё и вынудило бы нас пересобирать и перезагружать его при каждом добавлении новой функциональности.

Специфика программирования в ядре (03-12)

Программирование в ядре имеет определённые отличия от программирования пользовательских приложений и накладывает

определённые сложности и ограничения. И прежде чем переходить к конкретике программирования в ядре, разумно изучить те сложности, которые будут подстерегать и подготовиться к ним.

Для начала ответим на некоторые вопросы₍₀₂₋₂₉₎

Q: Можно ли писать и отрабатывать код модулей ядра, не имея в системе административных полномочий root?

A: Нет. Если вам недоступны права root в системе, вы не сможете отрабатывать код модулей ядра.

Q: Нужно ли для написания драйверов (модулей ядра) устанавливать в своей рабочей системе исходные коды ядра?

A: Нет. Для написания драйверов (модулей ядра) не нужно иметь в своей рабочей системе исходные коды ядра. Но нужно иметь заголовочные файлы (хедер-файлы, .h) ядра.

Q: Можно ли в программировании модулей ядра использовать какой-то другой язык программирования, кроме языка C?

A: Нет. Само ядро Linux написано на языке C, поэтому и модули ядра (являющиеся, по сути, плагинами к ядру) должны готовиться на языке C.

Q: Может ли в работе с модулями ядра использоваться компилятор, отличный от GCC?

A: В принципе, и само ядро, и модули к нему должны компилироваться компилятором GCC. Но есть сообщения, что ядро Linux (а значит, и модули ядра) успешно компилировались более новым компилятором Clang из проекта LLVM. В общем случае никакие другие компиляторы, кроме GCC, не должны использоваться для компиляции модулей ядра.

Q: Могут ли в Linux быть бинарные драйверы, «готовые» к инсталляции, независимо от версии ядра?

A: Нет, не могут. Драйверы, являющиеся модулями ядра, связываются с экспортируемыми именами ядра (вызываемыми функциями API или объектами данных) по их абсолютным

адресам, изменяющимися не только при изменении версии ядра, но даже при пересборке ядра с измененными конфигурационными параметрами. Поэтому драйверы Linux могут предоставляться только в виде исходных кодов на языке C, которые требуется компилировать для использования.

Необходимо помнить: (03-12)

1. В ядре недоступны никакие библиотеки, привычные из прикладного программирования, и известные как POSIX API. Как следствие, ядро оперирует со своим собственным набором API (kernel API), отличающимся от POSIX API (отличающихся набором функций, их наименованиями).

2. Как следствие этой автономности реализации API ядра является то, что одной из основных трудностей программирования модулей является **нахождение и выбор** адекватных средств API из набора плохо документированных и достаточно часто изменяющихся API ядра. Если по POSIX API существуют многочисленные обстоятельные справочники, то по именам ядра (вызовам и структурам данных) таких руководств нет. А общая размерность имён ядра (в /proc/kallsyms) приближается к 100000, из которых до 10000 — это экспортируемые имена ядра.

3. Одна из основных трудностей программирования модулей состоит в нахождении и выборе слабо документированных и изменяющихся API ядра. В этом нам значительную помощь оказывает динамические и статические таблицы разрешения имён ядра, и заголовочные файлы исходных кодов ядра, по которым мы должны постоянно сверяться на предмет актуальности ядерных API текущей версии используемого нами ядра.

По kernel API практически **отсутствует документация** и описания. Выяснять детали функционирования придётся по заголовочным файлам (хэдерам) и исходным кодам реализации (что особенно хлопотно). Некоторую ясность может внести изучение прототипов использования требуемых API, найденные контекстным поиском в Интернет. Особое внимание следует обратить на **комментарии** в хэдерах и исходном коде.

4. Ещё одной особенностью kernel API является их высокая волатильность от версии к версии ядра: вызов API, имеющий 3 параметра и успешно работающий в текущей версии, может получить 4 параметра в следующей версии, а код перестанет даже компилироваться. Это радикально отличает программирование ядра от POSIX API, где вызовы регламентированы стандартом, должны ему подчиняться и многократно описаны. Разработчики ядра, в отличие от POSIX, не связаны никакими соглашениями о совместимости сверху-вниз.

5. **Уникальными ресурсами**, позволяющими изучить и сравнить исходный код ядра для различных **версий** ядра и аппаратных **платформ**, являются ресурсы построенные на базе проекта LXR:

<http://lxr.free-electrons.com/source/> (Linux Cross Reference)

<http://lxr.linux.no/+trees> (the Linux Cross Reference)

<http://lxr.oss.org.cn/> (Linux Kernel Cross Reference)

Это основные источники, позволяющие сравнивать изменения в API и реализациях от версии к версии (начиная с самых ранних версий ядра). Часто изучение элементов кода ядра по этим ресурсам гораздо продуктивнее, чем то же, но по исходному коду непосредственно вашей инсталлированной системы. Вообще, как показывает практика, иметь в своей локальной системе исходный код ядра Linux собственного варианта (версия, платформа) и на него опираться при работе — *порочная практика*.

6. Следующей особенностью есть то, что при отладке кода ядра даже незначительная ошибка в коде может стать причиной полного краха ядра (например, элементарный вызов `strcpy()`). Вплоть до того (и это следует ожидать), что в некоторых случаях система перед гибелью даже не успевает дать предсмертное отладочное сообщение Ooops...

Отсутствие защиты памяти. Если обычная программа предпринимает попытку некорректного обращения к памяти, ядро аварийно завершит процесс, послав ему сигнал SIGSEGV. Если ядро предпримет попытку некорректного обращения к

памяти, результаты могут быть менее контролируемыми. К тому же ядро не использует замещение страниц: каждый байт, используемый в ядре – это один байт физической памяти.

7. Как следствие высокой вероятности краха системы даже в итоге незначительной ошибки кода является то, что **каждый** тестовый запуск может требовать перезагрузки системы, а это **в разы** замедляет темп проекта, в сравнении с ориентирами пользовательского пространства. Отличным решением (по крайней мере, когда не требуется какая-то аппаратная специфика) является решение вести разработку в среде **виртуальной машины** Linux, на начальном этапе проекта и до до 90% общего времени разработки. Для этого не очень пригодны развитые системы виртуализации типа XEN и подобные, но очень подходят достаточно простые гипервизоры Oracle VirtualBox, или QEMU и KVM (когда нужна архитектура отличная от x86).

8. Ещё один аспект применения виртуальных машин состоит в том, что разрабатываемый модуль может (должен) быть «прогнан» через ядра различных версий, для предоставления потребителю качественного результата и отсутствия рекламаций (невозможно предсказать в каком ядре потребитель станет собирать модуль). Провести такое тестирование на реальных инсталляциях практически невозможно. А иметь в виртуальном гипервизоре 10 различных виртуальных машин (и даже одновременно выполняющихся) — совершенно реально.

9. В ядре нельзя использовать вычисления с плавающей точкой. Активизация режима вычислений с плавающей точкой требует (при переключении контекста) сохранения и восстановления регистров устройства поддержки вычислений с плавающей точкой (FPU), помимо других рутинных операций. Вряд ли в модуле ядра могут понадобиться вещественные вычисления, но если такое и случится, то их нужно эмулировать через целочисленные вычисления (для этого существует множество библиотек, из которых может быть **заимствован** код).

10. Фиксированный стек – область адресного пространства, в которой выделяются локальные переменные. Локальные переменные – это все переменные, объявленные внутри блока,

открывающегося левой открывающей фигурной скобкой и не имеющие ключевого слова `static`. Стек в режиме ядра ограничен по размеру и не может быть изменён. Поэтому в коде ядра нужно крайне осмотрительно использовать (или не использовать) конструкции, расточающие пространство стека: рекурсию, передача параметром структуры, или возвращаемое значение из функции как структура, объявление крупных локальных структур внутри функций и подобных им. Обычно стек равен двум страницам памяти, что соответствует, например, 8 Кбайт для 32-бит систем и 16 Кбайт для 64-бит систем.

Создание среды разработки (03-14)

Первично установленная по умолчанию из пакетного дистрибутива система Linux для сборки модулей ядра **непригодна**. В ней отсутствуют некоторые специфические компоненты, такие как хэдеры ядра и другие подобные. Это и естественно, так как рядовому пользователю нет нужды собирать модули ядра, и незачем ему загружать файловую систему достаточно объёмными бесполезными данными. Но вам необходимо создать среду сборки, если вы не сделали этого ранее. Для этого нужно установить некоторые дополнительные пакеты из репозитория...

Начиная с того, что зачастую (во многих дистрибутивах) и сам компилятор GCC и утилита `make` могут отсутствовать в системе при дефолтной установке, что мы проверим так:

```
$ which gcc
$ which make
$
```

Нужно начать с установки этих основных средств программирования. Вот как это выглядит в дистрибутиве Debian (Ubuntu) Linux (и то, что пакеты потянут за собой по зависимостям):

```
# apt-get install gcc make
```

Чтение списков пакетов... Готово

Построение дерева зависимостей

Чтение информации о состоянии... Готово

Будут установлены следующие дополнительные пакеты:

binfmt-support binutils cpp-4.6 g++-4.6 gcc-4.6 gcc-4.6-base gcc-4.7 libc-dev-bin libc6-dev
libffi-dev libitm1 libllvm3.0 libstdc++6-4.6-dev linux-libc-dev

Предлагаемые пакеты:

binutils-doc gcc-4.6-locales g++-4.6-multilib gcc-4.6-doc libstdc++6-4.6-dbg gcc-multilib
autoconf automake1.9 libtool flex bison gdb gcc-doc gcc-4.6-multilib libmudflap0-4.6-dev
libgcc1-dbg libgomp1-dbg libquadmath0-dbg libmudflap0-dbg binutils-gold gcc-4.7-multilib
libmudflap0-4.7-dev gcc-4.7-doc gcc-4.7-locales libitm1-dbg libcloog-ppl0 libppl-c2 libppl7
glibc-doc libstdc++6-4.6-doc

...

\$ gcc --version

gcc (Debian 4.7.2-5) 4.7.2

...

\$ make --version

GNU Make 3.81

...

В RPM-дистрибутивах (Fedora, CentOS, RedHat, ...) установка этих же инструментов сборки делается командой:

yum install gcc make

...

Но сделанного нами мало — этого достаточно для прикладного программирования, но недостаточно для программирования модулей ядра. Нам нужно создать инфраструктуру для сборки модулей ядра (главным образом это заголовочные файлы ядра, но не только).

В Fedora, CentOS, RedHat, ... нам необходимы дополнительные пакеты `kernel-headers.*` (обычно устанавливается вместе с ядром) и `kernel-devel.*` :

\$ yum list all kernel*

...

0 packages excluded due to repository protections

Установленные пакеты

kernel.i686 3.12.7-300.fc20 @fedora-updates/\$releasever

kernel.i686 3.12.10-300.fc20 @updates

kernel-headers.i686 3.12.10-300.fc20 @updates

kernel-modules-extra.i686 3.12.7-300.fc20 @fedora-updates/\$releasever

kernel-modules-extra.i686 3.12.10-300.fc20 @updates

Доступные пакеты

kernel.i686 3.13.6-200.fc20 updates

...

kernel-devel.i686 3.13.6-200.fc20 updates

...

kernel-headers.i686 3.13.6-200.fc20 updates

kernel-modules-extra.i686 3.13.6-200.fc20 updates

...

Обращаем внимание на то, что пакет `kernel-devel.*` предоставляется в репозиториях только для последнего обновляемого ядра (а не для предыдущих установленных), то есть целесообразно начать с обновления ядра:

\$ sudo yum update kernel*

...

\$ sudo yum install kernel-devel.i686

...

Объем загрузки: 8.5 М

Объем изменений: 31 М

Is this ok [y/d/N]: y

...

Вот только после этого у вас создастся инфраструктура, для текущей версии ядра, для работы с модулями:

\$ ls /usr/lib/modules

3.12.10-300.fc20.i686 3.12.7-300.fc20.i686 3.13.6-200.fc20.i686

\$ tree -L 2 /usr/src/kernels/

/usr/src/kernels/

├── 3.13.6-200.fc20.i686

│ ├── arch

│ ├── block

│ ├── crypto

│ ├── drivers

│ ├── firmware

│ ├── fs

│ ├── include

│ ├── init

│ ├── ipc

│ └── Kconfig

- kernel
- lib
- Makefile
- mm
- Module.symvers
- net
- samples
- scripts
- security
- sound
- System.map
- tools
- usr
- virt
- vmlinux.id

21 directories, 5 files

В дистрибутивах Debian/Ubuntu картина, в общем, та же, только здесь вам необходима установка только одного пакета (linux-headers-* - выбранного для **вашей архитектуры ядра**) :

\$ cat /etc/debian_version

7.2

\$ aptitude show linux-headers

Нет в наличии или подходящей версии для linux-headers

Пакет: linux-headers

Состояние: не реальный пакет

Предоставляется: linux-headers-3.2.0-4-486, linux-headers-3.2.0-4-686-pae,

linux-headers-3.2.0-4-amd64, linux-headers-3.2.0-4-rt-686-pae, linux-headers-486,

linux-headers-686-pae, linux-headers-amd64, linux-headers-rt-686-pae

...

\$ sudo aptitude install linux-headers-3.2.0-4-486

...

Настраивается пакет linux-headers-3.2.0-4-486 (3.2.51-1) ...

\$ ls /lib/modules/3.2.0-4-486 -l

итого 3000

lrwxrwxrwx 1 root root 34 Сен 20 17:26 build -> /usr/src/linux-headers-3.2.0-4-486

drwxr-xr-x 9 root root 4096 Окт 13 19:38 kernel

-rw-r--r-- 1 root root 723786 Окт 13 19:51 modules.alias

-rw-r--r-- 1 root root 705194 Окт 13 19:51 modules.alias.bin

-rw-r--r-- 1 root root 2954 Сен 20 17:26 modules.builtin

-rw-r--r-- 1 root root 3960 Окт 13 19:51 modules.builtin.bin

-rw-r--r-- 1 root root 353853 Окт 13 19:51 modules.dep

-rw-r--r-- 1 root root 491462 Окт 13 19:51 modules.dep.bin

-rw-r--r-- 1 root root 325 Окт 13 19:51 modules.devname

-rw-r--r-- 1 root root 118244 Сен 20 17:26 modules.order

-rw-r--r-- 1 root root 131 Окт 13 19:51 modules.softdep

-rw-r--r-- 1 root root 286536 Окт 13 19:51 modules.symbols

-rw-r--r-- 1 root root 364265 Окт 13 19:51 modules.symbols.bin

lrwxrwxrwx 1 root root 37 Сен 20 17:26 source -> /usr/src/linux-headers-3.2.0-4-common

Во всех случаях смысл состоит в том, чтобы добиться, чтобы путь для текущего ядра /lib/modules/`uname -r`/build представлял не **пустую висящую ссылку**, а был иерархией заполненных каталогов для сборки модулей:

\$ ls -ld -w 100 /lib/modules/`uname -r`/build/*

/lib/modules/3.13.6-200.fc20.i686/build/arch

/lib/modules/3.13.6-200.fc20.i686/build/block

/lib/modules/3.13.6-200.fc20.i686/build/crypto

/lib/modules/3.13.6-200.fc20.i686/build/drivers

/lib/modules/3.13.6-200.fc20.i686/build/firmware

/lib/modules/3.13.6-200.fc20.i686/build/fs

/lib/modules/3.13.6-200.fc20.i686/build/include

/lib/modules/3.13.6-200.fc20.i686/build/init

/lib/modules/3.13.6-200.fc20.i686/build/ipc

/lib/modules/3.13.6-200.fc20.i686/build/Kconfig

/lib/modules/3.13.6-200.fc20.i686/build/kernel

/lib/modules/3.13.6-200.fc20.i686/build/lib

/lib/modules/3.13.6-200.fc20.i686/build/Makefile

/lib/modules/3.13.6-200.fc20.i686/build/mm

/lib/modules/3.13.6-200.fc20.i686/build/Module.symvers

```
/lib/modules/3.13.6-200.fc20.i686/build/net  
/lib/modules/3.13.6-200.fc20.i686/build/samples  
/lib/modules/3.13.6-200.fc20.i686/build/scripts  
/lib/modules/3.13.6-200.fc20.i686/build/security  
/lib/modules/3.13.6-200.fc20.i686/build/sound  
/lib/modules/3.13.6-200.fc20.i686/build/System.map  
/lib/modules/3.13.6-200.fc20.i686/build/tools  
/lib/modules/3.13.6-200.fc20.i686/build/usr  
/lib/modules/3.13.6-200.fc20.i686/build/virt  
/lib/modules/3.13.6-200.fc20.i686/build/vmlinux.id
```

Команды *lsmod*, *insmod*, *modprobe* (01-161)

Как модули попадают в ядро. Один из способов — это использование команды *insmod* вручную. Вы просто вводите эту команду, а в качестве аргумента указываете имя модуля, который вы хотите добавить в ядро, например:

```
# insmod /путь/модуль.o
```

Второй способ — это использование файла */etc/modules.conf* (в некоторых дистрибутивах — просто */etc/modules*). Вы прописываете в этом файле все модули ядра, необходимые вам постоянно, дабы не вводить команду *insmod* после каждой перезагрузки для каждого модуля.

В более современных дистрибутивах вместо файла */etc/modules.conf* используется файл */etc/modprobe.conf*, формат которого такой же, как у *modules.conf*. В листинге 13.1 приведен пример файла */etc/modprobe.conf*.

Листинг 13.1. Файл */etc/modprobe.conf*

```
alias eth0 pcnet32  
alias sound-slot-0 snd_ens1371  
install scsi_hostadapter /sbin/modprobe mptscsih; /sbin/modprobe mptspi;  
/sbin/modprobe ata_piix; /sbin/modprobe ahci; /bin/true  
install usb-interface /sbin/modprobe ehci_hcd; /sbin/modprobe uhci_hcd;  
/bin/true
```

Получить подробную информацию о формате этого файла можно в справочной системе:

man modprobe.conf

Команда *modprobe*, как правило, вызывается при загрузке системы из сценариев инициализации системы. Далее она читает свой файл конфигурации и загружает указанные в нем модули (или, наоборот, выгружает, если указана команда *remove*).

Команду *modprobe* можно использовать и для загрузки модулей. При этом ее удобнее использовать, даже чем *insmod*. Команда *insmod* ничего не знает о размещении модулей, и вам для загрузки модуля нужно указывать полный путь к нему вместе с "расширением" файла модуля. Команду *modprobe* использовать проще — нужно указать только имя модуля (без пути к файлу), например:

```
# modprobe ivtv
```

Данная команда найдет и загрузит модуль *ivtv*. Как видите, вам даже не нужно знать, в каком файле находится файл модуля.

Чуть ранее было сказано, что вместо файла */etc/modules.conf* используется файл */etc/modprobe.conf*. Это не совсем так. Модули, указанные в файле */etc/modules.conf* (или просто в */etc/modules*), загружаются при загрузке системы. А вот модули, указанные в */etc/modprobe.conf*, — при необходимости, например, когда ядру понадобилась поддержка того или иного устройства.

В системе очень много модулей, но это не означает, что все эти модули нужно перечислять в */etc/modprobe.conf*. При необходимости (даже если модуль не указан в */etc/modprobe.conf*) программа *modprobe* и так найдет и загрузит его. Файл */etc/modprobe.conf* используется для передачи параметров модулям ядра — иными словами, для конфигурации этих модулей. Например, вы можете передать модулю звуковой карты начальный уровень громкости.

Что же касается файла */etc/modules.conf*, то в современных дистрибутивах действительно есть его аналог — */etc/modprobe.preload*. В нем указываются модули, которые должны быть загружены при старте системы.

Кроме файлов */etc/modprobe.conf* и */etc/modprobe.preload* обычно имеются каталоги */etc/modprobe.d* и */etc/modprobe.preload.d*.

Зайдите в один из этих каталогов. В нем вы найдете несколько текстовых файлов, в каждом из которых — команды по управлению модулями. Все эти команды можно было бы прописать в файле */etc/modprobe.conf* или в */etc/modprobe.preload*. Но для большего удобства, особенно если команд много, можно их вынести в отдельный файл и поместить его в каталог */etc/modprobe.d* (или в */etc/modprobe.preload.d*, если модуль нужно загружать при старте системы).

Есть еще один способ — загрузка модулей из каталога */etc/sysconfig/modules*.

Итак, в каком же файле лучше прописывать модули? Учитывая, что время не стоит на месте, то лучше все-таки их указывать либо в */etc/modprobe.conf*, либо в */etc/modprobe.preload* (или создать отдельный файл в соответствующем каталоге).

Файлы */etc/modules.conf* и */etc/modules* могут даже не обрабатываться в некоторых дистрибутивах. Но в файловой системе будет один из этих файлов для обеспечения обратной совместимости.

В любом случае разработанные нами модули вы будете загружать с помощью команды *insmod* — она работает везде, а потом уже разберетесь с автоматической загрузкой модулей в вашем дистрибутиве. Команда *lsmod* позволяет вывести список загруженных модулей.

Данную команду нужно вводить от имени *root* и желательно перенаправить вывод на программу *less* — уж слишком много модулей будет в выводе *lsmod*:

```
# lsmod | less
```

Удалить загруженный модуль из ядра можно командой *rmmod*. Понятно, что команда удаляет модуль из ядра, но файл модуля никуда не денется с жесткого диска.

Окончательно: необходимо установить пакеты:

Прежде чем приступить к разработке собственного модуля, вам нужно установить дополнительные пакеты:

- ✓ ***make*** — содержит утилиту *make*, без которой просто невозможно выполнить сборку модуля ядра (точнее, возможно, но очень и очень неудобно);
- ✓ ***kernel-source*** — содержит исходные тексты ядра, пакет *kernel-source* называется одинаково во всех дистрибутивах;
- ✓ ***kernel-headers***, или *linux-userspace-headers* — заголовочные файлы ядра, название пакета может отличаться в зависимости от дистрибутива, произведите поиск по строке *headers* и прочитайте описание найденных пакетов — найти нужный не составит труда;
- ✓ ***kernel-default-devel*** — файлы, необходимые для разработки модулей ядра. В некоторых дистрибутивах этот пакет может называться иначе, например *kerneldevel* или *kernel-*-devel*.

Остальные необходимые пакеты, как правило, будут установлены автоматически при разрешении зависимостей.

Первый модуль (вариант 1Кол.) (01-164)

Наша задача — написать и скомпилировать первый модуль, точнее — болванку модуля. Наш модуль не будет выполнять каких-либо полезных действий, но вы сможете его загрузить в ядро и выгрузить из ядра командой *rmmod*.

Написание "болванки" начнем с подключения заголовочного файла *module.h*, необходимого для каждого модуля:

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

Также желательно подключить файл *kernel.h* — в нем содержатся полезные константы:

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

Если вы хотите просмотреть подключаемые файлы, то их стоит искать не в каталоге */usr/include/linux*, а в */usr/src/linux/include/linux*.

Любой модуль содержит функцию *init_module()*, которая запускается при загрузке модуля в ядро. При выгрузке модуля из ядра вызывается другая функция — *cleanup_module()*. Наш простой модуль будет состоять из этих двух функций.

Для демонстрации компиляции и использования модуля больше ничего не нужно. Но, чтобы создать видимость работы модуля, в эти функции мы добавим вызов функции *printk()*. Функция *printk()* позволяет отправить сообщение в системный журнал, обычно это */var/log/messages*.

Листинг 13.2. Модуль *first.c*

```
#include <linux/module.h>
#include <linux/kernel.h>
int init_module(void)
{
    printk(KERN_ALERT "Hello from kernel!\n");
    return 0;
}
void cleanup_module(void)
{
    printk(KERN_ALERT "first.c: removed\n");
}
```

Нужно отметить, что функция *init_module()* должна вернуть 0, что свидетельствует о том, что ошибок нет.

Если вернуть ненулевое значение, модуль загружен не будет, т. к. будет считаться, что при загрузке произошла ошибка. Функция *cleanup_module()* может не возвращать никаких значений.

Теперь рассмотрим функцию *printk()*. Это очень важная функция, она позволяет отправить сообщение в системный журнал (конечно, при условии, что запущен демон протоколирования — *syslog*, *klogd*, *syslog-ng* или любой другой). Сама функция *printk()* не производит запись в системный журнал.

Она записывает ваше сообщение в специальный буфер ядра, который и читается демоном протоколирования. Обычно демон *klogd* читает сообщения из этого буфера и передает демону *syslogd*, а тот, в свою очередь, записывает их в системный журнал.

Перед самым сообщением мы указали уровень протоколирования сообщения — `KERN_ALERT`. Это очень высокий уровень протоколирования, но благодаря этому сообщение будет не только передано демону протоколирования, но и выведено на консоль. Вообще, выводить сообщения на консоль нужно только в самых критических ситуациях, старайтесь избегать вывода на консоль просто так — это считается дурным тоном. Но при отладке своего модуля вы можете выводить сообщение на консоль, чтобы каждый раз не просматривать системный журнал.

Уровни протоколирования сообщения приведены в табл. 13.1.

Таблица 13.1. Уровни протоколирования сообщений

Уровень	Константа	Описание
7	<code>KERN_DEBUG</code>	Отладочные сообщения, самый низкий приоритет
6	<code>KERN_INFO</code>	Информационные сообщения
5	<code>KERN_NOTICE</code>	Это уже не информационное сообщение, но еще и не предупреждение
4	<code>KERN_WARNING</code>	Предупреждение: скоро может пойти что-то не так
3	<code>KERN_ERR</code>	Возникла ошибка
2	<code>KERN_CRIT</code>	Возникла критическая ошибка
1	<code>KERN_ALERT</code>	Тревога — система скоро "развалится"
0	<code>KERN_EMERG</code>	Система "развалилась" (система больше не может использоваться)

Сообщения с уровнем `KERN_ERR` и ниже (`KERN_CRIT`, `KERN_ALERT`, `KERN_EMERG`) обычно выводятся на консоль, хотя это зависит от настройки системы, но все сообщения записываются в системный журнал при условии, что демон протоколирования запущен и сама запись возможна. Например, `KERN_EMERG` может свидетельствовать об отказе жесткого диска, поэтому записать сообщение будет невозможно.

Конечно, если ничего страшного не случилось, а вы просто вывели сообщение с уровнем `KERN_ALERT`, на "здоровье" системы это никак не отразится.

Начиная с ядра версии 2.3.13 вы можете отказаться от стандартных названий функций `init_module()` и `cleanup_module()`. Вместо этих названий вы можете использовать собственные. Однако вам нужно указать, как называются новые функции инициализации и деинициализации модуля. Для этого используются макросы `module_init()` и `module_exit()`, определенные в `linux/init.h`.

В листинге 13.3 мы переопределили имена стандартных функций.

Листинг 13.3. Модуль `first2.c`

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
static int __init first_init(void)
{
    printk(KERN_ALERT "Hello from kernel\n");
    return 0;
}
static void __exit first_exit(void)
{
    printk(KERN_ALERT "first.c: removed\n");
}
module_init(first_init);
module_exit(first_exit);
```

В заголовочном файле `module.h` находятся также макросы, позволяющие указать информацию о модуле:

```
MODULE_LICENSE("GPL"); /* Лицензия */
MODULE_AUTHOR("Denis Kolisnichenko"); /* Автор */
MODULE_DESCRIPTION("Driver for /dev/mydev"); /* Описание */
MODULE_SUPPORTED_DEVICE("mydev"); /* Поддерживаемое устройство */
```

Все эти макросы на данном этапе не нужны, но понадобятся, когда вы будете создавать реальный модуль.

Компиляция модуля (01-165)

Компиляция модуля — это процесс, где можно легко наткнуться на множество подводных камней. Вроде бы все делаешь правильно, но модуль не собирается.

Вроде бы установил все заголовочные файлы, но компилятор упорно сообщает, что файл *linux/module.h* отсутствует, хотя вы уже десять раз убедились, что это не так...

Чтобы все работало, как следует, вам первым делом нужно сделать две вещи:

- ☐ убедиться, что установлены исходные коды ядра, а также заголовочные файлы;
- ☐ отказаться от использования X.Org (X Window).

Чем же не угодил X.Org? Дело в том, что сообщения, которые будут выводить ваши модули с помощью *printk()*, не будут отображаться в окне терминала!

Они будут занесены в системный журнал, но в окне терминала вы их не увидите. Модули не могут выводить сообщения на экран с помощью обычной функции *printf()* — это особенность модулей. Поэтому либо вы с комфортом работаете в консоли и видите все отладочные сообщения, выводимые модулем с помощью *printk()*, либо вы работаете в X.Org и читаете системный журнал */var/log/messages*.

Итак, начнем. Пусть наш модуль *first.c* находится в каталоге */root/module*. Создайте в этом каталоге файл *Makefile* следующего содержания:

```
obj-m := first.o
```

Да, это и есть наш *Makefile*. Всего одна строчка. Раньше *Makefile* для сборки одного модуля состоял как минимум из пяти строчек.

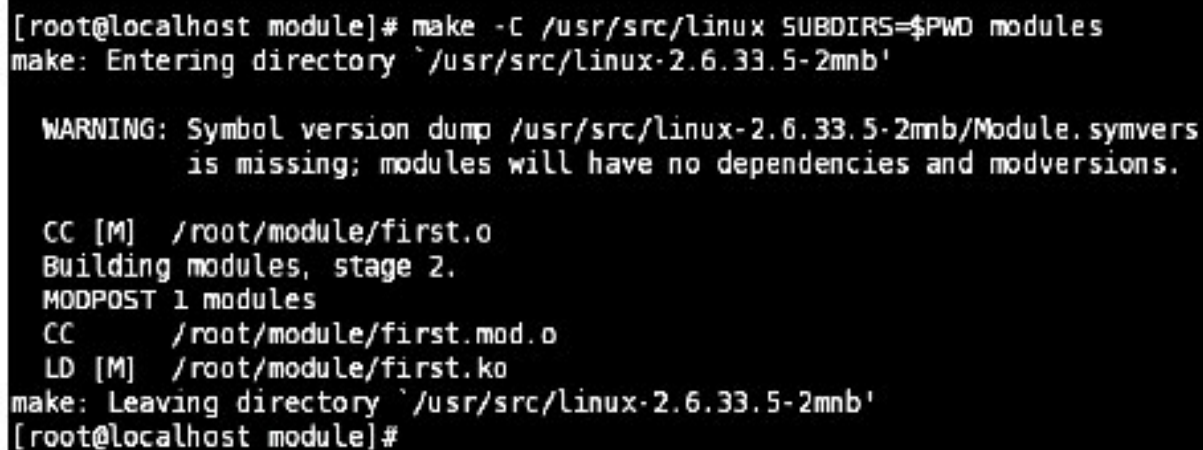
Теперь, находясь в каталоге */root/module*, введите команду:

```
# make -C /usr/src/linux SUBDIRS=$PWD modules
```

Разберемся, что означает эта команда. Как уже отмечалось, параметр `-C` задает каталог, в котором следует искать Makefile. Как видите, мы указываем не наш Makefile, а Makefile ядра. Обычно `/usr/src/linux` — это символическая ссылка на каталог `/usr/src/linux-2.6.x.x`.

Если в вашей системе нет такой ссылки, то или создайте ее, или укажите в команде путь к исходным текстам ядра. Команда `make` должна собрать цель `modules` (ведь мы же компилируем модуль!) из файла `/usr/src/linux/Makefile` (в нашем Makefile такой цели нет). А файлы, которые нужно скомпилировать, программа должна искать в текущем каталоге (значение переменной `$PWD` — именно поэтому важно вводить команду `make` из каталога, содержащего модуль). В этом каталоге программа `make` найдет наш Makefile модуля и обработает его.

Компилирование нашего простейшего модуля займет всего пару секунд.



```
[root@localhost module]# make -C /usr/src/linux SUBDIRS=$PWD modules
make: Entering directory `/usr/src/linux-2.6.33.5-2mnb'

WARNING: Symbol version dump /usr/src/linux-2.6.33.5-2mnb/Module.symvers
        is missing; modules will have no dependencies and modversions.

CC [M]  /root/module/first.o
Building modules, stage 2.
MODPOST 1 modules
CC      /root/module/first.mod.o
LD [M]  /root/module/first.ko
make: Leaving directory `/usr/src/linux-2.6.33.5-2mnb'
[root@localhost module]#
```

Рис. 13.1. Модуль скомпилирован

Но это только в том случае, если все пройдет гладко. Мы сначала проанализируем вывод команды `make`, а затем рассмотрим нештатные ситуации.

Из рис. 13.1 видно, что у нашего модуля отсутствует информация о зависимостях и описание версии модуля. Этим мы займемся позже — у нашего модуля пока не может быть никаких зависимостей, так что на предупреждение можете не обращать внимания. Начиная с ядра 2.6 файлы модулей имеют

"расширение" *ko* вместо *o*, чтобы их можно было легко отличить от обычных объектных файлов. В нашем случае создан модуль */root/module/first.ko*.

Теперь поговорим о нештатных ситуациях. При первой сборке модуля программа *make* может сообщить, что отсутствует конфигурация ядра (рис. 13.2).

```
***
*** You have not yet configured your kernel!
*** (missing kernel config file ".config")
***
*** Please run some configurator (e.g. "make oldconfig" or
*** "make menuconfig" or "make xconfig").
***
make[2]: *** [silentoldconfig] Ошибка 1
make[1]: *** [silentoldconfig] Ошибка 2

The present kernel configuration has modules disabled.
Type 'make config' and enable loadable module support.
Then build a kernel with module support enabled.

make: *** [modules] Ошибка 1
```

Рис. 13.2. Отсутствует конфигурация ядра

Избавиться от проблемы очень просто. Перейдите в каталог */usr/src/linux* и введите команду:

```
make menuconfig
```

В появившемся окне нажмите кнопку **Exit**. После чего *menuconfig* спросит, хотите ли вы сохранить конфигурацию ядра — то, что нам и нужно. Просто нажмите кнопку **Yes** (рис. 13.3), затем вы увидите сообщение, что конфигурация ядра записана (рис. 13.4).

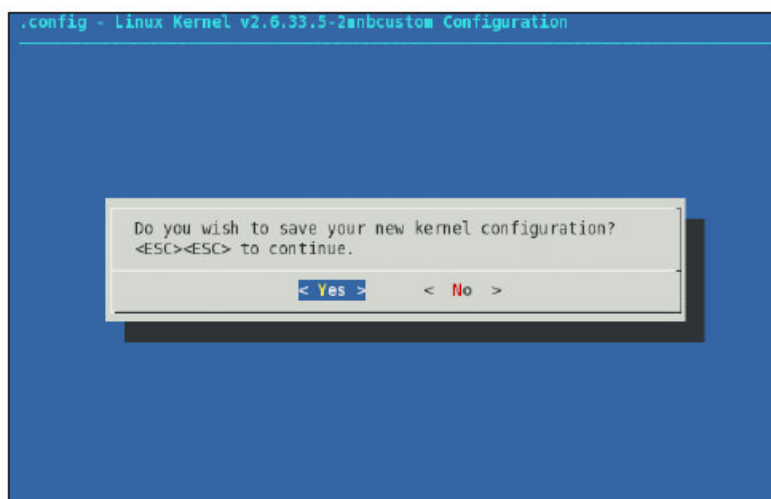


Рис. 13.3. Сохранить конфигурацию ядра?

```

[root@localhost module]# cd /usr/src/linux
[root@localhost linux]# make menuconfig
HOSTCC scripts/kconfig/lxdialog/checklist.o
HOSTCC scripts/kconfig/lxdialog/inputbox.o
HOSTCC scripts/kconfig/lxdialog/menubox.o
HOSTCC scripts/kconfig/lxdialog/textbox.o
HOSTCC scripts/kconfig/lxdialog/util.o
HOSTCC scripts/kconfig/lxdialog/yesno.o
HOSTCC scripts/kconfig/mconf.o
HOSTLD scripts/kconfig/mconf
scripts/kconfig/mconf arch/x86/Kconfig
#
# using defaults found in /boot/config-2.6.33.5-server-2mb
#
#
# configuration written to .config
#
*** End of Linux kernel configuration.
*** Execute 'make' to build the kernel or try 'make help'.

```

Рис. 13.4. Конфигурация ядра сохранена

После этого нужно перейти в каталог */root/module* и ввести ту самую команду *make*:

```
# make -C /usr/src/linux SUBDIRS=$PWD modules
```

На этот раз все должно пройти без замечаний.

Тестируем модуль

Теперь попробуем вставить модуль в ядро:

```
# insmod ./first.ko
```

Обратите внимание: нужно вставить в ядро именно *first.ko*, а не *first.o*. Если вам повезет, то на консоли и в системном журнале вы увидите заветное сообщение.

Но что делать, если вам не повезло и вместо приветствия модуля вы увидели сообщение:

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

Самое интересное, что весь Гугл пестрит сообщениями об этой ошибке, но ни в одном из руководств по созданию модуля не сказано, что с ней делать!

Проблема заключается в том, что модуль ядра должен соответствовать версии ядра.

Другими словами, в нашем случае ошибка заключалась в том, что использованы исходные тексты одной версии ядра, а

запустить модуль пытался под управлением другой версии ядра (точнее, сравниваются не версии, а сигнатуры версий ядра, но нам от этого не легче). Такая же ошибка могла произойти и у вас.

Например, вы изначально установили систему с ядром 2.6.33.3, также были установлены исходные тексты ядра, а ссылка `/usr/src/linux` указывала на каталог `/usr/src/linux-2.6.33.3`.

Затем вы перешли на новое ядро, скажем, на 2.6.33.5, загрузив сразу уже скомпилированную версию ядра — путем обновления RPM-пакета ядра.

Понятно, что сейчас вы используете версию 2.6.33.5, а исходные тексты вы обновить забыли. Потом без всякой задней мысли вы собираете модуль ядра — компиляция проходит без ошибок, потому что компилятору все равно, какие заголовочные файлы использовать.

Но при запуске модуля система обязательно сообщит вам, что ваш модуль имеет неправильный формат — и это не мудрено.

Бывает и другая причина отличия версий ядер. Разработчики дистрибутивов накладывают на оригинальные исходные тексты ядра множество патчей, поэтому могут возникнуть проблемы с компиляцией модулей ядра.

Вы, чтобы скомпилировать модуль, загружаете официальные исходные тексты ядра с *kernel.org*. Все бы хорошо, но ваш модуль является скомпилированным под загруженную версию ядра, но не под ту, которая сейчас используется

ПРИМЕЧАНИЕ

Если не совпадают сигнатуры версий ядра, теоретически модуль можно заставить работать, указав опцию `--force-vermagic` команды *modprobe*, например *modprobe --forcevermagic first*. Но это не выход из положения, и система может работать нестабильно, старайтесь не использовать данную опцию.

Что делать? Если вы уже собрали модуль, то оставьте его как есть. Перейдите в каталог с исходным кодом ядра и соберите ядро и модули заново:

```
# make menuconfig  
# make dep  
# make modules  
# make  
# make modules_install install
```

После этого перезагрузите систему, при загрузке из меню загрузчика выберите только что скомпилированное ядро (неплохо бы запомнить номер версии ядра — вы его узнаете из названия каталога `/usr/src/linux-x.x.x.x*`). После загрузки системы попробуйте вставить ваш модуль.

Все должно работать.

Правда, на данном этапе могут возникнуть (а могут и нет — все зависит от дистрибутива) проблемы куда более серьезные.

Как уже было отмечено, разработчики дистрибутива накладывают на оригинальные исходные тексты ядра всевозможные патчи. Может так случиться, что после перезагрузки системы с оригинальным ядром (которое вы скачали с *kernel.org*) система перестанет загружаться или откажут некоторые устройства.

Вам кажется процесс создания модулей ядра очень сложным? Так оно и есть, но, по сути, в этом нет ничего удивительного — ведь вы создаете часть операционной системы. А создание операционной системы никогда не было легким занятием.

Просмотреть информацию о модуле можно командой *modinfo* (рис. 13.5):

```
# modinfo ./first.ko
```



```

[root@localhost module]# insmod ./first.ko
insmod: error inserting './first.ko': -1 Invalid module format
[root@localhost module]# modinfo ./first.ko
filename:           ./first.ko
srcversion:         8826FC67F79883C144628DB
depends:
vermagic:           2.6.33.5-desktop-2mnb SMP mod_unload modversions 686
[root@localhost module]# _

```

Рис. 13.5. Ошибка при загрузке модуля и вывод информации о модуле

Посмотрите на рис. 13.5: модуль собран с использованием исходных текстов ядра 2.6.33.5-desktop, а в моей системе установлено ядро 2.6.33.5-server, хотя ошибка с номерами версий произошла по вине самих разработчиков дистрибутива. При установке пакета *kernel-source* не указывается сигнатура ядра, но логично было бы предположить, что при установке виртуального RPM-пакета должны быть установлены исходные тексты, соответствующие используемому ядру.

Другими словами, раз по умолчанию устанавливается ядро 2.6.33.5-server, то при установке пакета *kernel-source* должны быть установлены "исходники" именно "серверной" версии ядра. А на самом деле устанавливаются исходные тексты ядра 2.6.33.5-desktop.

Вот какой сюрприз подготовили разработчики Mandriva.

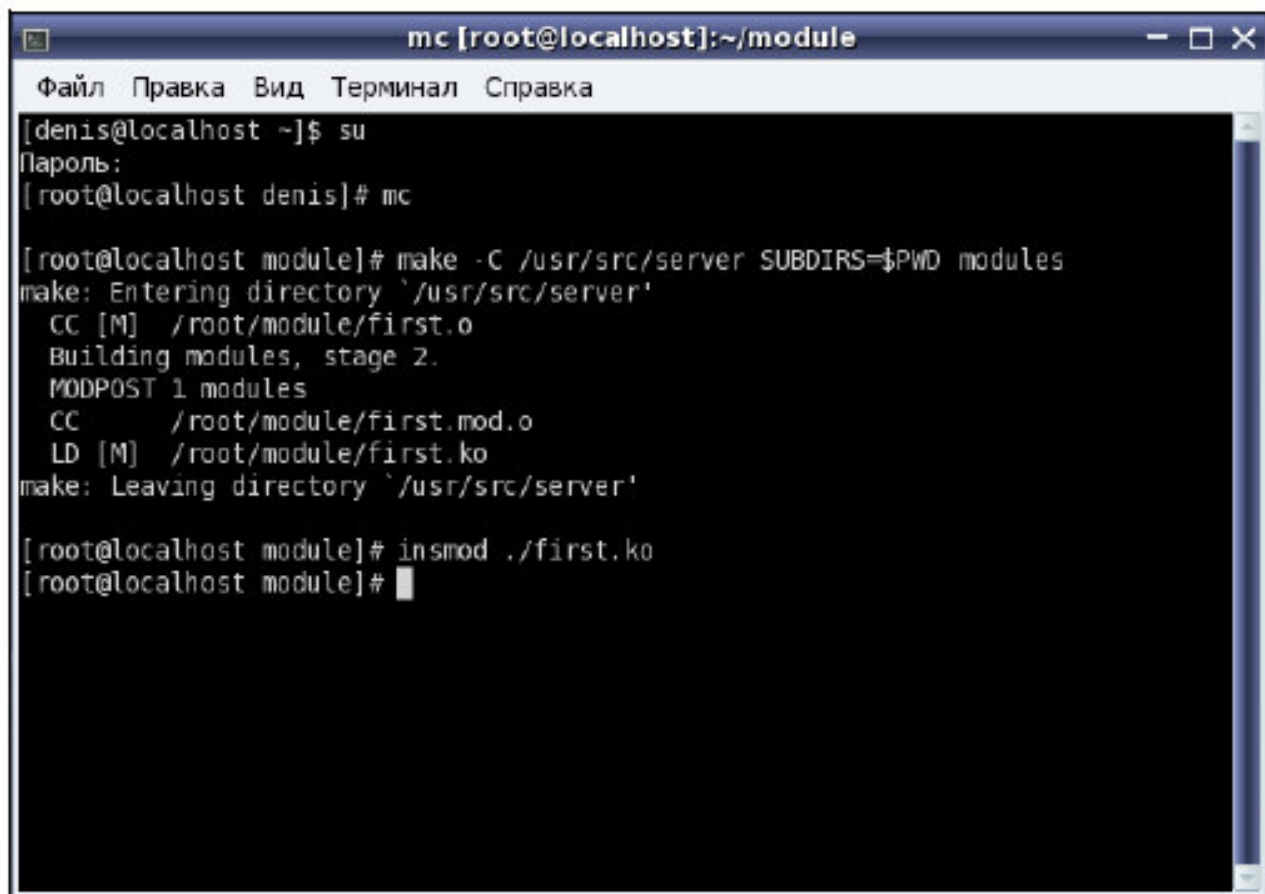
Проблема решилась следующим образом. Поскольку перекомпиляция ядра занимает довольно много времени, то проще установить пакет *kernel-server-devel-2.6.33.5-2mnb* и пересобрать наш модуль, чем компилировать исходные файлы "настольной" версии ядра и оставить модуль в покое.

После установки пакета в */usr/src* появился каталог *linux-server-2.6.33.5.2mnb* (для простоты переименованный в */usr/src/server*). Для компиляции модуля теперь используется команда:

```
# make -C /usr/src/server SUBDIRS=$PWD modules
```

Посмотрите на рис. 13.6: модуль собрался без ошибок, при установке командой *insmod* не было никаких "ругательств". Поскольку команда *insmod* была введена в терминале, то приветствие модуля мы не увидели, зато оно появилось в */var/log/messages* (рис. 13.7).

Информацию о загруженных модулях можно найти в файле `/proc/modules`. Наш модуль будет самый первый в списке загруженных модулей (рис. 13.8).

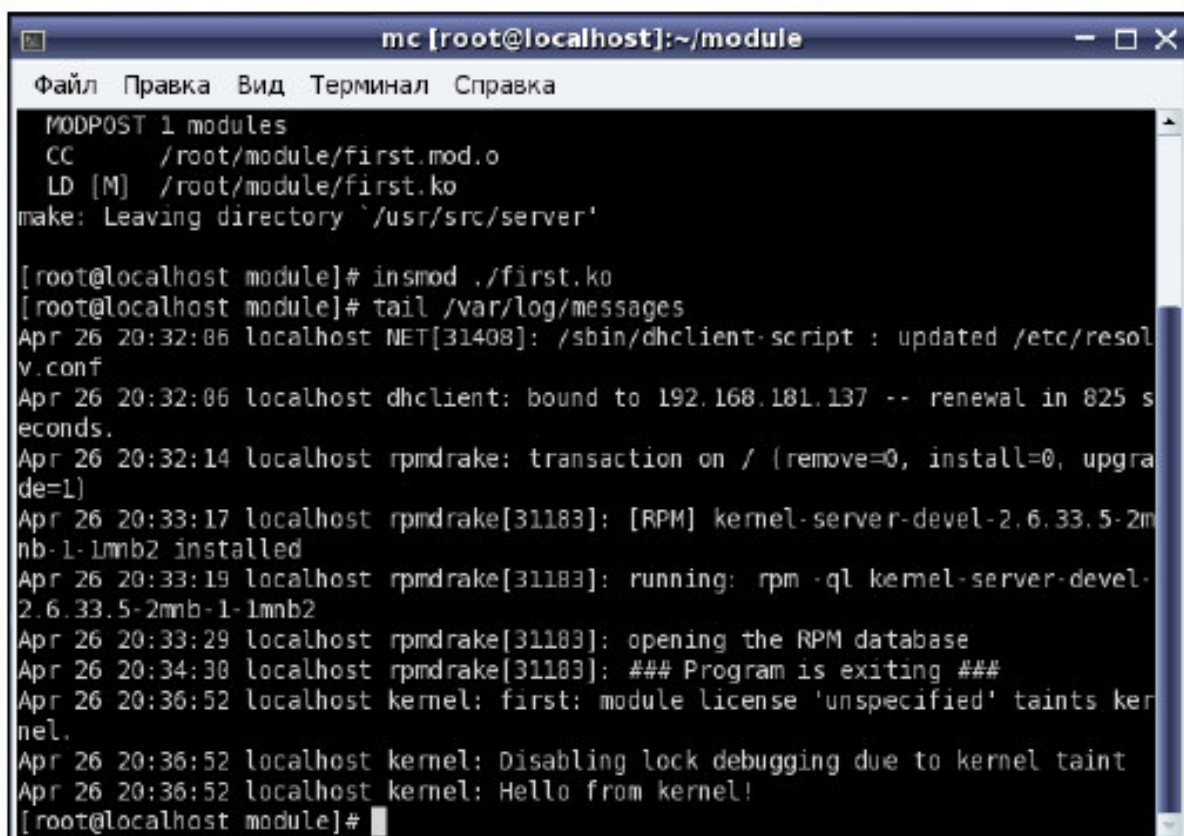


```
mc [root@localhost]:~/module
Файл Правка Вид Терминал Справка
[denis@localhost ~]$ su
Пароль:
[root@localhost denis]# mc

[root@localhost module]# make -C /usr/src/server SUBDIRS=$PWD modules
make: Entering directory `/usr/src/server'
CC [M] /root/module/first.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/module/first.mod.o
LD [M] /root/module/first.ko
make: Leaving directory `/usr/src/server'

[root@localhost module]# insmod ./first.ko
[root@localhost module]#
```

Рис. 13.6. Модуль добавлен в ядро без ошибок



```
mc [root@localhost]:~/module
Файл Правка Вид Терминал Справка
MODPOST 1 modules
CC /root/module/first.mod.o
LD [M] /root/module/first.ko
make: Leaving directory `/usr/src/server'

[root@localhost module]# insmod ./first.ko
[root@localhost module]# tail /var/log/messages
Apr 26 20:32:06 localhost NET[31408]: /sbin/dhclient-script : updated /etc/resolv.conf
Apr 26 20:32:06 localhost dhclient: bound to 192.168.181.137 -- renewal in 825 seconds.
Apr 26 20:32:14 localhost rpmrake: transaction on / (remove=0, install=0, upgrade=1)
Apr 26 20:33:17 localhost rpmrake[31183]: [RPM] kernel-server-devel-2.6.33.5-2mnb-1-lmnb2 installed
Apr 26 20:33:19 localhost rpmrake[31183]: running: rpm -ql kernel-server-devel-2.6.33.5-2mnb-1-lmnb2
Apr 26 20:33:29 localhost rpmrake[31183]: opening the RPM database
Apr 26 20:34:30 localhost rpmrake[31183]: ### Program is exiting ###
Apr 26 20:36:52 localhost kernel: first: module license 'unspecified' taints kernel.
Apr 26 20:36:52 localhost kernel: Disabling lock debugging due to kernel taint
Apr 26 20:36:52 localhost kernel: Hello from kernel!
[root@localhost module]#
```

Рис. 13.7. Приветствие модуля

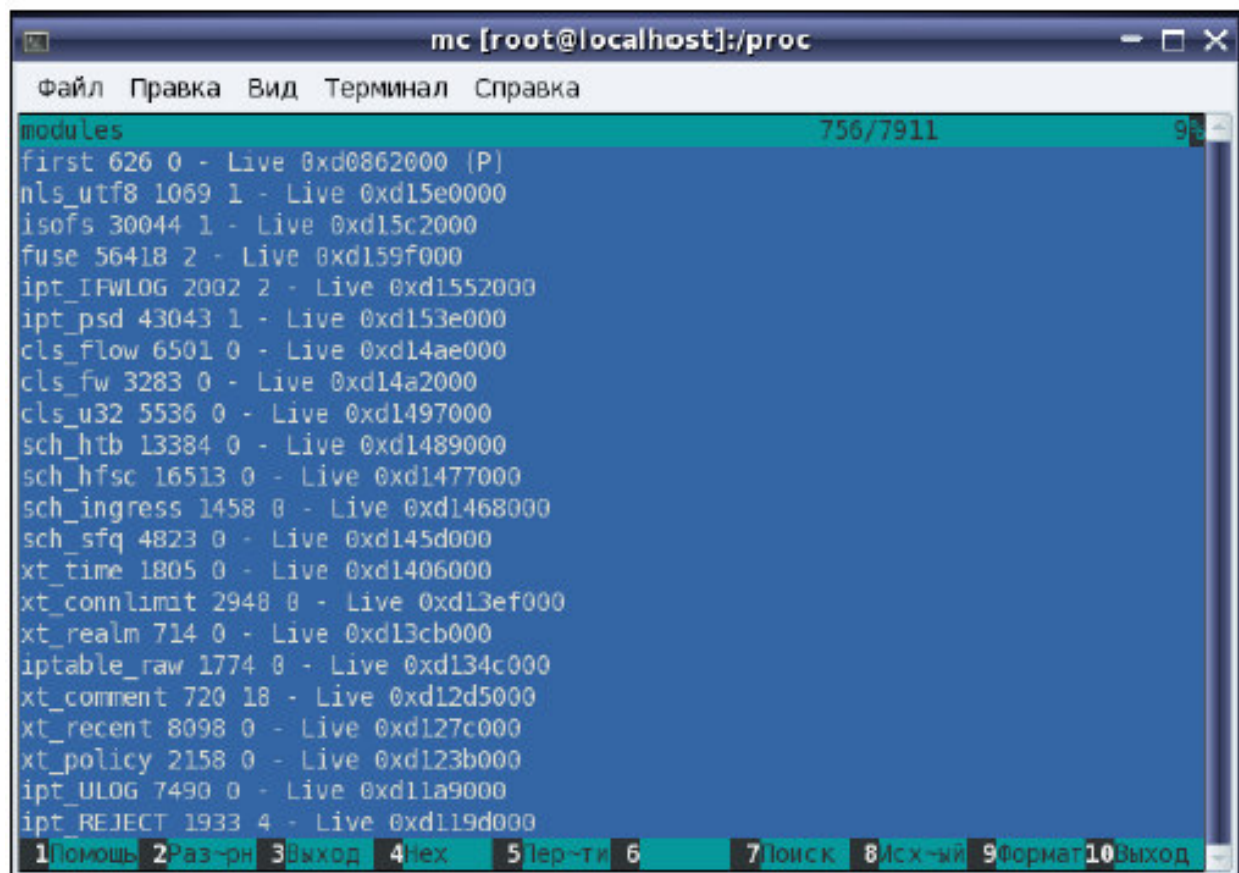


Рис. 13.8. Модуль *first* в списке модулей

Сборка сложных модулей

Сложные модули могут состоять из нескольких исходных файлов. К каждому исходному файлу нужно подключить как минимум два заголовочных файла (а также файлы, содержащие все необходимое для выполняемых модулем операций):

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

Представим, что у нас есть два файла с исходным текстом модуля (*file1.c* и *file2.c*). Тогда *Makefile* модуля будет выглядеть так:

```
obj-m += big.o
big-objs := file1.o file2.o
```

Простейший модуль для анализа (03-32)

В самом начале знакомства с техникой написания модулей ядра Linux проще не вдаваться в пространные объяснения, а создать простейший модуль (код такого модуля интуитивно понятен всякому программисту), собрать его и наблюдать его

работу. И только потом, ознакомившись с некоторыми основополагающими принципами и приемами работы из мира модулей, перейти к их систематическому изучению.

Вот с такого образца простейшего модуля ядра мы и начнем наш экскурс:

hello_printk.c :

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

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

static int __init hello_init( void ) {
    printk( "Hello, world!" );
    return 0;
}

static void __exit hello_exit( void ) {
    printk( "Goodbye, world!" );
}

module_init( hello_init );
module_exit( hello_exit );
```

Сборка модуля (3-34)

Раньше (ядро 2.4) для сборки модуля приходилось вручную писать достаточно замысловатые Makefile, и это вы можете встретить и в ряде публикаций. Но, постольку это однотипный процесс для любых модулей, разработчики ядра (к ядру 2.6) заготовили сценарии сборки модулей, а в вашем Makefile требуется только записать конкретизирующие значения переменных и формально вызвать макросы. С тех пор сборка модулей стала гораздо проще.

Для сборки созданного модуля используем скрипт сборки *Makefile*, который будет с минимальными изменениями повторяться при сборке всех модулей ядра:

Makefile :

```
CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
```

```

PWD = $(shell pwd)
DEST = /lib/modules/$(CURRENT)/misc
TARGET = hello_printk
obj-m := $(TARGET).o
default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
    @rm -f *.o *.cmd *.flags *.mod.c *.order
    @rm -f *.*.cmd *.symvers *~ *.*~ TODO.*
    @rm -fR .tmp*
    @rm -rf .tmp_versions

```

Цель сборки `clean` — присутствует в таком и неизменном виде практически во всех далее приводимых файлах сценариев сборки (Makefile) и не будет там далее показываться. В ней мы приводим все типы создаваемых временных файлов, список которых меняется от версии к версии ядра.

Делаем сборку модуля ядра, выполнив команду `make`. Почти наверняка при первой сборке модуля, или делая это в свежееустановленной системе Linux, вы получите результат, подобный следующему:

\$ make

```

make -C /lib/modules/3.12.10-300.fc20.i686/build
M=/home/Olej/2014_WORK/GlobalLogic/BOOK.Kernel.org/Examples.BOOK/first_hello modules
make: *** /lib/modules/3.12.10-300.fc20.i686/build: Нет
такого файла или каталога. Останов.
make: *** [default] Ошибка 2

```

Если такое случилось, то связано это с тем, что в системе, которая специально не готовилась для сборки ядра, **не установлены** те программные пакеты, которые необходимы для этой специфической деятельности, в частности, заголовочные файлы кода ядра. А если установлены все необходимые программные пакеты для сборки модулей ядра, то мы должны получить что-то подобное следующему:

\$ make

```

make -C /lib/modules/2.6.32.9-70.fc12.i686.PAE/build
M=/home/olej/2011_WORK/Linux-kernel/examples

```

```
make[1]: Entering directory
`/usr/src/kernels/2.6.32.9-70.fc12.i686.PAE'
  CC [M] /home/olej/2011_WORK/Linux-
kernel/examples/own-modules/1/hello_printk.o
Building modules, stage 2.
MODPOST 1 modules
  CC /home/olej/2011_WORK/Linux-
kernel/examples/own-modules/1/hello_printk.mod.o
  LD [M] /home/olej/2011_WORK/Linux-
kernel/examples/own-modules/1/hello_printk.ko
make[1]: Leaving directory
`/usr/src/kernels/2.6.32.9-70.fc12.i686.PAE'
```

На этом модуль создан. Начиная с ядер 2.6, расширение файлов модулей сменилось с *.o на *.ko (хотя формат модуля и совпадает с форматом объектного модуля):

```
$ ls *.ko
hello_printk.ko
```

```
$ file hello_printk.ko
hello_printk.ko: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV),
BuildID[sha1]=dc1a94b0bc8019d82aee89ffff3b3e562249f016, not stripped
```

Форматом модуля является обычный **объектный** ELF формат, но дополненный в таблице внешних имён некоторыми дополнительными именами, такими как : `__mod_author5`, `__mod_license4`, `__mod_srcversion23`, `__module_depends`, `__mod_vermagic5`, ... которые определяются специальными модульными макросами.

Загрузка и исполнение (02-34)

Наш модуль при загрузке/выгрузке выводит сообщение посредством вызова `printk()`. Этот вывод направляется на **текстовую консоль**. При работе в **терминале** (в графической системе X11) вывод не попадает в терминал, и его можно видеть **только** в лог файле `/var/log/messages` (в RPM-дистрибутивах) или в `/var/log/kern.log` (в свежих дистрибутивах Debian):

```
$ sudo insmod hello_printk.ko
$ sudo rmmod hello_printk
```

```
$ tail -n2 /var/log/kern.log
```

```
Jun 5 15:27:19 R420 kernel: [14377.822570] Hello, world!
```

```
Jun 5 15:27:32 R420 kernel: [14390.766363] Goodbye, world!
```

В любом случае вы можете смотреть, независимо от деталей логирования, сообщения ядра командой *dmesg*:

```
$ dmesg | tail -n2
```

```
[14377.822570] Hello, world!
```

```
[14390.766363] Goodbye, world!
```

Это два основных метода визуализации сообщений ядра (занесенных в системный журнал): утилита *dmesg* и прямое чтение файла журнала. Они имеют несколько отличающийся формат: файл журнала содержит «читабельные» метки времени поступления сообщений, что иногда бывает нужно. Кроме того, прямое чтение файла журнала требует в некоторых дистрибутивах наличия прав *root*.

Но и чисто в текстовую консоль (при отработке в текстовом режиме) вывод направляется не непосредственно, а через демон системного журнала, а выводится на экран только если демон конфигурирован для вывода такого уровня сообщений. Изучаем полученный файл модуля:

Изучаем файл модуля, загружаем его и исследуем несколькими часто используемыми командами:

```
$ modinfo hello_printk.ko
```

```
filename: /home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello/hello_printk.ko
```

```
author: Oleg Tsiliuric <olej.tsil@gmail.com>>
```

```
license: GPL
```

```
srcversion: 9526D45F847C0B4EF5BBBDC
```

```
depends:
```

```
retpoline: Y
```

```
name: hello_printk
```

```
vermagic: 5.4.0-113-generic SMP mod_unload modversions
```

```
$ sudo insmod hello_printk.ko  
$ lsmod | head -n2
```

Module Size Used by

```
hello_printk 16384 0
```

Все, что мы проделали, пока не имеет существенного смысла... Но отметим одно очень важное обстоятельство: сообщения «Hello, world!» и «Goodbye, world!» были записаны в системный журнал кодом, выполняющимся в привилегированном (супервизорном) режиме **ядра**. Такой код может иметь доступ ко всем возможностям компьютера без ограничения.

Точки входа и завершения (03-36)

Любой модуль должен иметь объявленные функции **входа** (инициализации) модуля и его **завершения** (не обязательно, может отсутствовать). Функция инициализации будет вызываться (после проверки и соблюдения всех достаточных условий) при выполнении команды *insmod* для модуля. Точно так же, функция завершения будет вызываться при выполнении команды *rmmod*.

Функция инициализации имеет прототип и объявляется именно как функция инициализации макросом *module_init()*, как это было сделано в только что рассмотренном примере:

```
static int __init hello_init(void) {  
    ...  
}  
  
module_init(hello_init);
```

Функция завершения, совершенно симметрично, имеет прототип и объявляется

макросом *module_exit()*, как было показано:


```
static void __exit hello_exit(void) {
...
}
module_exit(hello_exit);
```

Обратите внимание - функция завершения по своему прототипу не имеет возвращаемого значения, и, поэтому, она даже **не может сообщить** о невозможности каких-либо действий, когда она уже начала выполняться. Идея состоит в том, что система при *rtmod* сама проверит допустимость вызова функции завершения, и если они не соблюдены, просто не вызовет эту функцию.

Показанные выше соглашения по объявлению функций инициализации и завершения являются общепринятыми. Но существует ещё один не документированный способ описания этих функций: воспользоваться непосредственно их **предопределёнными** именами, а именно *init_module()* и *cleanup_module()*. Это может быть записано так:

```
int init_module(void) {
...
}
void cleanup_module(void) {
...
}
```

При такой записи необходимость в использовании макросов *module_init()* и *module_exit()* отпадает, а использовать квалификатор *static* с этими функциями нельзя (они должны быть известны при связывании модуля с ядром).

Конечно, такая запись никак не способствует улучшению читаемости текста, но иногда может существенно сократить рутину записи, особенно в коротких иллюстративных примерах. Мы будем иногда использовать её в демонстрирующих программных примерах. Кроме того, такую запись нужно понимать, так как она используется кое-где в коде ядра, в литературе и в обсуждениях по ядру.

Внутренний формат модуля (03-37)

Относительно структуры модуля ядра мы можем увидеть, для начала, что собранный нами модуль является **объектным** файлом ELF формата:

```
$ file hello_printk.ko
```

```
hello_printk.ko: ELF 32-bit LSB relocatable, Intel 80386,
version 1 (SYSV), not stripped
```

Всесторонний анализ объектных файлов производится утилитой *objdump*, имеющей множество опций в зависимости от того, что мы хотим посмотреть:

```
$ objdump
```

```
Usage: objdump <option(s)> <file(s)>
```

```
Display information from object <file(s)>.
```

```
....
```

Структура секций объектного файла модуля (показаны только те, которые могут нас заинтересовать — теперь или в дальнейшем):

```
$ objdump -h hello_printk.ko
```

```
hello_printk.ko: file format elf32-i386
```

```
Sections:
```

Idx	Name	Size	VMA	LMA	File off	Algn
...						
1	.text	00000000	00000000	00000000	00000058	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
2	.exit.text	00000015	00000000	00000000	00000058	2**0
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
3	.init.text	00000011	00000000	00000000	0000006d	2**0
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
...						
5	.modinfo	0000009b	00000000	00000000	000000a0	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
6	.data	00000000	00000000	00000000	0000013c	2**2
	CONTENTS, ALLOC, LOAD, DATA					
...						
8	.bss	00000000	00000000	00000000	000002a4	2**2
	ALLOC					
...						

Здесь секции:

– .text — код модуля (инструкции);

- `.init.text`, `.exit.text` — код инициализации модуля и завершения, соответственно;
- `.modinfo` — текст макросов модуля;
- `.data` — инициализированные данные;
- `.bss` — не инициализированные данные (Block Started Symbol);

Ещё один род чрезвычайно важной информации о модуле — это список имён модуля (которые могут иметь локальную или глобальную видимость, и могут экспортироваться модулем), эту информацию извлекаем так:

```
$ objdump -t hello_printk.ko
hello_printk.ko:      file format elf32-i386
SYMBOL TABLE:
...
00000000 l      F .exit.text      00000015 hello_exit
00000000 l      F .init.text      00000011 hello_init
00000000 l      O .modinfo      00000026 __mod_author5
00000028 l      O .modinfo      0000000c __mod_license4
...
```

Вывод диагностики модуля

Для диагностического вывода из модуля используем вызов `printk()`. Он настолько подобен по своим правилам и формату общеизвестному из пользовательского пространства `printf()`, что даже не требует дополнительного описания. Отметим только некоторые тонкие особенности `printk()` относительно `printf()`:

Сам вызов `printk()` и все сопутствующие ему константы и определения найдёте в файле определений `/lib/modules/`uname -r`/build/include/linux/kernel.h`:

```
asmlinkage int printk( const char * fmt, ... )
```

Первому параметру (форматной строке) **может** предшествовать (а может и не предшествовать) константа квалификатор, определяющая уровень сообщений. Определения констант для 8 уровней сообщений, записываемых в вызове `printk()` вы найдёте в файле `printk.h`:

```
#define KERN_EMERG      "<0>"    /* system is unusable          */
#define KERN_ALERT      "<1>"    /* action must be taken immediately */
#define KERN_CRIT       "<2>"    /* critical conditions          */
#define KERN_ERR        "<3>"    /* error conditions             */
#define KERN_WARNING    "<4>"    /* warning conditions           */
#define KERN_NOTICE     "<5>"    /* normal but significant condition */
#define KERN_INFO       "<6>"    /* informational                */
#define KERN_DEBUG      "<7>"    /* debug-level messages         */
```

Предшествующая константа не является отдельным параметром (не отделяется запятой), и (как видно из определений) представляет собой символьную строку определённого вида, которая **конкатенируется** с первым параметром (являющимся, в общем случае, **форматной** строкой). Если такая константа не записана, то устанавливается уровень вывода этого сообщения по умолчанию. Таким образом, следующие формы записи могут быть эквивалентны:

```
printk( KERN_WARNING "string" );
printk( "<4>" "string" );
printk( "<4>string" );
printk( "string" );
```

Основные ошибки модуля

Нормальная загрузка модуля командой `insmod` происходит без сообщений. Но при ошибке выполнения загрузки команда выводит сообщение об ошибке — модуль в этом случае **не будет загружен** в состав ядра. Вот наиболее часто получаемые ошибки при неудачной загрузке модуля, и то, как их следует толковать:

insmod: *can't read './params': No such file or directory* — неверно указан путь к файлу модуля, возможно, в указании имени файла не включено стандартное расширение файла модуля (*.ko), но это нужно делать обязательно.

insmod: *error inserting './params.ko': -1 Operation not permitted* — наиболее вероятная причина: у вас элементарно нет прав root для выполнения операций установки модулей. Другая причина того же сообщения: функция инициализации модуля возвратила ненулевое значение, нередко такое завершение планируется **преднамеренно**, особенно на этапах отладки модуля (и о таком варианте использования мы будем говорить неоднократно).

insmod: *error inserting './params.ko': -1 Invalid module format* — модуль скомпилирован для другой версии ядра; перекомпилируйте модуль. Это та ошибка, которая почти наверняка возникнет, когда вы перенесёте любой рабочий пример модуля на другой компьютер, и попытаетесь там загрузить модуль: совпадение сигнатур разных инсталляций до уровня подверсий — почти невероятно.

insmod: *error inserting './params.ko': -1 File exists* — модуль с таким именем уже загружен, попытка загрузить модуль повторно.

insmod: *error inserting './params.ko': -1 Invalid parameters* — модуль запускается с указанным параметром, не соответствующим по типу ожидаемому для этого параметра.

insmod: *ERROR: could not insert module ./jiff.ko: Unknown symbol in module* — это тонкая ошибка, которая может привести в замешательство, и последняя, на которой мы остановимся, она может возникнуть по нескольким причинам:

- вы использовали в коде имя (функции), которое описано в заголовочных файлах (поэтому код и компилируется), но которое **не экспортируется** ядром, и выясняется это только на этапе загрузки модуля (связывания имён), типичный пример тому — использование функций системных вызовов вида `sys_write()`;
- в коде модуля отсутствует макрос определения лицензии GPL вида:

```
/MODULE_LICENSE( "GPL" );
```

При этом имена, экспортируемые для GPL лицензируемого кода могут стать не экспортируемыми для вашего кода, пример тому `kallsyms_on_each_symbol()`. Тот же эффект можно получить если указана другая лицензия, как BSD или MIT.

Варианты загрузки модулей

При отработке нового модуля его загрузку на тестирование вы будете, скорее всего, производить утилитой `insmod`. Утилита

insmod получает **имя файла модуля**, и пытается загрузить его без проверок каких-либо взаимосвязей, как это описано ниже. Если некоторые требуемые имена **не экспортированы** к моменту insmod, то загрузка отвергается.

В последующей эксплуатации оттестированных модулей для их загрузки предпочтительнее утилита modprobe. Утилита modprobe сложнее: ей передаётся или **универсальный идентификатор**, или непосредственно **имя модуля**. Если modprobe получает универсальный идентификатор, то она сначала пытается найти соответствующее имя модуля в файле /etc/modprobe.conf (устаревшее), или в файлах *.conf каталога /etc/modprobe.d, где каждому универсальному идентификатору поставлено в соответствие имя модуля (в строке alias ... , смотри modprobe.conf(5)).

Далее, по имени модуля утилита modprobe, по содержимому файла зависимостей:

```
$ ls -l /lib/modules/`uname -r`/*.dep
```

```
-rw-r--r--  1 root root 206131 Mar  6 13:14 /lib/modules/2.6.32.9-70.fc12.i686.PAE/modules.dep
```

- пытается установить зависимости запрошенного модуля: модули, от которых зависит запрошенный, будут загружаться утилитой прежде него. Файл зависимостей modules.dep формируется командой :

```
# depmod -a
```

Той же командой (время от времени) мы обновляем и большинство других файлов modules.* этого каталога:

```
$ ls /lib/modules/`uname -r`
```

build	modules.block	modules.inputmap	modules.pcimap	updates
extra	modules.ccwmap	modules.isapnpmap	modules.seriomap	vdso
kernel	modules.dep	modules.modesetting	modules.symbols	weak-updates
misc	modules.dep.bin	modules.networking	modules.symbols.bin	
modules.alias	modules.drm	modules.ofmap	modules.usbmap	
modules.alias.bin	modules.ieee1394map	modules.order	source	

```
Интересующий нас файл содержит строки вида:  
$ cat /lib/modules/`uname -r`/modules.dep  
...  
kernel/fs/ubifs/ubifs.ko: kernel/drivers/mtd/ubi/ubi.ko kernel/drivers/mtd/mtd.ko  
...
```

Каждая такая строка содержит: а). модули, от которых зависит данный (например, модуль `ubifs` зависит от 2-х модулей `ubi` и `mtd`), и б). полные пути к файлам всех модулей. После этого загрузить модули не представляет труда, и непосредственно для этой работы включается (по каждому модулю последовательно) утилита `insmod`.

Утилита `rmmod` выгружает ранее загруженный модуль, в качестве параметра утилита должна получать **имя модуля** (не **имя файла модуля**). Если в системе есть модули, зависящие от выгружаемого (счётчик ссылок использования модуля больше нуля), то выгрузка модуля не произойдёт, и утилита `rmmod` завершится аварийно.

Совершенно естественно, что все утилиты `insmod`, `modprobe`, `depmod`, `rmmod` слишком кардинально влияют на поведение системы, и для своего выполнения требуют права `root`.

Пример 1: клавиатурный шпион

Наверняка все пользовались особыми программами — клавиатурными снифферами. Такие программы также называют клавиатурными шпионами (`key loggers`).

Клавиатурный шпион запускается незаметно — его никто не видит, но он есть. Он перехватывает все нажатия клавиш и записывает их в отдельный файл, анализируя который, можно понять, какие клавиши нажимал пользователь.

Мы реализуем наш клавиатурный шпион в виде модуля ядра — идеальная среда для написания подобного рода программ. Наш шпион будет записывать информацию о нажатых клавишах не в отдельный файл, а в системный журнал — для упрощения кода модуля.

На примере клавиатурного шпиона будет продемонстрирована обработка прерываний (IRQ) и очереди задач. Наш модуль будет содержать обработчик прерывания 1 (это и есть прерывание клавиатуры на архитектуре Intel). Вы только подумайте: средняя скорость печати опытного пользователя на клавиатуре составляет 100–250 символов в минуту. Следовательно, каждую минуту в системный журнал будет записываться до 250 записей, что может отрицательно сказаться на производительности системы. Поэтому и используем очереди: каждую минуту будет до 250 раз вызываться обработчик прерывания 1, а запись, благодаря очереди задач, в системный журнал будет происходить тогда, когда удобно ядру.

Для реализации нашего шпиона нужно подключить следующие заголовочные файлы:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <asm/io.h>
```

Затем нужно определить очередь заданий myqueue:

```
#define MY_WORK_QUEUE_NAME "WQsched.c"
static struct workqueue_struct *myqueue;
```

Некоторые функции, относящиеся к очереди заданий, работают, только если модуль лицензирован под GPL, поэтому нам ничего не остается, как объявить это:

```
MODULE_LICENSE("GPL");
```

В процессе инициализации модуля мы заменим оригинальный обработчик прерывания клавиатуры (IRQ 1). Логично было бы предположить, что при удалении модуля мы должны восстановить оригинальный обработчик прерывания, но поскольку это невозможно (оригинальный обработчик будет восстановлен только после клавиатуры), то функция cleanup_module() не будет ничего делать:

```
void cleanup_module()
```



```
{  
}
```

Функция инициализации модуля выглядит так:

```
int init_module()  
{  
    /* Создаем очередь */  
    myqueue = create_workqueue(MY_WORK_QUEUE_NAME);  
    /* Освобождаем старый обработчик прерывания IRQ 1 */  
    free_irq(1, NULL);  
    return request_irq(1,                /* Номер IRQ */  
        keyboard_handler,                /* наш обработчик */  
        SA_SHIRQ,  
        "keyboard_irq_handler",  
        (void *) (keyboard_handler));  
}
```

Первым делом функция *init_module()* создает очередь, затем она освобождает стандартный обработчик прерывания 1 с помощью вызова *free_irq()*. Потом мы назначаем новый обработчик прерывания с помощью вызова *request_irq()*. Первый параметр этого вызова — номер прерывания, затем идет название функции обработчика. Флаг *SA_SHIRQ* означает, что это прерывание может совместно обслуживаться другими обработчиками. Следующий параметр — название нашего обработчика, которое будет отображено в */proc/interrupts*. Последний параметр — ID устройства. Если вы не используете флаг *SA_SHIRQ*, установите его в *NULL*, в противном случае нужно указать адрес функции-обработчика прерывания.

Теперь рассмотрим функцию *keyboard_handler()*. Задача этой функции — получить состояние клавиатуры и скан-код нажатой клавиши и передать все это функции *log_char()*, которая обеспечит протоколирование нажатой клавиши. Вот только вместо прямого вызова *log_char()* функция *keyboard_handler()* добавляет вызов *log_char()* в очередь заданий. Для формирования задания используется структура *work_struct*. В качестве задания

выступает вызов `log_char()` с передачей ему параметра — скан-кода клавиши.

```
irqreturn_t keyboard_handler(int irq, void *dev_id,
struct pt_regs *regs)
{
static int initialised = 0;
static unsigned char scancode;
static struct work_struct mytask;
unsigned char status;
/* Читаем состояние клавиатуры и скан-код клавиши */
status = inb(0x64);
scancode = inb(0x60);
/* Формируем задачу mytask — ею будет функция
log_char */
if (initialised == 0) {
INIT_WORK(&mytask, log_char, &scancode);
initialised = 1;
} else {
PREPARE_WORK(&mytask, log_char, &scancode);
}
/* Добавляем задачу mytask в очередь myqueue */
queue_work(myqueue, &mytask);
return IRQ_HANDLED;
}
```

Функция `log_char()` очень проста. Ее задача — записать в системный журнал сканкод клавиши. Для облегчения поиска строк, выводимых сниффером в журнал, я добавил маркер `=007=`. После этого вывести все строки можно будет так:

```
cat /var/log/messages | grep =007= | less
```

Вот код самой функции:

```
static void log_char(void *scancode)
{
    printk(KERN_INFO "=007=: Scancode %x %s.\n",
```

```

        (int)*((char *)scancode) & 0x7F,
        *((char *)scancode) & 0x80 ? "Released" :
        "Pressed");
}

```

В листинге 13.6 представлен полный код модуля без лишних комментариев.

Листинг 13.6. Модуль `sniffer.c`

```

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <asm/io.h>

#define MY_WORK_QUEUE_NAME "WQsched.c"
static struct workqueue_struct *myqueue;
MODULE_LICENSE("GPL");

static void log_char(void *scancode)
{
    printk(KERN_INFO "=007=: Scancode %x %s.\n",
        (int)*((char *)scancode) & 0x7F,
        *((char *)scancode) & 0x80 ? "Released" :
        "Pressed");
}

irqreturn_t keyboard_handler(int irq, void *dev_id,
struct pt_regs *regs)
{
    static int initialised = 0;
    static unsigned char scancode;
    static struct work_struct mytask;
    unsigned char status;

    /* Читаем состояние клавиатуры и скан-код клавиши */
    status = inb(0x64);
    scancode = inb(0x60);

    /* Формируем задачу mytask — ею будет функция
    log_char */
    if (initialised == 0) {

```

```

    INIT_WORK(&mytask, log_char, &scancode);
    initialised = 1;
} else {
    PREPARE_WORK(&mytask, log_char, &scancode);
}
/* Добавляем задачу mytask в очередь myqueue */
queue_work(myqueue, &mytask);
return IRQ_HANDLED;
}
int init_module()
{
    /* Создаем очередь */
    myqueue = create_workqueue(MY_WORK_QUEUE_NAME);

    /* Освобождаем старый обработчик прерывания IRQ 1 */
    free_irq(1, NULL);
    return request_irq(1, /* Номер IRQ */
        keyboard_handler, /* Наш обработчик */
        SA_SHIRQ,
        "keyboard_irq_handler",
        (void *) (keyboard_handler));
}
void cleanup_module()
{
}

```

Пример 2: драйвер абстрактного символьного устройства

Драйверы устройств и ядро

Представим, что у вас есть помощник и вы его попросили доставить какие-то документы вашему партнеру. Вас не интересует, как он это сделает — будет ли он добираться на

метро, на машине или просто пешком. Вам важно, чтобы он выполнил ваше задание — доставил документы.

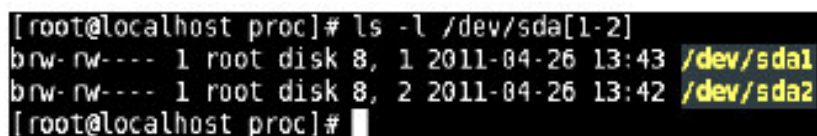
Так вот, драйвер устройства — это ваш помощник. Ядро не знает особенностей устройства, но оно может подать стандартные команды, которые должны принимать все драйверы подобного рода устройств. Например, ядро подает команду уменьшить громкость — драйвер звуковой платы примет команду и выполнит ее (как он это сделает — мы не знаем, все зависит от самого устройства).

Понятно, что нельзя отправить команду уменьшения громкости жесткому диску.

При разработке драйверов устройства вам нужно знать, что такое младший и старший номер устройства. Выполните команду:

```
ls -l /dev/sda[1-2]
```

Если на вашем жестком диске больше, чем два раздела (например, четыре раздела), в скобках можете указать другие числа (1–4). Результат выполнения этой команды изображен на рис. 13.9.



```
[root@localhost proc]# ls -l /dev/sda[1-2]
brw-rw---- 1 root disk 8, 1 2011-04-26 13:43 /dev/sda1
brw-rw---- 1 root disk 8, 2 2011-04-26 13:42 /dev/sda2
[root@localhost proc]#
```

Рис. 13.9. Старший и младший номера устройств

Взгляните на числа, разделенные запятой: старший номер (8) одинаковый для каждого устройства, поскольку оба устройства (/dev/sda1 и /dev/sda2) обслуживаются одним и тем же драйвером. После запятой следует младший номер устройства.

Итак, старший номер — это номер драйвера. У каждого драйвера свой уникальный номер. Младший номер используется драйвером, чтобы различать устройства, которыми он управляет. Поскольку у каждого устройства разный младший номер, то драйвер "видит" их как разные аппаратные устройства, а не как одно целое.

Как мы уже знаем, все устройства можно разделить на две группы: *блочные и символьные*. С блочными устройствами обмен данными осуществляется блоками (наборами байтов), а с символьными — посимвольно. Посмотрите на рис. 13.9: перед правами доступа (перед буквой *r*) вы видите букву *b* (от англ. *block*). Это означает, что устройство блочное, если же на этом месте будет буква *c* (от англ. *char*), то перед вами символьное устройство.

Создать устройство можно командой `mknod`, например:

```
mknod /dev/sda3 b 8 3
```

Вы только что создали еще одно блочное устройство (*b*) `/dev/sda3`, старший номер устройства равен 8, младший — 3. Файлы устройств не обязательно размещать в каталоге `/dev` — вы можете выбрать любой другой, но так требует традиция.

Понятие устройства в Linux очень абстрактное. Хотя мы и создали файл устройства `/dev/sda3`, это не означает, что на нашем жестком диске появился еще один раздел: ведь этот файл никак не связан с аппаратной частью. Да и никак не будет связан, поскольку драйвер жесткого диска "увидит", что третьего раздела на жестком диске нет, и попросту не будет использовать "лишний" файл.

Символьные устройства

Возможные операции

Драйвер может выполнять какие-то операции с устройством. Операции стандартизированы и описаны в структуре `file_operations` в файле `linux/fs.h`.

Данная структура содержит указатели функций драйвера, которые используются для выполнения разных операций с устройством. Понятно, что драйвер не должен реализовать все функции. Функции, не реализованные драйвером, заменяются пустым указателем `NULL`. Рассмотрим саму структуру:

```
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);
};

```

Обратите внимание на название функций: они аналогичны знакомым нам системным вызовам. Представим, что вы желаете использовать не все функции, а только `read`, `write`, `open` и `release`, тогда структура `file_operations` может быть заполнена так:

```

struct file_operations fops = {
    .read = mydevice_read,
    .write = mydevice_write,
    .open = mydevice_open,
    .release = mydevice_release
};

```

Обратите внимание на название структуры — `fops`. Указатель на структуру типа `file_operations` обычно называется именно `fops` — так программисты договорились между собой. Конечно, вы можете использовать и другое название — ничего страшного не случится. Но когда в Интернете

встретите фрагмент кода с указателем `fops`, знайте, это указатель на структуру `file_operations`.

Каждое устройство в системе представлено в виде файла. Именно поэтому структура операций над устройством называется `file_operations`. А само устройство (т. е. его файл) описывается структурой `file`, которая тоже представлена в `linux/fs.h`. Указатель на структуру `file` обычно называется `filp`.

Регистрация устройства

Мало создать файл устройства. Нужно его еще связать с самим устройством, т. е. зарегистрировать в ядре. Регистрация говорит ядру, что данный файл устройства будет обслуживаться таким-то драйвером.

При добавлении драйвера происходит его регистрация в ядре и получение старшего номера драйвера. Для регистрации драйвера символьного устройства используется функция `register_chrdev()`, определенная в `linux/fs.h`:

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

Первый параметр — запрашиваемый старший номер устройства, второй — имя файла устройства, третий — указатель на структуру типа `file_operations`. Функции не передается младший номер устройства — ведь это не забота ядра, об обслуживании устройств и назначении им младших номеров должен заботиться сам драйвер.

Как узнать, какой старший номер не используется? Существуют два способа. Первый заключается в использовании динамического номера — просто задайте 0 в качестве первого параметра и ядро само назначит вашему драйверу первый неиспользуемый номер. Второй способ менее правильный — загляните в `Documentation/devices.txt`, посмотрите, какие номера устройств заняты, и установите незанятый номер.

У этого способа есть огромный недостаток: если вы выберете номер, скажем, 77, а потом этот номер будет официально закреплен за другим устройством, ваш драйвер работать не будет.

В случае с динамическим номером тоже есть один неприятный момент: вы не знаете наперед, какой номер устройства будет вам выделен, следовательно, не можете создать файл устройства (ведь при создании нужно указать старший номер устройства).

Лучшее решение этой проблемы — использовать системный вызов `mknod()` для создания файла устройства — сразу после получения старшего номера устройства от ядра. Только не забудьте в функции `cleanup_module()` удалить файл устройства.

Драйвер абстрактного символьного устройства

Сейчас мы напишем драйвер абстрактного (несуществующего в природе) символьного устройства. При запуске модуля мы получим старший номер устройства и запишем его в системный журнал `/var/log/messages`. Дабы не усложнять код модуля, мы не будем вызывать системный вызов `mknod()`: после запуска модуля вы вручную создадите устройство, указав полученный старший номер как аргумент команды `mknod`.

Наш модуль будет поддерживать операции открытия, чтения и освобождения устройства. Операция записи устройства будет поддерживаться частично: вместо полноценной функции будет заглушка.

Код модуля:

```

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

/* Для большего удобства определяем прототипы функций */
int init_module(void);
void cleanup_module(void);
static int mydevice_open(struct inode *, struct file *);
static int mydevice_release(struct inode *, struct file *);
static ssize_t mydevice_read(struct file *, char *, size_t, loff_t *);
static ssize_t mydevice_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "mydev"      /* Имя устройства */
#define BUF_LEN 128              /* Максимальная длина сообщения */

static int Major;                /* Старший номер устройства */
static int Device_Status = 0;    /* Статус устройства, 1 = открыто */
static char msg[BUF_LEN];        /* Буфер для сообщения */
static char *msg_Ptr;

/* Структура операций */
static struct file_operations fops = {
    .open = mydevice_open,
    .release = mydevice_release,
    .read = mydevice_read,
    .write = mydevice_write
};

/* Инициализация модуля */
int init_module(void)
{
    /* Пытаемся зарегистрировать устройство */
    Major = register_chrdev(0, DEVICE_NAME, &fops);

    if (Major < 0) {
        printk(KERN_ALERT "register_chrdev() error %d\n", Major);
        return Major;
    }

    printk(KERN_NOTICE "Major number %d\n", Major);
    printk(KERN_NOTICE "Please create a dev file with\n");
    printk(KERN_NOTICE "'mknod /dev/mydev c %d 0'.\n", Major);

```

```

    return SUCCESS;
}

void cleanup_module(void)
{
    /* Удаляем устройство */
    int res = unregister_chrdev(Major, DEVICE_NAME);
    if (res < 0)
        printk(KERN_ALERT "unregister_chrdev() error: %d\n", res);
}

/* Обработчики устройства */

/* Обработчик открытия устройства */
static int mydevice_open(struct inode *inode, struct file *file)
{
    if (Device_Status)
        return -EBUSY;

    Device_Open++;
    sprintf(msg, "Hello\n");
    msg_Ptr = msg;
    try_module_get(THIS_MODULE);

    return SUCCESS;
}

/* Обработчик закрытия устройства */
static int mydevice_release(struct inode *inode, struct file *file)
{
    Device_Status--;
    module_put(THIS_MODULE);
    return SUCCESS;
}

/* Обработчик записи устройства */
static ssize_t
mydevice_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
    printk(KERN_ALERT "Unsupported operation: write()\n");
    return -EINVAL;
}

/* Обработчик чтения устройства — вызывается, когда процесс пытается прочитать
уже открытый файл.
filp — указатель на структуру file
buffer — буфер, в который нужно занести данные
length — длина буфера
offset — смещение */

```

```

mystatic ssize_t mydevice_read(struct file *filp, char *buffer, size_t length,
loff_t * offset)
{

    int bytes_read = 0; /* Сколько байтов записано в буфер */

    if (*msg_Ptr == 0)          /* Достигнут конец сообщения,
                                возвращаем 0 как признак конца файла */

        return 0;

    /* Самый важный момент: перемещение данных. Поскольку буфер
    находится в сегменте данных пользовательского процесса
    (в пространстве пользователя), а не в пространстве ядра,
    то мы не можем выполнить простое присваивание. Чтобы
    скопировать данные, нам нужно использовать функцию put_user(),
    которая перенесет данные из пространства ядра в пространство пользователя
    */
    while (length && *msg_Ptr) {

        put_user(*(msg_Ptr++), buffer++);

        length--;
        bytes_read++;
    }

    /* Возвращаем количество реально записанных в буфер байтов */
    return bytes_read;
}

MODULE_LICENSE("GPL");                /* Лицензия */
MODULE_AUTHOR("Denis Kolisnichenko"); /* Автор */
MODULE_DESCRIPTION("Driver for /dev/mydev"); /* Описание */
MODULE_SUPPORTED_DEVICE("mydev");      /* Устройство */

```

Давайте разберем наш модуль по кирпичику. Мы подключаем заголовочные файлы:

```

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

```

Первые два стандартны для любого модуля. В третьем находится структура `file_operations()`, необходимая для переопределения обработчиков устройства. В последнем

находится функция `put_user()`, передающая данные из пространства ядра в пространство пользователя. Далее объявляем три константы и четыре статических переменных:

```
#define SUCCESS 0
#define DEVICE_NAME "mydev"          /* Имя устройства */
#define BUF_LEN 128                  /* Максимальная длина сообщения */

static int Major;                    /* Старший номер устройства */
static int Device_Status = 0;        /* Статус устройства, 1 = открыто */
static char msg[BUF_LEN];            /* Буфер для сообщения */
static char *msg_Ptr;
```

Первая константа нужна только для красоты — вместо нее можно было бы просто передавать 0, если вызов обработчика устройства завершился успешно.

Вторая константа задает имя устройства, а третья — длину буфера сообщений.

В переменной `Major` будет храниться старший номер устройства. Переменная `Device_Status` хранит статус устройства: 1 — открыто (уже используется каким-то процессом), 0 — свободно. Переменная `msg` содержит буфер для сообщения, которое получит процесс, когда попытается прочитать устройство `/dev/mydev`. Последняя переменная — указатель на буфер.

При инициализации устройства мы получаем его старший номер — заполняем переменную `Major`. Если зарегистрировать устройство не удалось, модуль прекращает свою работу — функция инициализации возвращает значение, отличное от 0, а именно — код возврата функции `register_chrdev()`. Если же все прошло успешно, модуль сообщает старший номер устройства, чтобы вы могли создать само устройство командой `mknod`. Обратите внимание: критические ошибки (когда модуль больше не может продолжить работу) выводятся как `KER_ALERT` (будут видны на консоли), а все остальные сообщения — как `KERN_NOTICE` (будут видны только в системном журнале):


```

Major = register_chrdev(0, DEVICE_NAME, &fops);

if (Major < 0) {
    printk(KERN_ALERT "register_chrdev() error %d\n", Major);
    return Major;
}

printk(KERN_NOTICE "Major number %d\n", Major);
printk(KERN_NOTICE "Please create a dev file with\n");
printk(KERN_NOTICE "'mknod /dev/mydev c %d 0'.\n", Major);

```

При завершении работы модуля мы должны удалить регистрацию устройства (файл, созданный командой `mknod`, останется в каталоге `/dev`, но связь между файлом устройства и драйвером будет аннулирована):

```
int res = unregister_chrdev(Major, DEVICE_NAME);
```

После определения двух главных функций модуля можно приступить к написанию обработчиков устройства, которые объявлены в структуре `file_operations`:

```

static struct file_operations fops = {
    .open = mydevice_open,
    .release = mydevice_release,
    .read = mydevice_read,
    .write = mydevice_write
};

```

Прежде всего, разберемся, когда вызывается тот или иной обработчик:

- ❑ `mydevice_open()` — вызывается, когда процесс пытается открыть файл;
- ❑ `mydevice_release()` — вызывается при закрытии файла;
- ❑ `mydevice_read()` — вызывается, когда процесс пытается прочитать файл устройства;

□ `write()` — когда процесс пытается записать что-то в файл устройства, например

```
echo info > /dev/mydev.
```

При открытии устройства мы должны проверить его статус: если он равен 1, значит, устройство уже используется другим процессом, и мы возвращаем значение `-EBUSY`: устройство занято.

Если же статус устройства равен 0, тогда мы увеличиваем статус (все — с этого момента устройство занято) и заполняем буфер текстовой строкой.

```
if (Device_Status)
    return -EBUSY;

Device_Open++;

sprintf(msg, "Hello\n");

msg_Ptr = msg;
```

При закрытии устройства мы должны установить статус устройства в 0:

```
Device_Status--;
```

Функция записи вообще элементарна — это просто заглушка. Мы выводим сообщение о том, что операция записи не поддерживается. На этом вся запись заканчивается. А вот функция чтения устройства заслуживает внимания. Обычно все функции чтения возвращают количество прочитанных байтов, что будет делать и наша функция, поэтому она начинается с переменной `bytes_read`. Далее в цикле `while` мы посимвольно передаем содержимое нашего буфера пользовательскому процессу с помощью функции `put_user()`. Также в цикле `while` увеличивается переменная `bytes_read`, которая потом и возвращается пользовательскому процессу:

```

int bytes_read = 0;          /* Сколько байтов записано в буфер */

if (*msg_Ptr == 0)          /* Достигнут конец сообщения, возвращаем 0
                             как признак конца файла */
    return 0;

while (length && *msg_Ptr) {

    put_user(*(msg_Ptr++), buffer++);

    length--;
    bytes_read++;
}

return bytes_read;

```

Вот теперь все стало на свои места. Попробуем усложнить наш модуль, добавив поддержку записи. Для этого нужно изменить всего лишь одну функцию — обработчик записи устройства. Если функция чтения устройства передавала пользовательскому процессу информацию с помощью функции `put_user()`, то сейчас мы будем получать информацию от пользовательского процесса с помощью функции `get_user()`:

```

static ssize_t
mydevice_write(struct file *file,
               const char __user * buffer, size_t length, loff_t * offset)
{
    int bwrite;

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

    Message_Ptr = Message;

    /* Возвращаем к-во принятых байтов */
    return bwrite;
}

```

Все бы хорошо, и наше абстрактное виртуальное устройство отлично работает. Но на практике вам придется столкнуться с управлением самим физическим устройством. В зависимости от

специфики устройства, возможно, вам придется использовать функцию `ioctl()`, ее название является сокращением от Input/Output ConTroL.

Поскольку я не знаю специфики вашего устройства, приводить какой-либо код — глупо. Возможно, он подойдет одному читателю из 1000, а может, наоборот, будет интересен 1000 читателям, но не лично вам. При использовании `ioctl` структура операций будет выглядеть так:

```
struct file_operations fops = {  
    .read = mydevice_read,  
    .write = mydevice_write,  
    .ioctl = mydevice_ioctl,  
    .open = mydevice_open,  
    .release = mydevice_release,  
};
```

Пример 3: Создание файла в /proc

Файловая система `/proc` используется для предоставления информации о процессах и о системе, например в файле `/proc/modules` содержится список загруженных модулей. Некоторые файлы в `/proc` поддерживают не только запись, но и чтение и могут даже использоваться для настройки системы на лету, но сейчас нас такие файлы не интересуют.

Попробуем создать файл `/proc/mydev`, предоставляющий информацию о нашем устройстве.

Для работы с `/proc` нужно подключить заголовочный файл `proc_fs.h`. Но поскольку мы все еще разрабатываем модуль ядра, то нам нужны файлы `module.h` и `kernel.h`:

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

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

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

Для работы с `/proc` нам нужна структура `proc_dir_entry`:

```
struct proc_dir_entry *pde;
```

Структура `proc_dir_entry` заполняется так:

```

/* имя файла и права доступа */
pde = create_proc_entry("mydev", 0644, NULL);
pde->read_proc = proc_read;          /* Обработчик чтения */
pde->owner = THIS_MODULE;             /* Владелец прос-файла */
pde->mode = S_IFREG | S_IRUGO;        /* Режим */
pde->uid = 0;                         /* UID */
pde->gid = 0;                         /* GID */
pde->size = 50;                       /* Размер */

```

Теперь рассмотрим функцию `proc_read()` — обработчик чтения прос-файла:

```

ssize_t
proc_read(char *buffer,
           char **buffer_location,
           off_t offset, int buffer_length, int *eof, void *data)

```

Ей нужно передать шесть параметров:

- ☐ буфер с данными — сюда можно записать все что угодно;
- ☐ указатель на строку с данными — если вы не хотите использовать параметр `buffer`;
- ☐ смещение — текущая позиция в файле;
- ☐ длина буфера — собственно, длина как она есть;
- ☐ признак конца файла — если `eof = 1`, достигнут конец файла;
- ☐ указатель на данные — нужен, только если у вас один обработчик, а прос- файлов — несколько.

Код функции `proc_read` может быть таким:

```

ssize_t
proc_read(char *buffer,
           char **buffer_location,
           off_t offset, int buffer_length, int *eof, void *data)
{
    int length = 0;          /* Число байт */
    static int c = 1;        /* Сколько раз был прочитан файл */

    /* Достигнут конец файла, нужно сообщить об этом процессу,
    читающему файл */

```

```

-----v-----
if (offset > 0) {
    printk(KERN_INFO "Offset %d, Bytes %d\n", (int) (offset), length);
    *eof = 1;
    return length;
}

/* Заполняем буфер */
length = sprintf(buffer,
    "This file has been read %d times\n", c);
c++;

return length;
}

```

Функция инициализации модуля заполняет структуру pde:

```

int init_module()
{
    int res = 0;
    pde = create_proc_entry("mydev", 0644, NULL);
    pde->read_proc = proc_read;
    pde->owner = THIS_MODULE;
    pde->mode = S_IFREG | S_IRUGO;
    pde->uid = 0;
    pde->gid = 0;
    pde->size = 50;

    printk(KERN_INFO "Module started. Creating /proc/mydev...");

    if (pde == NULL) {
        res = -1;
        remove_proc_entry("mydev", &proc_root);
        printk(KERN_INFO "Error\n");
    } else
        printk(KERN_INFO "OK\n");

    return res;
}

```

Если вызов `create_proc_entry` не удался, то мы удаляем наш прос-файл вызовом `remove_proc_entry()` и завершаем работу модуля. Функция `cleanup_module()` очень проста и выглядит так:

```

void cleanup_module()

```

```

{
remove_proc_entry("mydev", &proc_root);
printk(KERN_INFO "Module stopped\n");
}

```

Теперь нам осталось собрать все вместе (листинг 13.5).

Листинг 13.5. Модуль procfs

```

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>

struct proc_dir_entry *pde;

ssize_t
proc_read(char *buffer, char **buffer_location,
          off_t offset, int buffer_length, int *eof, void *data)
{
    int length = 0;          /* Число байт */
    static int c = 1;        /* Сколько раз был прочитан файл */

    /* Достигнут конец файла, нужно сообщить об этом процессу,
    читающему файл */
    if (offset > 0) {
        printk(KERN_INFO "Offset %d, Bytes %d\n", (int)(offset), length);
        *eof = 1;
        return length;
    }

    /* Заполняем буфер */
    length = sprintf(buffer,
                     "This file has been read %d times\n", c);
    c++;

    return length;
}

int init_module()
{
    int res = 0;
    pde = create_proc_entry("mydev", 0644, NULL);
    pde->read_proc = proc_read;
    pde->owner = THIS_MODULE;
    pde->mode = S_IFREG | S_IRUGO;
}

```

```

pde->uid = 0;
pde->gid = 0;
pde->size = 50;

printk(KERN_INFO "Module started. Creating /proc/mydev...");

if (pde == NULL) {
    res = -1;
    remove_proc_entry("mydev", &proc_root);
    printk(KERN_INFO "Error\n");
} else
    printk(KERN_INFO "OK\n");

return res;
}

void cleanup_module()
{
    remove_proc_entry("mydev", &proc_root);
    printk(KERN_INFO "Module stopped\n");
}

```

Скомпилируйте и установите модель (командой `insmod`). После этого загляните в каталог `/proc` — в нем будет файл `mydev`. Откройте его и просмотрите содержимое.

Практические задания

1. Проследите в своей системе какие макросы, в какой последовательности и в каких файлах вызывает простейший сценарий Makefile сборки модуля:

TARGET = hello_printk

obj-m := \$(TARGET).o

default:

\$(MAKE) -C \$(KDIR) M=\$(PWD) modules

Опишите подробно что происходит на 2-х последовательных стадиях сборки модуля.

2. Соберите простой модуль, который бы предусматривал **все** возможные типы допускаемых параметров загрузки в командной строке. Рассмотрите (проанализируйте) реакцию модуля на

ошибочные действия пользователя (неправильные вводимые параметры). Покажите различия в реакции на ошибки в разных версиях ядра.

3. Подсчитайте общее число **функций** API ядра (kernel API) в вашем дистрибутиве Linux.

4. В качестве примера, покажите и перечислите все функции строчной обработки API ядра, имеющие вид str*().

5. Напишите модуль, который должен зафиксировать в **системный журнал** значение счётчика системного таймера ядра jiffies, и сразу же завершается.

6. (*) Напишите модуль, подобный предыдущему, но чтобы он выводил значение jiffies не в системный журнал, а **на терминал**. Почему в API ядра отсутствует функция вывода на терминал? На какой терминал (прикреплённый терминал какого задания) происходит вызов?

7. В последних версиях ядра (3.13 и далее) собираемые вами модули при загрузке будут создавать сообщения (в системный журнал, dmesg) вида:

```
confm: module verification failed: signature and/or required key missing - tainting kernel
```

Выясните природу их происхождения, предназначение, и проблемы, которые могут порождаться.

8. Напишите драйвер символьного устройства (любой способ регистрации), который может писать в статический буфер (достаточно большого размера, скажем 1024) и затем читать оттуда записанное значение. Добейтесь, чтобы устройство допускало чтение и запись для любых пользователей (флаги доступа 0666). Предполагаем, что драйвер предназначен только для **символьных** (ASCIZ) данных.

9. (*) Напишите драйвер символьного устройства, подобный предыдущему, которое работает по принципу очереди: записываемые данные помещаются в конец очереди, а считываемые берутся из головы, и удаляются из очереди. Для организации очереди используйте структуру struct list_head,

использую повсеместно в ядре для организации динамических структур. Предусмотрите изменяющиеся, в том числе и весьма большие, объёмы данных, помещённых в очередь (файлы данных). Предполагаем, что драйвер предназначен для записи-чтения любых **бинарных** данных — проверьте работоспособность записью с контрольным считыванием самого файла полученного модуля .ko.