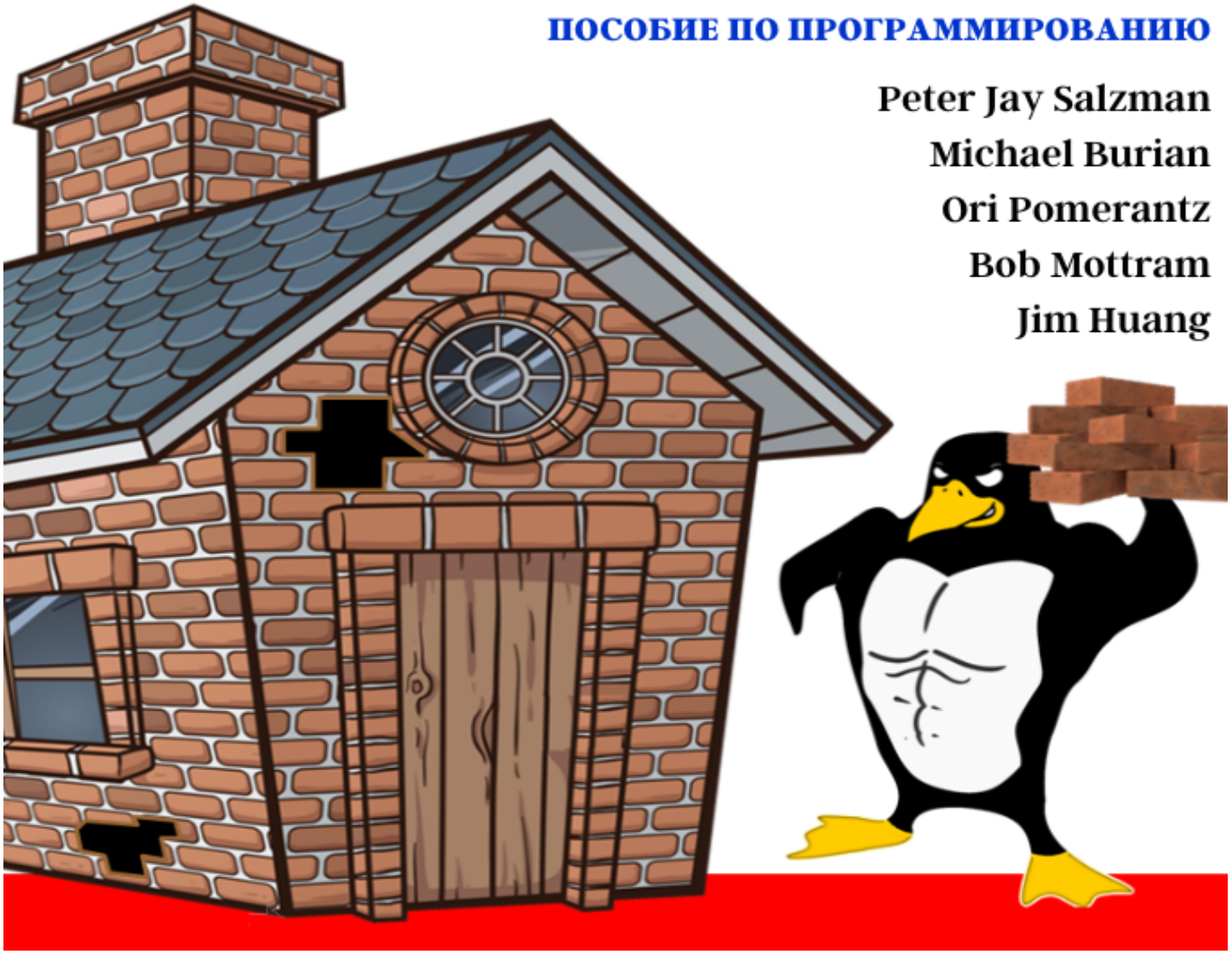
Linux
Модули ядра

ПОСОБИЕ ПО ПРОГРАММИРОВАНИЮ

Peter Jay Salzman
Michael Burian
Ori Pomerantz
Bob Mottram
Jim Huang



Авторы: Peter Jay Salzman, Michael Burian, Ori Pomerantz, Bob Mottram, Jim Huang

Ссылка на оригинал: The Linux Kernel Module Programming Guide

Перевод: команда RUVDS.com

ОГЛАВЛЕНИЕ

1. Вступление

Эта книга задумана для распространения в качестве полезного подручного материала, но не предоставляет никаких гарантий, в том числе гарантий соответствия ожиданиям читателя или пригодности для конкретной цели. Её автор призывает к активному распространению этого материала для личного и коммерческого использования при условии соответствия вышеуказанной лицензии. Проще говоря, вы можете копировать и распространять эту книгу как бесплатно, так и за деньги. Делать это можно и в электронном, и в физическом формате, не спрашивая дополнительного разрешения у автора.

Производные работы и переводы этого документа должны также публиковаться под лицензией Open Software License с упоминанием оригинального источника. Если вы внесёте в книгу новый материал, то сделайте этот материал и исходный код доступными для своих ревизий. Ревизии и обновления должны быть доступны непосредственно мейнтейнеру документа, Джиму Хуангу <jserv@ccns.ncku.edu.tw>. Это позволит делать мерджи обновлений и предоставлять согласованные ревизии сообществу Linux.

Если вы будете публиковать или распространять книгу в коммерческих целях, то автор и проект документирования Linux (LDP) будут весьма признательны за пожертвования, роялти-отчисления и предоставление печатных версий. Участие в общем деле таким образом показывает вашу поддержку бесплатного ПО и LDP. По всем вопросам можете писать на приведённый выше адрес.

1.1 Авторство

Первое «Руководство по программированию модулей ядра» написал Ори Померанц для ядер версии 2.2. В конечном итоге у Ори не стало времени для поддержания актуальности этого документа, что не удивительно, ведь ядро очень динамично в своём развитии. После этого поддержку руководства взял на себя Питер Джей Зальцман, который обновил его под версию 2.4. Но в итоге и Питеру стало не хватать времени, чтобы довести пособие до соответствия ядру 2.6. В этой ситуации на выручку пришёл Майкл Буриан, который помог его обновить. Следующим был Боб Моттрам, доработавший примеры под ядро 3.8+. Последним же на данный момент является Джим Хуанг, который довёл руководство до соответствия последним версиям ядра 5.x и скорректировал документ LaTeX.

1.2 Благодарности

Ниже перечислены сторонние участники, которые вносили изменения и давали полезные рекомендации:

2011eric, 25077667, Arush Sharma, asas1asas200, Benno Bielmeier, Bob Lee, Brad Baker, ccs100203, Chih-Yu Chen, Ching-Hua (Vivian) Lin, ChinYikMing, Cyril Brulebois, Daniele Paolo Scarpazza, David Porter, demonsome, Dimo Velev, Ekang Monyet, fennecJ, Francois Audeon, gagachang, Gilad Reti, Horst Schirmeier, Hsin-Hsiang Peng, Ignacio Martin, JianXing Wu, linD026, lyctw, manbing, Marconi Jiang, mengxinayan, RinHizakura, Roman Lakeev, Stacy Prowell, Steven Lung, Tristan Lelong, Tucker Polomik, VxTeemo, Wei-Lun Tsai, xatier, Ylowy.

1.3 Что такое модуль ядра?

Итак, вы хотите написать модуль для ядра. Вы знаете Си, уже написали несколько неплохих программ для выполнения в качестве процессов, и теперь вам захотелось попасть на территорию реального экшена, где единственный недействительный указатель может стереть всю вашу файловую систему, а дамп памяти означает перезагрузку.

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.

1.4 Пакеты модулей ядра

В дистрибутивах Linux для работы с пакетами модулей есть команды `modprobe`, `insmod` и `depmod`.

В Ubuntu/Debian:

```
sudo apt-get install build-essential kmod
```

В Arch Linux:

```
sudo pacman -S gcc kmod
```

1.5 Какие модули содержатся в моём ядре?

Узнать, какие модули загружены в ядро, можно командой `lsmod`:

```
sudo lsmod
```

Хранятся модули по пути */proc/modules*, значит, их также можно просмотреть с помощью:

```
sudo cat /proc/modules
```

Список может оказаться длинным, и вам потребуется искать что-то конкретное. Вот пример поиска модуля `fat`:

```
sudo lsmod | grep fat
```

1.6 Нужно ли скачивать и компилировать ядро?

Для целей, связанных с этим руководством, это делать не обязательно. Однако будет мудрым решением выполнять примеры в тестовом дистрибутиве на виртуальной машине, чтобы избежать возможного нарушения работы системы.

1.7 Перед началом

Прежде чем переходить к коду, нужно разобрать кое-какие нюансы. Поскольку у каждого своя система и свои настройки, иногда компиляция и корректная загрузка вашей программы могут вызывать сложности.

Но будьте уверены, после первого разрешения всех возможных трудностей, дальнейший полёт будет гладким.

1. ***Версионирование модулей.*** Модуль, скомпилированный для одного ядра, не загрузится для другого, если не включить в этом ядре `CONFIG_MODVERSIONS`. Подробнее о версионировании мы ещё поговорим позднее. Если в вашем ядре версионирование включено, то поначалу, пока мы не разберём эту тему подробнее, примеры могут у вас не работать. Правда, включено оно обычно в большинстве базовых дистрибутивов, и если из-за этого у вас возникнут проблемы с загрузкой модулей, скомпилируйте ядро, отключив их версионирование.

2. ***Использование X Window System.*** Настоятельно рекомендуем извлекать, компилировать и загружать все приводимые в руководстве примеры из консоли. Работать с ними в X Window System не стоит.

Модули не могут выводить информацию на экран подобно `printf()`. При этом они могут логировать информацию и предупреждения, которые в итоге на экран выводятся, но только в консоли. Если вы вставите модуль (`insmod`) из xterm, то информация и предупреждения залогируются, но только в системный журнал.

То есть увидеть вы все эти данные сможете лишь через `journalctl`. Подробности описаны в разделе 4. Для получения прямого доступа ко всей этой информации, выполняйте все действия в консоли.

2. Заголовочные файлы

Прежде чем вы сможете что-либо создавать, вам нужно установить для ядра заголовочные файлы.

в Ubuntu/Debian:

```
sudo apt-get update
apt-cache search linux-headers-`uname -r`
```

в Arch Linux:

```
sudo pacman -S linux-headers
```

Так вы узнаете, какие заголовочные файлы ядра доступны. Затем можно выполнить, например:

```
sudo apt-get install kmod linux-headers-5.4.0-80-generic
```

3. Примеры

Все примеры этого документа доступны в подкаталоге *examples*. Если возникнут ошибки компиляции, причиной может быть то, что у вас установлена более свежая версия ядра, или же просто недостаёт необходимых заголовочных файлов.

4. Hello World

4.1 Простейший модуль

Большинство людей, изучающих программирование, начинают с какого-нибудь примера «Hello world». Не знаю, что бывает с теми, кто от этой традиции отходит, но, думаю, лучше и не знать. Мы начнём с серии программ «Hello world», которые продемонстрируют различные основы написания модуля ядра.Ниже описан простейший пример модуля.

Создайте тестовый каталог:

```
mkdir -p ~/develop/kernel/hello-1
cd ~/develop/kernel/hello-1
```

Вставьте следующий код в редактор и сохраните как *hello-1.c*:

```
/*
 * hello-1.c – простейший модуль ядра.
 */
#include <linux/kernel.h> /* необходим для pr_info() */
#include <linux/module.h> /* необходим для всех модулей */

int init_module(void)
{
    pr_info("Hello world 1.\n");

    /* Если вернётся не 0, значит, init_module провалилась; модули загрузить не
получится. */
    return 0;
}

void cleanup_module(void)
{
    pr_info("Goodbye world 1.\n");
}

MODULE_LICENSE("GPL");
```

Теперь вам потребуется Makefile.

Если вы будете копировать следующий код, то сделайте отступы табами, не пробелами:

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

PWD := $(CURDIR)

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

В Makefile инструкция `$(CURDIR)` может быть установлена на абсолютный путь текущего рабочего каталога (затем идёт обработка всех опций `-C`, если таковые присутствуют).

Подробнее о `CURDIR` читайте в мануале GNU make.

В завершении просто выполните `make`.

```
make
```

Если в Makefile не будет инструкции `PWD := $(CURDIR)`, он может не скомпилироваться корректно с помощью `sudo make`. Поскольку некоторые переменные среды регулируются политикой безопасности, наследоваться они не могут. По умолчанию эта политика определяется файлом *sudoers*. В нём изначально включена опция `env_reset`, которая запрещает переменные среды.

В частности, переменные `PATH` из пользовательской среды не сохраняются, а устанавливаются на значения по умолчанию (подробнее можно почитать в мануале по `sudoers`).

Установки для переменных среды можно посмотреть так:

```
$ sudo -s
# sudo -V
```

Вот пример простого Makefile, демонстрирующий описанную выше проблему:

```
all:
    echo $(PWD)
```

Далее можно использовать флаг `-p` для вывода всех значений переменных среды из Makefile:

```
$ make -p | grep PWD
PWD = /home/ubuntu/temp
OLDPWD = /home/ubuntu
    echo $(PWD)
```

Переменная `PWD` при выполнении `sudo` унаследована не будет.

```
$ sudo make -p | grep PWD
    echo $(PWD)
```

Тем не менее эту проблему можно решить тремя способами:

1. Использовать флаг `-E` для их временного сохранения:

```
$ sudo -E make -p | grep PWD
```

```
    PWD = /home/ubuntu/temp
    OLDPWD = /home/ubuntu
    echo $(PWD)
```

2. Отключить `env_reset`, отредактировав */etc/sudoers* из-под рут-пользователя с помощью `visudo`:

```
## файл sudoers.
  ##
  ...
  Defaults env_reset
  ## В предыдущей строке измените env_reset на !env_reset, чтобы сохранить все
переменные среды.
```

Затем выполните `env` и `sudo env` по отдельности:

```
# отключить env_reset
    echo "user:" > non-env_reset.log; env >> non-env_reset.log
    echo "root:" >> non-env_reset.log; sudo env >> non-env_reset.log
    # включить env_reset
    echo "user:" > env_reset.log; env >> env_reset.log
    echo "root:" >> env_reset.log; sudo env >> env_reset.log
```

Можете просмотреть и сравнить эти логи, чтобы понять отличия между `env_reset` и `!env_reset`.

3. Сохранить переменные среды, добавив их в `env_keep` в */etc/sudoers*.

```
 Defaults env_keep += "PWD"
```

После применения этого изменения можете проверить установки переменных сред с помощью:

```
        $ sudo -s
         # sudo -V
```

Если всё пройдёт гладко, вы получите скомпилированный модуль *hello-1.ko*.

Информацию о нём можно вывести командой:

```
modinfo hello-1.ko
```

На этом этапе команда:

```
sudo lsmod | grep hello
```

не должна ничего возвращать. Можете попробовать загрузить свой новоиспечённый модуль с помощью:

```
sudo insmod hello-1.ko
```

При этом символ тире превратится в нижнее подчёркивание. Теперь, когда вы снова выполните:

```
sudo lsmod | grep hello
```

то увидите загруженный модуль. Удалить его можно с помощью:

```
sudo rmmod hello_1
```

Обратите внимание — тире было заменено нижним подчёркиванием. Чтобы увидеть произошедшее в логах, выполните:

```
sudo journalctl --since "1 hour ago" | grep kernel
```

Теперь вам известны основы создания, компиляции, установки и удаления модулей. Далее мы подробнее разберём, как они работают.

Модули ядра должны иметь не менее двух функций:

- **стартовую** (инициализация), которая называется `init_module()` и вызывается при внедрении (insmod) модуля в ядро;
- **завершающую** (очистка), которая зовётся `cleanup_module()` и вызывается непосредственно перед извлечением модуля из ядра.

В действительности же с версии 2.3.13 произошли кое-какие изменения. Теперь стартовую и завершающую функцию модулей можно называть на своё усмотрение, и об этом будет подробнее сказано в разделе 4.2. На деле этот новый метод даже предпочтительней, хотя многие по прежнему используют названия `init_module()` и `cleanup_module()`.

Как правило, `init_module()` или регистрирует обработчик чего-либо с помощью ядра, или заменяет одну из функций ядра собственным кодом (обычно кодом, который

выполняет определённые действия и вызывает исходную функцию). Что касается `cleanup_module()`, то она должна отменять всё, что сделала `init_module()`, чтобы безопасно выгрузить модуль.

Наконец, каждый модуль ядра должен включать `<linux/module.h>`. Нам нужно было включить `<linux/kernel.h>` только для расширения макроса уровня журнала `pr_alert()`, о чём подробнее сказано в пункте 2.

1. ***Примечание о стиле написания кода.*** Есть нюанс, который может не быть очевиден тем, кто только начинает заниматься программированием ядра. Имеется в виду то, что отступы в коде должны делаться с помощью табов, а не пробелов. Это одно из общих соглашений. Оно вам может не нравиться, но придётся привыкать, если вы соберётесь отправлять патч в основную ветку ядра.

2. ***Добавление макросов вывода.*** Изначально использовалась функция `printk`, обычно сопровождаемая приоритетом уровня журнала `KERN_DEBUG` или `KERN_INFO`. Недавно же появилась возможность выражать это в сокращённой форме с помощью макросов вывода `pr_info` и `pr_debug`. Такой подход просто избавляет от лишних нажатий клавиш и выглядит более лаконично. Найти эти макросы можно в *include/linux/printk.h*. Рекомендую уделить время и прочесть о доступных макросах приоритетов.

3. ***Насчёт компиляции.*** Модули ядра нужно компилировать несколько иначе, нежели обычные приложения пространства пользователя. Прежние версии ядра требовали от нас особого внимания к этим настройкам, которые обычно хранились в Makefile. И несмотря на иерархическую организованность, в make-файлах нижнего уровня скапливалось множество лишних настроек, что делало эти файлы большими и усложняло их обслуживание. К счастью, появился новый способ делать всё это, который называется `kbuild`, и процесс сборки для внешних загружаемых модулей теперь полностью интегрирован в стандартный механизм сборки ядра. Подробнее о компиляции модулей, не являющихся частью официального ядра (таких как примеры в этом руководства), читайте в *Documentation/kbuild/modules.rst*.

Дополнительные подробности о make-файлах для модулей ядра доступны в *Documentation/kbuild/makefiles.rst*. Обязательно прочтите эту документацию и изучите связанные с ней файлы – это наверняка избавит вас от большого объёма лишней работы.

А вот вам одно бонусное упражнение. Видите комментарий над инструкцией `return` в `init_module()`? Измените возвращаемое значение на отрицательное, после чего перекомпилируйте и заново загрузите модуль. Что произойдёт?

4.2 Hello и Goodbye

В ранних версиях ядра вам нужно было использовать функции `init_module` и `cleanup_module`, как в нашем первом примере «Hello world», но сегодня их уже можно именовать на своё усмотрение с помощью макросов `module_init` и `module_exit`, которые определены в *include/linux/module.h*. Единственное требование – это чтобы функции инициализации и очистки были определены до вызова этих макросов, в противном случае возникнут ошибки компиляции.

Вот пример:

```
/*
 * hello-2.c - демонстрация макросов module_init() и module_exit().
 * Этот вариант предпочтительнее использования init_module() и cleanup_module().
 */
#include <linux/init.h> /* Необходим для макросов */
#include <linux/kernel.h> /* Необходим для pr_info() */
#include <linux/module.h> /* Необходим всем модулям */

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

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

module_init(hello_2_init);
module_exit(hello_2_exit);

MODULE_LICENSE("GPL");
```

Теперь у нас есть уже два реальных модуля ядра. Добавить ещё один будет совсем несложно:

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

```
PWD := $(CURDIR)

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Загляните в *drivers/char/Makefile*, чтобы увидеть реальный пример. Как видите, некоторые элементы включаются в ядро жёстко (`obj-y`), но куда делись все `obj-m`? Те, кто знаком со скриптами оболочки, смогут без проблем их обнаружить.

Для остальных подскажу, что записи `obj-$(CONFIG_FOO)`, которые вы видите повсюду, расширяются на `obj-y` или `obj-m` в зависимости от того, на какое значение была установлена переменная `CONFIG_FOO` — `y` или `m`.

Попутно отмечу, что именно эти переменные вы установили в файле *.config* в каталоге верхнего уровня дерева исходного кода в последний раз, когда выполнили `make menuconfig` или что-то в том духе.

4.3 Макросы __init и __exit

Макрос `__init` приводит к отбрасыванию функции инициализации и освобождению занятой ей памяти по завершении её выполнения для встроенных драйверов, но не загружаемых модулей. И это вполне разумно, если учесть, когда эта функция вызывается.

Также есть `__initdata`, которая работает аналогично `__init`, но для переменных инициализации, а не для функций.

Макрос `__exit` приводит к пропуску функции, если модуль встроен в ядро, то есть аналогично `__init` не влияет на загружаемые модули.

Опять же, если учесть, когда выполняется функция очистки, то это полностью оправданно. Встроенным драйверам не требуется очистка, а вот загружаемым модулям как раз да.

Эти макросы определены в *include/linux/init.h* и используются для освобождения памяти ядра.

Если при его загрузке вы видите сообщение вроде `Freeing unused kernel memory: 236k freed`, то знайте – это тот самый процесс.

```
/*
 * hello-3.c – демонстрация макросов __init, __initdata и __exit.
```

```
 */
#include <linux/init.h> /* Необходим для макросов */
#include <linux/kernel.h> /* Необходим для pr_info() */
#include <linux/module.h> /* Необходим для всех модулей */

static int hello3_data __initdata = 3;

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

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

module_init(hello_3_init);
module_exit(hello_3_exit);

MODULE_LICENSE("GPL");
```

4.4 Лицензирование и документирование модулей

Даже не знаю, кто вообще загружает или вообще задумывается об использовании проприетарных модулей? Если вы из числа таких людей, то наверняка видели нечто подобное:

```
$ sudo insmod xxxxxx.ko
loading out-of-tree module taints kernel.
module license 'unspecified' taints kernel.
```

Для обозначения лицензии вашего модуля вы можете использовать ряд макросов, например: «GPL», «GPL v2», «GPL and additional rights», «Dual BSD/GPL», «Dual MIT/GPL», «Dual MPL/GPL» и «Proprietary».

Определены они в *include/linux/module.h*. Для указания используемой лицензии существует макрос `MODULE_LICENSE`.
Он и ещё пара макросов, описывающих модуль, приведены в примере ниже:

```
/*
 * hello-4.c - Демонстрирует документирование модуля.
 */
```

```c
#include <linux/init.h> /* Необходим для макросов */
#include <linux/kernel.h> /* Необходим для pr_info() */
#include <linux/module.h> /* Необходим для всех модулей */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("LKMPG");
MODULE_DESCRIPTION("A sample driver");

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

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

module_init(init_hello_4);
module_exit(cleanup_hello_4);
```

4.5 Передача в модуль аргументов командной строки

Модулям можно передавать аргументы командной строки, но не через `argc`/`argv`, к которым вы, возможно, привыкли.

Чтобы получить такую возможность, нужно объявить переменные, которые будут принимать значения аргументов командной строки как глобальные и затем использовать макрос `module_param()` (определяемый в *include/linux/moduleparam.h*) для настройки этого механизма. Во время выполнения `insmod` будет заполнять эти переменные получаемыми аргументами, например, `insmod mymodule.ko myvariable=5`.

Для большей ясности объявления переменных и макросов необходимо размещать в начале модулей. Более наглядно всё это продемонстрировано в примере кода.

Макрос `module_param()` получает 3 аргумента: имя переменной, её тип и разрешения для соответствующего файла в `sysfs`. Целочисленные типы могут быть знаковыми, как обычно, или беззнаковыми. Если вы хотите использовать массивы целых чисел или строк, к вашим услугам `module_param_array()` и `module_param_string()`.

```c
int myint = 3;
module_param(myint, int, 0);
```

Массивы тоже поддерживаются, но в современных версиях работает это несколько иначе, нежели раньше. Для отслеживания количества параметров необходимо передать указатель на число переменных в качестве третьего аргумента.

При желании вы можете вообще проигнорировать подсчёт и передать `NULL`. Вот пример обоих вариантов:

```
int myintarray[2];
module_param_array(myintarray, int, NULL, 0); /* если подсчёт не интересует */

short myshortarray[4];
int count;
module_param_array(myshortarray, short, &count, 0); /* подсчёт происходит в
переменной "count" */
```

Хорошим применением для этого варианта будет предустановка значений переменных модуля, таких как порт или адрес ввода-вывода. Если переменные содержат предустановленные значения, выполнять автообнаружение. В противном случае оставлять текущее значение. Позже об этом будет сказано подробнее.

Наконец, есть ещё макрос `MODULE_PARM_DESC()`, используемый для документирования аргументов, которые может принять модуль. Он получает два параметра: имя переменной и строку в свободной форме, эту переменную описывающую.

Пример передачи аргументов командной строки в модуль:

```
/*
 * hello-5.c - демонстрирует передачу аргументов командной строки в модуль.
 */
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";
static int myintarray[2] = { 420, 420 };
static int arr_argc = 0;

/* module_param(foo, int, 0000)
 * Первым аргументом указывается имя параметра.
```

```
 * Вторым указывается его тип.
 * Третьим указываются биты разрешений
 * для представления параметров в sysfs (если не нуль) позднее.
 */
module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");

/* module_param_array(name, type, num, perm);
 * Первым аргументом идёт имя параметра (в данном случае массива).
 * Второй аргумент – это тип элементов массива.
 * Третий – это указатель на переменную, которая будет хранить количество элементов
массива, инициализированных пользователем при загрузке модуля.
 * Четвёртый аргумент – это биты разрешения.
 */
module_param_array(myintarray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintarray, "An array of integers");

static int __init hello_5_init(void)
{
    int i;

    pr_info("Hello, world 5\n=============\n");
    pr_info("myshort is a short integer: %hd\n", myshort);
    pr_info("myint is an integer: %d\n", myint);
    pr_info("mylong is a long integer: %ld\n", mylong);
    pr_info("mystring is a string: %s\n", mystring);

    for (i = 0; i < ARRAY_SIZE(myintarray); i++)
        pr_info("myintarray[%d] = %d\n", i, myintarray[i]);

    pr_info("got %d arguments for myintarray.\n", arr_argc);
    return 0;
}

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

module_init(hello_5_init);

module_exit(hello_5_exit);
```

Рекомендую поэкспериментировать со следующим кодом:

```
$ sudo insmod hello-5.ko mystring="bebop" myintarray=-1
$ sudo dmesg -t | tail -7
myshort is a short integer: 1
myint is an integer: 420
mylong is a long integer: 9999
mystring is a string: bebop
myintarray[0] = -1
myintarray[1] = 420
got 1 arguments for myintarray.

$ sudo rmmod hello-5
$ sudo dmesg -t | tail -1
Goodbye, world 5

$ sudo insmod hello-5.ko mystring="supercalifragilisticexpialidocious"
myintarray=-1,-1
$ sudo dmesg -t | tail -7
myshort is a short integer: 1
myint is an integer: 420
mylong is a long integer: 9999
mystring is a string: supercalifragilisticexpialidocious
myintarray[0] = -1
myintarray[1] = -1
got 2 arguments for myintarray.

$ sudo rmmod hello-5
$ sudo dmesg -t | tail -1
Goodbye, world 5

$ sudo insmod hello-5.ko mylong=hello
insmod: ERROR: could not insert module hello-5.ko: Invalid parameters
```

4.6 Модули, состоящие из нескольких файлов

Иногда есть смысл поделить модуль на несколько файлов.
Вот пример такого модуля:

```
/*
 * start.c - пример модулей, состоящих из нескольких файлов.
 */

#include <linux/kernel.h> /* Выполнение работы ядра. */
#include <linux/module.h> /* В частности, модуля. */
```

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

MODULE_LICENSE("GPL");
```

Второй файл:

```c
/*
 * stop.c — пример модулей, состоящих из нескольких файлов.
 */

#include <linux/kernel.h> /* Выполнение работы ядра. */
#include <linux/module.h> /* В частности, модуля. */

void cleanup_module(void)
{
    pr_info("Short is the life of a kernel module\n");
}

MODULE_LICENSE("GPL");
```

И, наконец, Makefile:

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

PWD := $(CURDIR)

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Это полный Makefile для всех примеров, которые мы успели рассмотреть. Первые пять строчек не представляют ничего особенного, но для последнего примера нам потребуется

две строки. В первой мы придумываем имя объекта для нашего комбинированного модуля, а во второй сообщаем `make`, какие объектные файлы являются его частью.

4.7 Сборка модулей для скомпилированного ядра

Естественно, мы настоятельно рекомендуем вам перекомпилировать ядро, чтобы иметь возможность активировать ряд полезных функций отладки, таких как принудительная выгрузка модулей (`MODULE_FORCE_UNLOAD`): когда эта опция включена, можно с помощью команды `sudo rmmod -f module` принудить ядро выгрузить модуль, даже если оно сочтёт это небезопасным.

В процессе разработки модуля эта опция может сэкономить вам много времени и избавить от лишних перезагрузок. Если вы не хотите перекомпилировать ядро, то рассмотрите вариант выполнения примеров внутри тестового дистрибутива на виртуальной машине. В таком случае при нарушении работоспособности вы сможете легко перезагрузиться или восстановить VM.

Существует ряд случаев, в которых вам может потребоваться загрузить модуль в уже скомпилированное работающее ядро. Как вариант, это может быть типичный дистрибутив Linux или ядро, которое вы сами скомпилировали ранее. Бывает, что загрузить модуль нужно в работающее ядро, перекомпилировать которое нет возможности, или на машину, перезагружать которую нежелательно.

Если вам сложно представить случай, который может вынудить вас использовать модули для уже скомпилированного ядра, то просто пропустите этот раздел и расценивайте оставшуюся часть главы как большое примечание.

Итак, если вы просто установите дерево исходного кода, используете его для компиляции модуля и попытаетесь внедрить этот модуль в ядро, то в большинстве случаев получите ошибку:

```
insmod: ERROR: could not insert module poet.ko: Invalid module format
```

Более понятная информация логируется в системный журнал:

```
kernel: poet: disagrees about version of symbol module_layout
```

Иными словами, ваше ядро отказывается принимать модуль, потому что строки версии (точнее, *vermagic*, см. *include/linux/vermagic.h*) не совпадают. К слову, строки версии хранятся в объекте модуля в виде статической строки, начинающейся с `vermagic:`.

Данные версии вставляются в модуль, когда он линкуется с файлом *kernel/module.o*. Для просмотра сигнатуры версии и прочих строк, хранящихся в конкретном модуле, выполните команду `modinfo module.ko`:

```
$ modinfo hello-4.ko
description:    A sample driver
author:         LKMPG
license:        GPL
srcversion:     B2AA7FBFCC2C39AED665382
depends:
retpoline:      Y
name:           hello_4
vermagic:       5.4.0-70-generic SMP mod_unload modversions
```

Для преодоления этой проблемы можно задействовать опцию `--force-vermagic`, но такое решение не гарантирует безопасность и однозначно будет неприемлемым в создании модулей. Следовательно, модуль нужно скомпилировать в среде, которая была идентична той, где создано наше скомпилированное ядро. Этому и будет посвящён остаток текущей главы.

Во-первых, убедитесь, что дерево исходного кода ядра вам доступно и имеет одинаковую версию с вашим текущим ядром. Далее найдите файл конфигурации, который использовался для компиляции ядра.

Обычно он доступен в текущем каталоге *boot* под именем вроде *config-5.14.x*. Его будет достаточно скопировать в дерево исходного кода вашего ядра:

```
cp /boot/config-`uname -r` .config
```

Далее мы ещё раз сосредоточимся на предыдущем сообщении об ошибке: более пристальное рассмотрение строк версий говорит о том, что даже в случае двух абсолютно одинаковых файлов конфигурации небольшое отличие в версии всё же возможно, и будет достаточно исключить внедрение модуля в ядро.

Это небольшое отличие, а именно пользовательская строка, которая присутствует в версии модуля, но отсутствует в версии ядра, вызвано изменением относительно оригинала в Makefile, который содержат некоторые дистрибутивы.

Далее вам нужно просмотреть собственный Makefile и обеспечить, чтобы представленная информация версии в точности соответствовала той, что указана в текущем ядре.

Например, ваш Makefile может начинаться так:

```
VERSION = 5
PATCHLEVEL = 14
```

```
SUBLEVEL = 0
EXTRAVERSION = -rc2
```

В этом случае необходимо восстановить значение символа `EXTRAVERSION` на `-rc2`. Мы рекомендуем держать резервную копию Makefile, используемого для компиляции ядра, в */lib/modules/5.14.0-rc2/build*. Для этого будет достаточно выполнить:

```
cp /lib/modules/`uname -r`/build/Makefile linux-`uname -r`
```

Здесь `linux-`uname -r`` — это исходный код ядра, которое вы собираетесь собрать.

Теперь выполните `make` для обновления конфигурации вместе с заголовками версии и объектами:

```
$ make
  SYNC    include/config/auto.conf.cmd
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  HOSTCC  scripts/kconfig/confdata.o
  HOSTCC  scripts/kconfig/expr.o
  LEX     scripts/kconfig/lexer.lex.c
  YACC    scripts/kconfig/parser.tab.[ch]
  HOSTCC  scripts/kconfig/preprocess.o
  HOSTCC  scripts/kconfig/symbol.o
  HOSTCC  scripts/kconfig/util.o
  HOSTCC  scripts/kconfig/lexer.lex.o
  HOSTCC  scripts/kconfig/parser.tab.o
  HOSTLD  scripts/kconfig/conf
```

Если же вы не хотите фактически компилировать ядро, то можете прервать процесс сборки (CTRL-C) сразу же после строки `SPLIT`, поскольку в этот момент необходимые вам файлы уже готовы.

Теперь можно вернуться в каталог модуля и скомпилировать его: он будет собран в точном соответствии с настройками текущего ядра и загрузится в него без каких-либо ошибок.

5. Общие сведения

5.1 Начало и завершение модулей

Программа обычно начинается с функции `main()`, выполняет ряд инструкций, после чего завершается. А вот модули ядра работают несколько иначе. Модуль всегда начинается

либо с `init_module`, либо с функции, которую мы указываем вызовом `module_init`. У модулей это функция входа, которая сообщает ядру, какую функциональность модуль несёт, и настраивает ядро на выполнение этой функциональности при необходимости. По завершении функции входа, модуль переходит в состояние бездействия, пока ядру не потребуется от него некая функциональность для работы с кодом.

Заканчиваются все модули вызовом либо `cleanup_module`, либо функции, указываемой вызовом `module_exit`. У модулей это функция выхода. Она отменяет всё, что до этого сделала функция входа, и отменяет регистрацию всей ранее введённой ей функциональности.

Обе описанные функции входа и выхода должны присутствовать в каждом модуле. А поскольку для их определения существует не один способ, я постараюсь использовать общие термины «функция входа» и «функция выхода», но если вдруг по недосмотру назову их `init_module` и `cleanup_module`, то, думаю, вы поймёте, что я имею в виду.

5.2 Функции, доступные модулям

Программисты используют функции, не требующие постоянного переопределения. Хорошим примером этого является `printf()`. Это лишь одна из функций, предоставляемых стандартной библиотекой libc. Фактически их определения не попадают в программу до этапа линковки, который гарантирует доступность кода (например, для `printf()`) и направляет на этот код инструкцию вызова.

Здесь модули ядра тоже отличаются. В примере “Hello world” вы могли заметить, что мы использовали функцию `pr_info`, но не включали стандартную библиотеку ввода-вывода. Причина в том, что модули – это объектные файлы, чьи символы разрешаются при выполнении `insmod` или `modprobe`.

Определение для символов поступает из самого ядра. Единственными внешними функциями, которые можно использовать, являются те, что предоставляет ядро. Если вам интересно узнать, какие символы ваше ядро экспортировало, загляните в `/proc/kallsyms`.

При всём при этом нужно помнить о различии между библиотечными функциями и системными вызовами. Библиотечные функции работают на более высоком уровне, выполняясь полностью в пользовательском пространстве и предоставляя программисту более удобный интерфейс для доступа к функциям, которые и совершают реальную работу – системным вызовам. Эти вызовы, в свою очередь, выполняются в режиме ядра от имени пользователя и предоставляются самим ядром.

Библиотечная функция `printf()` может выглядеть как обобщённая функция вывода, но на деле она лишь форматирует данные в строки и записывает эти строчные данные с

помощью низкоуровневого системного вызова `write`, который затем отправляет их в стандартный вывод.

Хотите увидеть, какие системные вызовы совершает `printf()`? Легко! Скомпилируйте следующую программу с помощью `gcc -Wall -o hello hello.c`:

```
#include <stdio.h>

int main(void)
{
    printf("hello");
    return 0;
}
```

Запустите исполняемый файл командой `strace ./hello`. Впечатлены? Каждая строка, которую вы видите, соответствует системному вызову. *strace* – это удобная утилита, сообщающая подробности о том, какие системные вызовы совершает программа, включая то, какие аргументы эти вызовы содержат и какие результаты возвращают.

Это невероятно ценный инструмент, позволяющий выяснять, к каким файлам обращается программа. Ближе к концу вы увидите строку вроде `write(1, "hello", 5hello)`. Вот оно – лицо, скрытое за маской `printf()`.

Вы можете быть незнакомы с `write()`, поскольку большинство людей для файлового ввода-вывода используют библиотечные функции (например, `fopen`, `fputs`, `fclose`).

Если так и есть, то рекомендую заглянуть в мануал, `man 2 write`. Второй раздел в нём посвящён системным вызовам (вроде `kill()` и `read()`). Третий раздел описывает библиотечные вызовы (вроде `cosh()` и `random()`), с которыми вы наверняка уже более знакомы.

Вы даже можете писать модули на замену системных вызовов ядра, чем мы вскоре и займёмся. Взломщики зачастую используют подобные приёмы для бэкдоров или троянов, но вы можете создавать собственные модули из более доброжелательных побуждений, например, чтобы ядро писало `"Tee hee, that tickles!"` (Хи-хи, щекотно!) каждый раз, когда кто-то пытается удалить в системе файл.

5.3 Пользовательское пространство и пространство ядра

Ядро (по своей сути), регулирует доступ к ресурсам, будь то видеокарта, жёсткий диск или память. При этом программы зачастую соперничают за право использовать один и тот же ресурс. Как только я сохранил документ, `updatedb` начала обновлять локальную базу данных. Мой сеанс VIM и `updatedb` используют жёсткий диск конкурентно. Ядру

необходимо сохранять во всём этом порядок, а не давать пользователям доступ к ресурсам в любой момент, когда им вздумается.

В связи с этим ЦПУ может работать в нескольких режимах. Каждый режим даёт определённый уровень свободы действий в системе. В архитектуре 80386 есть 4 таких режима, называемых кольцами защиты. В Unix используется только два таких кольца: внутреннее (кольцо 0, также известное как «режим супервизора», в котором допустимы все действия) и внешнее, называемое «режим пользователя».

Вспомним разговор о библиотечных и системных вызовах. Обычно мы используем библиотечную функцию в режиме пользователя. Эта библиотечная функция, в свою очередь, совершает один или более системных вызовов, которые выполняют от её имени действия, но делают это уже в режиме супервизора, поскольку являются частью самого ядра. Как только системный вызов завершает задачу, он делает возврат, и выполнение передаётся обратно в режим пользователя.

5.4 Пространство имён

Когда вы пишете небольшую программу Си, то используете удобные переменные, которые будут иметь смысл для пользователя. Если же, напротив, вы пишете подпрограммы, которые станут частью более крупной задачи, то любые используемые глобальные переменные являются частью коллекции глобальных переменных других людей, в связи с чем иногда могут возникать коллизии между их имён. Когда в программе используется множество глобальных переменных, которые недостаточно значительны, чтобы проводить между ними различие, у нас получается загрязнение пространства имён. В крупных проектах необходимо стремиться запоминать зарезервированные имена и вырабатывать схему для именования уникальных переменных и символов.

При написании кода ядра даже малейший модуль будет залинкован со всем ядром, так что это определённо важно. Проще всего в таком случае объявлять все переменные статическими и использовать для символов грамотные префиксы. По соглашению все префиксы в ядре пишутся в нижнем регистре. Если же вы не хотите объявлять что-либо статично, то другой вариант – объявить таблицу символов и зарегистрировать её с помощью ядра. Чуть позже мы об этом поговорим.

В файле */proc/kallsyms* хранятся все символы, о которых ядро знает, и которые, благодаря этому, являются доступными для модулей, поскольку находятся в едином пространстве кода ядра.

5.5 Кодовое пространство

Управление памятью является очень сложной темой, и большая часть книги Understanding The Linux Kernel издательства O'Reilly посвящено именно ей. Мы не ставим задачу стать

экспертами в этой области, но для того, чтобы даже задуматься над написанием реальных модулей нам необходимо знать пару фактов.

Если вы ещё не думали о том, что в самом деле значит `segfault` (ошибка сегментации), то можете удивиться, услышав, что в действительности указатели не указывают на области памяти, по крайней мере, на реальные. При создании процесса ядро выделяет часть реальной физической памяти и передаёт её этому процессу для размещения в ней выполняемого кода, переменных, стека, кучи и прочих вещей, о которых должен знать специалист по информатике.

Эта память начинается с `0x00000000` и простирается до необходимых значений. Поскольку область памяти для любых двух процессов не пересекается, все процессы, которые могут обращаться к адресу памяти, скажем `0xbffff978`, будут обращаться к разным областям реальной физической памяти. Они будут обращаться к индексу `0xbffff978`, указывающему на некое смещение в области памяти, выделенной конкретно для этого процесса. В большинстве случаев процесс вроде нашей программы “Hello World” не может получить доступ к пространству другого процесса, хотя для этого есть определённые способы, о которых мы поговорим позже.

У ядра также есть собственная область памяти. Поскольку модуль является кодом, который может внедряться в ядро и извлекаться из него (в противоположность полуавтономному объекту), он использует кодовое пространство ядра, не имея собственного. Следовательно, если ваш модуль допускает ошибку сегментации, то и с ядром происходит то же самое. И если вы начнёте производить запись поверх данных в результате ошибки смещения на единицу, то происходить это будет поверх данных (или кода) ядра. На деле это даже хуже, чем звучит, так что будьте очень осторожны.

Кстати, хочу отметить, что описанное выше касается любой операционной системы, использующей монолитное ядро. Это не совсем то же, что «встраивание всех модулей в ядро», хотя суть аналогична. Существует такое понятие, как микроядра, которые имеют модули, получающие собственное кодовое пространство. Примерами таких микроядер являются GNU Hurd и Zircon.

5.6 Драйверы устройств

Одним из классов модулей являются драйверы устройств, которые предоставляют функциональность для оборудования вроде последовательных портов. В Unix каждый элемент оборудования представлен файлом устройства, расположенным в */dev* и предоставляющим средства для связи с этим оборудованием. Драйвер устройства обеспечивает связь со стороны пользовательской программы. Например, драйвер звуковой карты *es1370.ko* может подключать файл устройства */dev/sound* к звуковой карте Ensoniq IS1370. В результате программа в пользовательском пространстве, например, *mp3blaster*, может использовать */dev/sound*, даже не зная, какая именно звуковая карта установлена.

Рассмотрим некоторые файлы устройств. Ниже приведены их примеры, которые представляют первые три раздела на ведущем HDD:

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

Обратите внимание на числа, отделённые запятой. Первое называется старшим (major) номером устройства, а второе младшим (minor). Старший номер сообщает, какой драйвер используется для доступа к оборудованию. Каждому драйверу присваивается уникальный старший номер.

Все файлы устройств с одинаковым старшим номером управляются одним драйвером. Выше мы видим в качестве таких номеров три `3`, поскольку всеми этими устройствами управляет один драйвер.

При этом по младшим номерам драйвер отличает один управляемый им компонент оборудования от другого. В примере выше, несмотря на то что все три устройства управляются одним драйвером, их младшие номера отличаются, поскольку этот драйвер видит их как разные компоненты оборудования.

Устройства делятся на два типа: блочные и символьные. Отличие между ними в том, что блочные имеют буфер для запросов, благодаря чему могут выбирать наилучший порядок, в котором на эти запросы отвечать.

Это важно в случае устройств хранения, когда получается быстрее считывать/записывать близкорасположенные сектора, нежели те, что удалены друг от друга.

Ещё одним отличием является то, что блочные устройства могут получать вход и возвращать выход только блоками (чей размер может отличаться в зависимости от устройства), а символьным дозволено использовать любое необходимое им количество байтов. Большинство устройств являются именно символьными, поскольку им не требуется подобная буферизация, и они не работают с фиксированным размером блоков.

Понять, для какого устройства используется файл устройства – блочного или символьного – можно по первому символу вывода команды `ls -l`. Если это `b`, значит — устройство блочное, а если `c`, значит — символьное. Приведённые выше устройства все являются блочными, а вот несколько символьных (последовательные порты):

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

Если хотите увидеть, какие устройствам были присвоены старшие номера, можете заглянуть в Documentation/admin-guide/devices.txt.

При установке системы все эти файлы устройств создавались командой `mknod`. Для создания нового символьного устройства под именем `coffee` со старшим/младшим номерам `12`/`2` просто выполните `mknod /dev/coffee c 12 2`.

Вам не обязательно помещать файлы устройств в */dev*, но того требует соглашение. Линукс размещает эти файлы в */dev*, и вам стоит делать так же. Однако при создании файла устройства для тестирования вполне допустимо разместить его в рабочем каталоге, где вы компилируете модуль ядра. Только не забудьте перенести его в нужное место, когда закончите написание драйвера.

Напоследок хочу дополнительно прояснить момент, который может быть неочевиден из пояснения выше. Когда происходит обращение к файлу устройства, ядро по его старшему номеру определяет, какой драйвер нужно использовать для обработки этого обращения. То есть ядру не обязательно использовать, или даже знать, младший номер. Этот номер интересует лишь драйвер устройства, который использует его для различения отдельных компонент оборудования.

Кстати, когда я говорю «оборудование», то подразумеваю несколько более абстрактное понятие, нежели какая-нибудь PCI-карта, которую вы держите в руках. Взгляните на эти два файла устройств:

```
$ ls -l /dev/sda /dev/sdb
brw-rw---- 1 root disk 8,  0 Jan  3 09:02 /dev/sda
brw-rw---- 1 root disk 8, 16 Jan  3 09:02 /dev/sdb
```

Теперь, глядя на них, вы можете сходу понять, что они являются блочными устройствами и обрабатываются одним драйвером. Иногда два файла устройств с одним старшим, но разными младшими номерами на деле могут представлять один и тот же компонент оборудования.

Так что имейте в виду, что слово «оборудование» в этом пособии может иметь весьма абстрактное значение.

6. Драйверы символьных устройств

6.1 Структура file_operations

Структура `file_operations` находится в *include/linux/fs.h* и содержит указатели на определённые драйвером функции, которые выполняют различные действия с

устройством. Каждое поле этой структуры соответствует адресу некой функции, определённой драйвером для обработки операции запроса.

Например, каждый символьный драйвер должен определять функцию, считывающую данные с устройства. Структура `file_operations` содержит адрес функции модуля, которая выполняет эту операцию.

Вот как это определение выглядит в ядре 5.4:

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iopoll)(struct kiocb *kiocb, bool spin);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    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);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
        loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
    ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
        loff_t, size_t, unsigned int);
    loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
            struct file *file_out, loff_t pos_out,
```

```
            loff_t len, unsigned int remap_flags);
    int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

При этом некоторые операции драйвером не реализуются.

Например, драйверу, обрабатывающему видеокарту, не требуется выполнять чтение из структуры каталогов. Соответствующие записи в структуре `file_operations` должны быть установлены на `NULL`.

Для компилятора gcc есть расширение, которое упрощает присваивание значений в этой структуре. В современных драйверах оно встречается довольно часто, так что не удивляйтесь, если его увидите.

Так выглядит новый способ присваивания значений в структуре:

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

Однако присваивать элементам структуры значения можно и в соответствии со стандартом C99, с помощью назначенных инициализаторов. Причём такой способ определённо предпочтительнее, чем применение расширения GNU. Этот синтаксис желательно использовать в случае, когда стоит задача портировать драйвер, так как он обеспечит лучшую совместимость:

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

Смысл ясен и вам нужно иметь в виду, что любой член структуры, которому вы не присвоите значение явно, gcc инициализирует с `NULL`.
Экземпляр `struct file_operations`, содержащий указатели на функции, используемые для реализации системных вызовов `read`, `write`, `open` и так далее, обычно называется `fops`.

Начиная с Linux v3.14, операции чтения, записи и поиска гарантированно потокобезопасны за счёт использования специальной блокировки `f_pos`, которая

превращает обновление позиции файла во взаимное исключение. Благодаря этому, можно безопасно реализовывать подобные операции без излишних блокировок.

Начиная с Linux v5.6, была введена структура `proc_ops`, заменившая использование структуры `file_operations` при регистрации обработчиков процессов.

6.2 Структура file

Каждое устройство представлено в ядре структурой `file`, которая определяется в *include/linux/fs.h*. Имейте ввиду, что `file` – это структура уровня ядра, которая никогда не появляется в программе пользовательского пространства. Это не то же самое, что `FILE`, который определяется glibc и никогда не встречается в функции пространства ядра. Кроме того, само имя структуры может сбивать с толку, так как представляет абстрактный открытый `file`, а не файл на диске, который представляется структурой `inode`.

Экземпляр структуры `file` обычно называется `filp`. Вы также увидите, что порой её называют структурой *file object* – пусть это не вводит вас в заблуждение.

Загляните в определение `file`. Большинство записей здесь, такие как `struct dentry`, не используются драйверами устройств, и их можно игнорировать. Причина в том, что драйверы не заполняют `file` непосредственно, а лишь используют содержащиеся в ней структуры, которые создаются где-то ещё.

6.3 Регистрация устройства

Как уже говорилось, обращение к символьным устройствам происходит через файлы устройств, обычно расположенные в */dev*.

Тем не менее — при написании драйвера вполне допустимо поместить файл устройства в текущий рабочий каталог с тем условием, что по завершении он будет перенесён в */dev*.

Старший номер сообщает, какой драйвер какой файл устройства обрабатывает.

Младший же номер используется только самим драйвером для определения конкретного устройства, с которым он работает.

Добавление драйвера в систему означает регистрацию его с помощью ядра. Это аналогично присваиванию ему старшего номера во время инициализации модуля и выполняется с помощью функции `register_chrdev`, определённой в *include/linux/fs.h*.

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

Здесь `unsigned major int` является старшим номером, который мы хотим запросить, `const char *name` – это имя устройства в том виде, в котором оно отобразится в */proc/devices*, а `struct file_operations *fops` – это указатель на таблицу `file_operations` для вашего драйвера. Отрицательное возвращаемое значение означает, что регистрация провалилась.

Заметьте, что мы не передавали в `register_chrdev` младший номер. Ещё раз напомню, что ядру он не важен, его использует только драйвер.

Следующий вопрос в том, как получить старший номер, не взяв случайно тот, что уже используется? Проще всего заглянуть в Documentation/admin-guide/devices.txt и выбрать свободный. Но это будет не самый удачный способ, поскольку вы никогда не сможете быть уверены, что выбранный вами номер не окажется присвоен где-то позднее.
Решением будет попросить ядро присвоить динамический старший номер.

Если передать в `register_chrdev` старший номер `0`, возвратным значением будет его динамически выделяемое значение. Недостаток такого решения в том, что не получится создать файл устройства наперёд, поскольку вы не будете знать, какой ему будет присвоен старший номер.

Выйти из ситуации можно несколькими путями:

- номер может выводить сам драйвер, и мы будем создавать файл устройства вручную.
- регистрируемое устройство будет иметь запись в */proc/devices*, и мы сможем либо сами создать файл устройства, либо написать для этого специальный скрипт оболочки.
- можно сделать и так, чтобы наш драйвер сам создавал файл устройства, используя функцию `device_create` после успешной регистрации, и `device_destroy` во время вызова `cleanup_module`.

Однако `register_chrdev()` будет занимать ряд младших номеров, связанных с заданным старшим. Поэтому с целью уменьшения лишних затрат при регистрации символьного устройства рекомендуется использовать интерфейс `cdev`. Этот более свежий интерфейс завершает регистрацию в два раздельных этапа. Во-первых, нам нужно зарегистрировать серию номеров устройств, что можно сделать с помощью `register_chrdev_region` или `alloc_chrdev_region`:

```
int register_chrdev_region(dev_t from, unsigned count, const char *name);
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char
*name);
```

Выбор одной из этих функций будет зависеть от того, известны ли вам старшие номера вашего устройства. Используйте `register_chrdev_region`, если знаете их, и `alloc_chrdev_region`, если хотите сделать их выделение динамическим.

Вторым этапом необходимо инициализировать для нашего символьного устройства структуру данных `struct cdev` и связать её с номерами устройства.

Эту инициализацию можно осуществить следующей последовательностью команд:

```
struct cdev *my_dev = cdev_alloc();
my_cdev->ops = &my_fops;
```

Тем не менее в стандартном сценарии `struct cdev` будет встроена в вашу собственную связанную с устройством структуру. В этом случае нам для инициализации необходима `cdev_init`.

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

По завершении инициализации можно добавить символьное устройства в систему с помощью `cdev_add`.

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

Пример использования этого интерфейса можно найти в *ioctl.c*, описанном в разделе 9.

6.4 Отмена регистрации устройства

Мы не можем позволить рут-пользователю извлекать (`rmmod`) модуль ядра в любой момент, когда ему это вздумается. Если извлечь модуль в то время, когда файл устройства будет открыт процессом, то использование этого файла приведёт к вызову из области памяти, где ранее находилась нужная функция (чтения/записи).

В лучшем случае, если никакой другой код в эту область ещё записан не был, мы просто получим неприятную ошибку. В худшем же в эту память уже мог быть загружен другой модуль, что приведёт к перескакиванию в середину уже иной функции внутри ядра, вызвав непредсказуемый и явно не радужный результат.

Как правило, когда вы хотите запретить какое-то действие, то возвращаете код ошибки (отрицательное число) из функции, которая это действие должна была выполнить. В случае с `cleanup_module` так сделать не получится, поскольку это пустая функция.

Тем не менее существует счётчик, который отслеживает, сколько процессов используют ваш модуль. Значение этого счётчика можно увидеть в 3 поле вывода команды `cat /proc/modules` или `sudo lsmod`. Если это не нуль, значит, `rmmod` провалится.

Имейте в виду, что проверять счётчик в `cleanup_module` не нужно, так как эта проверка будет выполнена за вас системным вызовом `sys_delete_module`, определённым в *include/linux/syscalls.h*.

Этот счётчик не требуется использовать непосредственно, но *include/linux/module.h* содержит функции, которые позволяют вам при необходимости увеличивать, уменьшать и отображать его:

- `try_module_get(THIS_MODULE)`: инкрементирует число активных обращений к текущему модулю;
- `module_put(THIS_MODULE)`: декрементирует число активных обращений к текущему модулю;
- `module_refcount(THIS_MODULE)`: возвращает число активных обращений к текущему модулю.

Важно поддерживать точное значение счётчика. Если вы вдруг утратите верный счёт, то уже не сможете выгрузить модуль, и останется единственный выход – перезагрузка. В процессе разработки модуля такая ситуация с вами рано или поздно неизбежно случится.

6.5 chardev.c

Код ниже создаёт символьный драйвер `chardev`. Можете сделать дамп его файла устройства.

```
cat /proc/devices
```

(Либо откройте этот файл программой), и драйвер добавит в него значение, указывающее количество раз, которое он был считан. Запись в этот файл (вроде `echo "hi" > /dev/hello`) мы не поддерживаем, перехватывая такие попытки и сообщая пользователю, что данная операция недопустима.

Не беспокойтесь, если не видите, что мы делаем с данными, которые считываем в буфер – они просто считываются, и выводится сообщение, подтверждающее их получение.

В многопоточной среде без защиты параллельное обращение к одному участку памяти может привести к состоянию гонки и снизить производительность. В модуле ядра эта

проблема может происходить в результате обращения нескольких экземпляров программ к общим ресурсам.

Решается она обеспечением индивидуального доступа. Мы используем атомарную инструкцию сравнения с обменом (CAS) для сохранения состояний `CDEV_NOT_USED` и `CDEV_EXCLUSIVE_OPEN`, чтобы определять, открыт ли в данный момент файл какой-либо программой. CAS сравнивает содержимое области памяти с ожидаемым значением и только в случае их совпадения изменяет содержимое этой памяти на нужное значение. Подробнее о конкурентности читайте в разделе 12.

Код chardev.c:

```
/*
 * chardev.c: создаёт символьное устройство, которое сообщает, сколько
 * раз происходило считывание из файла.
 */

#include <linux/cdev.h>
#include <linux/delay.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/irq.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/poll.h>

/*  Prototypes - обычно помещается в файл .h */
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char __user *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char __user *, size_t,
                            loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev" /* Имя устройства, как оно показано в /proc/devices
*/
#define BUF_LEN 80 /* Максимальная длина сообщения устройства. */

/* Глобальные переменные объявляются как static, поэтому являются глобальными в
пределах файла. */

static int major; /* Старший номер, присвоенный драйверу устройства */

enum {
    CDEV_NOT_USED = 0,
    CDEV_EXCLUSIVE_OPEN = 1,
```

```
};

/* Устройство открыто? Используется для предотвращения множественных обращений к
устройству. */
static atomic_t already_open = ATOMIC_INIT(CDEV_NOT_USED);

static char msg[BUF_LEN]; /* msg, которое устройство будет выдавать при запросе. */

static struct class *cls;

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

static int __init chardev_init(void)
{
    major = register_chrdev(0, DEVICE_NAME, &chardev_fops);

    if (major < 0) {
        pr_alert("Registering char device failed with %d\n", major);
        return major;
    }

    pr_info("I was assigned major number %d.\n", major);

    cls = class_create(THIS_MODULE, DEVICE_NAME);
    device_create(cls, NULL, MKDEV(major, 0), NULL, DEVICE_NAME);

    pr_info("Device created on /dev/%s\n", DEVICE_NAME);

    return SUCCESS;
}

static void __exit chardev_exit(void)
{
    device_destroy(cls, MKDEV(major, 0));
    class_destroy(cls);

    /* Отмена регистрации устройства. */
    unregister_chrdev(major, DEVICE_NAME);
}

/* Методы. */

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

```
 * "sudo cat /dev/chardev"
 */
static int device_open(struct inode *inode, struct file *file)
{
    static int counter = 0;

    if (atomic_cmpxchg(&already_open, CDEV_NOT_USED, CDEV_EXCLUSIVE_OPEN))
        return -EBUSY;

    sprintf(msg, "I already told you %d times Hello world!\n", counter++);
    try_module_get(THIS_MODULE);

    return SUCCESS;
}

/* Вызывается, когда процесс закрывает файл устройства. */
static int device_release(struct inode *inode, struct file *file)
{
    /* Теперь можно принимать следующий вызов. */
    atomic_set(&already_open, CDEV_NOT_USED);

    /* Декрементируйте число использований, иначе, открыв файл, вы уже
     * не сможете извлечь модуль.
     */
    module_put(THIS_MODULE);

    return SUCCESS;
}

/* Вызывается, когда процесс, который уже открыл файл устройства,
 * пытается из него считать.
 */
static ssize_t device_read(struct file *filp, /* см. include/linux/fs.h   */
                           char __user *buffer, /* буфер для данных. */
                           size_t length, /* длина буфера.   */
                           loff_t *offset)
{
    /* Количество байт, обычно записываемых в буфер. */
    int bytes_read = 0;
    const char *msg_ptr = msg;

    if (!*(msg_ptr + *offset)) { /* мы находимся в конце сообщения. */
        *offset = 0; /* сброс смещения. */
        return 0; /* обозначение конца файла. */
    }

    msg_ptr += *offset;
```

```
    /* Помещение данных в буфер. */
    while (length && *msg_ptr) {
        /* Буфер находится в пользовательском сегменте данных, а не в
         * сегменте ядра, поэтому присваивание "*" не сработает. Тут 133         *
нужно использовать put_user, которая копирует данные из
         * сегмента ядра в пользовательский сегмент.
         */
        put_user(*(msg_ptr++), buffer++);
        length--;
        bytes_read++;
    }

    *offset += bytes_read;

    /* Большинство функций чтения возвращают количество байт, помещённых в буфер. */
    return bytes_read;
}

/* Вызывается, когда процесс производит запись в файл устройства: echo "hi" >
/dev/hello */
static ssize_t device_write(struct file *filp, const char __user *buff,
                            size_t len, loff_t *off)
{
    pr_alert("Sorry, this operation is not supported.\n");
    return -EINVAL;
}

module_init(chardev_init);
module_exit(chardev_exit);

MODULE_LICENSE("GPL");
```

6.6 Создание модулей для нескольких версий ядра

Системные вызовы, являющиеся основным интерфейсом, который ядро раскрывает процессам, обычно среди разных версий сохраняются. Иногда могут добавляться новые системные вызовы, но старые, как правило, продолжают работать по-прежнему. Это необходимо для обратной совместимости – новая версия ядра не должна нарушать работу стандартных процессов. Файлы устройств в большинстве случаев также остаются неизменными. С другой стороны, внутренние интерфейсы ядра между версиями вполне могут меняться.

Различные версии ядра определённо имеют между собой отличия, и если вам нужна поддержка нескольких версий, то придётся писать дополнительные директивы компиляции. Делается это путём сопоставления макроса `LINUX_VERSION_CODE` с

макросом `KERNEL_VERSION`. В версии `a.b.c` ядра значение этого макроса будет $2^{16}a + 2^{8}b + c$.

7. Файловая система /proc

В Linux существует дополнительный механизм, позволяющий ядру и модулям отправлять информацию процессам – файловая система `/proc`. Изначально созданная для реализации удобного доступа к информации о процессах (отсюда и название), теперь она используется каждым элементом ядра, обладающим полезной информацией. Например, `/proc/modules` предоставляет список модулей, а `/proc/meminfo` собирает статистику потребления памяти.

Способ использования `procfs` очень схож с использованием драйверов устройств – сперва создается структура со всей информацией, необходимой для файла `/proc`, включая указатели на любые функции-обработчики (в нашем случае такая всего одна, вызываемая при попытке считывания из файла `/proc`). Далее `init_module` регистрирует эту структуру с помощью ядра, а `cleanup_module` ее регистрацию снимает.

Обычные файловые системы располагаются на диске, а не просто в памяти (где находится `/proc`), и в этом случае номером индексного дескриптора (inode) является указатель на область диска, где располагается `inode` файла. Этот `inode` содержит информацию о файле, например разрешения, а также указатель на область или области диска, где находятся данные этого файла.

Поскольку при открытии и закрытии файла вызов мы не получаем, в этом модуле нет места, куда можно было бы внести `try_module_get` и `module_put`, и если при открытом файле вдруг удалить модуль, то это чревато последствиями.

Вот простой пример, демонстрирующий использование файла `/proc`. Это Hello World для файловой системы `/proc`. Здесь у нас три части: создание файла `/proc/helloworld` в функции `init_module`, возвращение значения (и буфера) `/proc/helloworld` в функции обратного вызова `procfile_read` при считывании этого файла и его удаление в функции `cleanup_module`.

Указанный файл создается при загрузке модуля функцией `proc_create`. Возвращаемым значением здесь окажется `struct proc_dir_entry`, которое будет использовано для конфигурирования `/proc/helloworld` (например, указания владельца этого файла). Нулевое возвращаемое значение означает провал создания.

При каждом считывании `/proc/helloworld` вызывается функция `procfile_read`. У этой функции есть два важных параметра: буфер (второй параметр) и смещение (четвертый). Содержимое буфера будет возвращаться приложению, которое его считывает (например, команде `cat`).

Смещение – это текущая позиция файла. Если возвращаемое значение функции не нулевое, тогда эта функция вызывается повторно. Так что будьте с ней внимательны – если она никогда не вернет нуль, то будет вызываться бесконечно.

Код procfs1.c:

```
$ cat /proc/helloworld
HelloWorld!
/*
 * procfs1.c
 */

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

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

#define procfs_name "helloworld"

static struct proc_dir_entry *our_proc_file;

static ssize_t procfile_read(struct file *filePointer, char __user *buffer,
                             size_t buffer_length, loff_t *offset)
{
    char s[13] = "HelloWorld!\n";
    int len = sizeof(s);
    ssize_t ret = len;

    if (*offset >= len || copy_to_user(buffer, s, len)) {
        pr_info("copy_to_user failed\n");
        ret = 0;
    } else {
        pr_info("procfile read %s\n", filePointer->f_path.dentry->d_name.name);
        *offset += len;
    }

    return ret;
}

#ifdef HAVE_PROC_OPS
static const struct proc_ops proc_file_fops = {
    .proc_read = procfile_read,
```

```
};
#else
static const struct file_operations proc_file_fops = {
    .read = procfile_read,
};
#endif

static int __init procfs1_init(void)
{
    our_proc_file = proc_create(procfs_name, 0644, NULL, &proc_file_fops);
    if (NULL == our_proc_file) {
        proc_remove(our_proc_file);
        pr_alert("Error:Could not initialize /proc/%s\n", procfs_name);
        return -ENOMEM;
    }

    pr_info("/proc/%s created\n", procfs_name);
    return 0;
}

static void __exit procfs1_exit(void)
{
    proc_remove(our_proc_file);
    pr_info("/proc/%s removed\n", procfs_name);
}

module_init(procfs1_init);
module_exit(procfs1_exit);

MODULE_LICENSE("GPL");
```

7.1 Структура proc_ops

В Linux v5.6+ структура `proc_ops` определена в `include/linux/proc_fs.h`. В более старых версиях она использовала `file_operations` для реализации в `/proc` пользовательских хуков. Однако в ней содержатся некоторые члены, которые в VFS не нужны, и всякий раз, когда VFS расширяет набор `file_operations`, код `/proc` раздувается. С другой стороны, этой структурой экономилось не только пространство, но и некоторые операции, что повышало ее быстродействие.

Например, файл, который никогда не исчезает в `/proc`, может устанавливать `proc_flag` как `PROC_ENTRY_PERMANENT`, экономя в каждой последовательности открытия/чтения/закрытия 2 атомарных операции: 1 выделение памяти и 1 освобождение.

7.2 Считывание и запись файла /proc

Выше был описан очень простой пример использования `/proc`, в котором мы просто считывали файл `/proc/helloworld`. При этом в `/proc` также можно производить запись. Принцип тот же, что и в случае со считыванием – при записи в файл `/proc` вызывается соответствующая функция.

Но здесь есть небольшое отличие – данные поступают от пользователя – значит их нужно импортировать из пользовательского пространства в пространство ядра (с помощью `copy_from_user` или `get_user`).

Причина использования `copy_from_user` либо `get_user` в том, что память Linux сегментирована (на некоторых процессорах с архитектурой Intel это может быть не так).

То есть указатель сам по себе ссылается не на уникальную область в памяти, а на область в ее сегменте, и для использования этой памяти необходимо знать, что это за сегмент. Существует один сегмент памяти для ядра и по одному для каждого из процессов.

Процессам доступен только их собственный сегмент памяти, поэтому при написании стандартных программ для выполнения в качестве процессов беспокоится о сегментах не приходится.

Когда вы создаете модуль ядра, то обычно вам нужно иметь доступ к сегменту памяти ядра, и это обрабатывается системой автоматически.

Однако, когда содержимое буфера памяти необходимо передать между выполняющимся процессом и ядром, функция ядра получает указатель на буфер памяти, находящийся в сегменте процесса. К этой памяти позволяют обращаться макросы `put_user` и `get_user`, но обрабатывают эти функции только один символ. Для обработки нескольких можно задействовать `copy_to_user` и `copy_from_user`.

Поскольку буфер (в функции чтения или записи) находится в пространстве ядра, для функции записи данные необходимо импортировать, потому что они поступают из пространства пользователя. Функции чтения это не касается, так как в этом случае данные уже находятся в пространстве ядра.

Код procfs2.c:

```
/*
 * procfs2.c - создание "файла" в /proc
 */

#include <linux/kernel.h> /* Для работы с ядром. */
#include <linux/module.h> /* Для модулей. */
```

```c
#include <linux/proc_fs.h> /* Для использования procfs.*/
#include <linux/uaccess.h> /* Для copy_from_user. */
#include <linux/version.h>

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

#define PROCFS_MAX_SIZE 1024
#define PROCFS_NAME "buffer1k"

/* Эта структура содержит информацию о файле /proc. */
static struct proc_dir_entry *our_proc_file;

/* Этот буфер используется под хранение символа для данного модуля. */
static char procfs_buffer[PROCFS_MAX_SIZE];

/* Размер буфера. */
static unsigned long procfs_buffer_size = 0;

/* Эта функция вызывается при считывании файла /proc. */
static ssize_t procfile_read(struct file *filePointer, char __user *buffer,
                             size_t buffer_length, loff_t *offset)
{
    char s[13] = "HelloWorld!\n";
    int len = sizeof(s);
    ssize_t ret = len;

    if (*offset >= len || copy_to_user(buffer, s, len)) {
        pr_info("copy_to_user failed\n");
        ret = 0;
    } else {
        pr_info("procfile read %s\n", filePointer->f_path.dentry->d_name.name);
        *offset += len;
    }

    return ret;
}

/* Эта функция вызывается при записи файла /proc. */
static ssize_t procfile_write(struct file *file, const char __user *buff,
                              size_t len, loff_t *off)
{
    procfs_buffer_size = len;
    if (procfs_buffer_size > PROCFS_MAX_SIZE)
        procfs_buffer_size = PROCFS_MAX_SIZE;

    if (copy_from_user(procfs_buffer, buff, procfs_buffer_size))
```

```
        return -EFAULT;

    procfs_buffer[procfs_buffer_size & (PROCFS_MAX_SIZE - 1)] = '\0';
    pr_info("procfile write %s\n", procfs_buffer);

    return procfs_buffer_size;
}

#ifdef HAVE_PROC_OPS
static const struct proc_ops proc_file_fops = {
    .proc_read = procfile_read,
    .proc_write = procfile_write,
};
#else
static const struct file_operations proc_file_fops = {
    .read = procfile_read,
    .write = procfile_write,
};
#endif

static int __init procfs2_init(void)
{
    our_proc_file = proc_create(PROCFS_NAME, 0644, NULL, &proc_file_fops);
    if (NULL == our_proc_file) {
        proc_remove(our_proc_file);
        pr_alert("Error:Could not initialize /proc/%s\n", PROCFS_NAME);
        return -ENOMEM;
    }

    pr_info("/proc/%s created\n", PROCFS_NAME);
    return 0;
}

static void __exit procfs2_exit(void)
{
    proc_remove(our_proc_file);
    pr_info("/proc/%s removed\n", PROCFS_NAME);
}

module_init(procfs2_init);
module_exit(procfs2_exit);

MODULE_LICENSE("GPL");
```

7.3 Управление файлом /proc с помощью стандартной файловой системы

Мы уже видели, как считывать и записывать файл в `procfs` с помощью интерфейса `/proc`. Но управлять такими файлами также можно и с помощью `inode`. Основная суть здесь в использовании продвинутых функций, таких как разрешения.

В Linux есть стандартный механизм для регистрации файловой системы. Поскольку у каждой такой системы должны быть собственные функции для обработки операций с `inode` и файлами, существует особая структура для хранения указателей на эти функции, `struct inode_operations`, которая также включает указатель на `struct proc_ops`.

Отличает операции с `inode` от операций с файлами то, что последние работают непосредственно с самими файлами, а первые со способами обращения к файлу, например создавая на него ссылки.

В `/proc` при каждой регистрации нового файла мы допустили указание, какая `struct inode_operations` будет использоваться для доступа к нему. Этот механизм мы и используем — `struct inode_operations`, которая включает указатель на `struct proc_ops`, которая, в свою очередь, включает указатели на наши функции `procf_read` и `procfs_write`.

Еще один интересный момент – это функция `module_permission`. Она вызывается всякий раз, когда процесс пытается сделать что-то с файлом `/proc`, и может решать, допускать его к этому файлу или нет.

Сейчас она основана лишь на операции и uid текущего пользователя (в текущей ситуации это доступно из указателя на структуру, которая включает информацию о выполняющемся в данный момент процессе), но также может основываться и на чем-то другом, например на том, какие еще процессы работают с тем же файлом, на времени дня или последнем полученном вводе.

Здесь важно пояснить, что стандартные роли функций чтения и записи в ядре реверсируются. Первые используются для вывода, а вторые для ввода.

Объясняется это тем, что чтение и запись происходят со стороны пользователя – если процесс что-то из ядра считывает, то для ядра это является выводом, а если процесс производит запись в ядро, тогда для ядра это выглядит как ввод.

Код procfs3.c:

```
/*
 * procfs3.c
 */
```

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include <linux/version.h>

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

#define PROCFS_MAX_SIZE 2048
#define PROCFS_ENTRY_FILENAME "buffer2k"

static struct proc_dir_entry *our_proc_file;
static char procfs_buffer[PROCFS_MAX_SIZE];
static unsigned long procfs_buffer_size = 0;

static ssize_t procfs_read(struct file *filp, char __user *buffer,
                           size_t length, loff_t *offset)
{
    static int finished = 0;

    if (finished) {
        pr_debug("procfs_read: END\n");
        finished = 0;
        return 0;
    }
    finished = 1;

    if (copy_to_user(buffer, procfs_buffer, procfs_buffer_size))
        return -EFAULT;

    pr_debug("procfs_read: read %lu bytes\n", procfs_buffer_size);
    return procfs_buffer_size;
}
static ssize_t procfs_write(struct file *file, const char __user *buffer,
                            size_t len, loff_t *off)
{
    if (len > PROCFS_MAX_SIZE)
        procfs_buffer_size = PROCFS_MAX_SIZE;
    else
        procfs_buffer_size = len;
    if (copy_from_user(procfs_buffer, buffer, procfs_buffer_size))
        return -EFAULT;

    pr_debug("procfs_write: write %lu bytes\n", procfs_buffer_size);
```

```
    return procfs_buffer_size;
}
static int procfs_open(struct inode *inode, struct file *file)
{
    try_module_get(THIS_MODULE);
    return 0;
}
static int procfs_close(struct inode *inode, struct file *file)
{
    module_put(THIS_MODULE);
    return 0;
}

#ifdef HAVE_PROC_OPS
static struct proc_ops file_ops_4_our_proc_file = {
    .proc_read = procfs_read,
    .proc_write = procfs_write,
    .proc_open = procfs_open,
    .proc_release = procfs_close,
};
#else
static const struct file_operations file_ops_4_our_proc_file = {
    .read = procfs_read,
    .write = procfs_write,
    .open = procfs_open,
    .release = procfs_close,
};
#endif

static int __init procfs3_init(void)
{
    our_proc_file = proc_create(PROCFS_ENTRY_FILENAME, 0644, NULL,
                                &file_ops_4_our_proc_file);
    if (our_proc_file == NULL) {
        remove_proc_entry(PROCFS_ENTRY_FILENAME, NULL);
        pr_debug("Error: Could not initialize /proc/%s\n",
                 PROCFS_ENTRY_FILENAME);
        return -ENOMEM;
    }
    proc_set_size(our_proc_file, 80);
    proc_set_user(our_proc_file, GLOBAL_ROOT_UID, GLOBAL_ROOT_GID);

    pr_debug("/proc/%s created\n", PROCFS_ENTRY_FILENAME);
    return 0;
}

static void __exit procfs3_exit(void)
{
```

```
    remove_proc_entry(PROCFS_ENTRY_FILENAME, NULL);
    pr_debug("/proc/%s removed\n", PROCFS_ENTRY_FILENAME);
}

module_init(procfs3_init);
module_exit(procfs3_exit);


MODULE_LICENSE("GPL");
```

Хотите больше примеров с `procfs`? Что ж, в первую очередь имейте ввиду, что по некоторой неофициальной информации `procfs` доживает свои дни, и нужно ориентироваться на использование `sysfs`.

Поэтому, если хотите самостоятельно задокументировать что-то связанное с ядром, то подумайте о применении именно этого механизма.

7.4 Управление файлом /proc с помощью seq_file

Как мы видели, создание файла в `/proc` может вызывать сложности. Поэтому в качестве вспомогательного средства существует API `seq_file`, который помогает форматировать файл `/proc` для вывода.

Основан этот API на выполнении последовательности из 3 функций: `start()`, `next()` и `stop()`.

Запускает `seq_file` эту последовательность, когда пользователь считывает файл `/proc`.

Начинается все с вызова функции `start()` – если она вернет не `NULL`, то вызывается функция `next()`.

Эта функция является итератором, перебирающим все данные. При каждом вызове `next()` также вызывается `show()`, которая записывает значения данных в буфер, считываемый пользователем.

Функция `next()` вызывается до тех пор, пока не вернет `NULL`, после чего последовательность завершается, и вызывается функция `stop()`.

Внимание! После окончания текущей последовательности начинается следующая. Это означает, что по завершению функции `stop()` снова вызывается `start()`. Заканчивается этот цикл, когда функция `start()` возвращает `NULL`. Общая схема описанного процесса показана на рис. 1. ниже

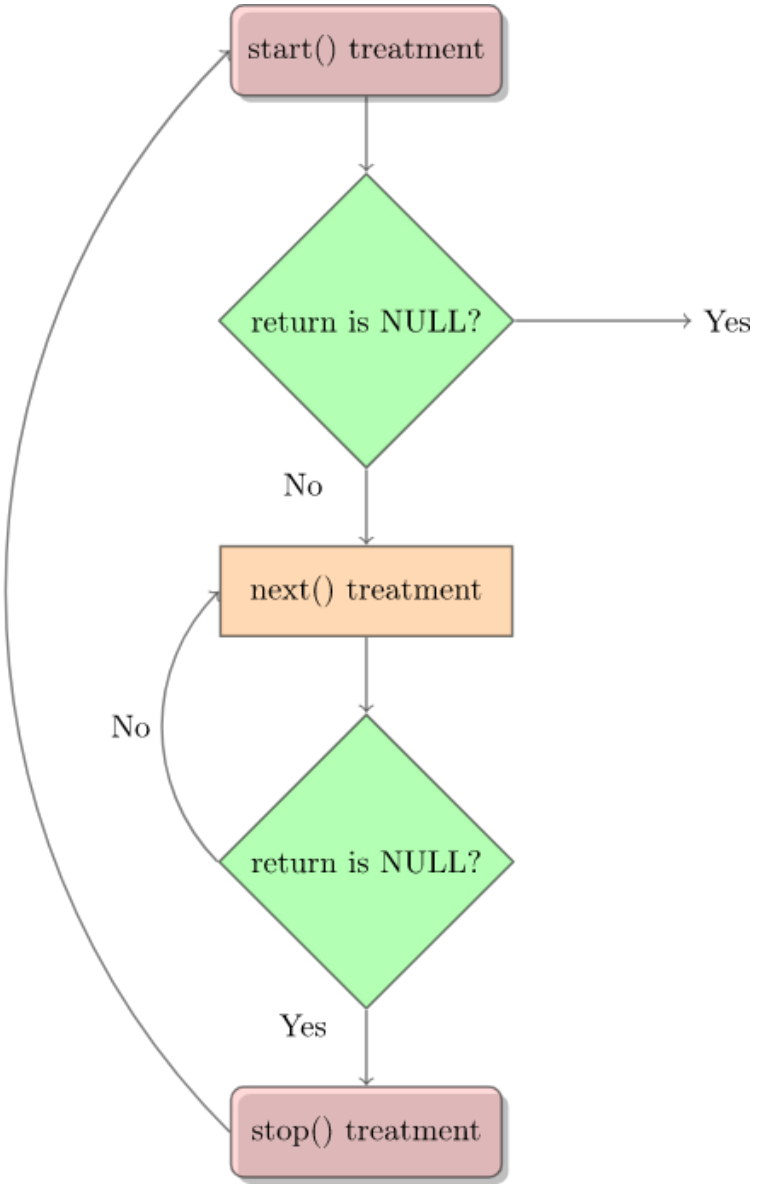


Рис. 1: принцип работы seq_file

Интерфейс `seq_file` предоставляет базовые функции для `proc_ops`, такие как `seq_read`, `seq_lseek` и некоторые другие, но ничего для выполнения записи в файл `/proc`. Хотя вы по-прежнему можете использовать способ из предыдущего примера.

Код procfs4.c:

```
/*
 * procfs4.c - создание "файла" в /proc
 * Эта программа задействует для управления файлом /proc библиотеку seq_file.
 */

#include <linux/kernel.h> /* Для работы с ядром. */
#include <linux/module.h> /* Для модулей. */
#include <linux/proc_fs.h> /* Для использования procfs */
#include <linux/seq_file.h> /* Для seq_file */
#include <linux/version.h>

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
```

```c
#define HAVE_PROC_OPS
#endif

#define PROC_NAME "iter"

/* Эта функция вызывается в начале последовательности.
 * То есть, когда:
 *   - первый раз считывается файл /proc
 *   - после завершения функции (в конце последовательности)
 */
static void *my_seq_start(struct seq_file *s, loff_t *pos)
{
    static unsigned long counter = 0;

    /* Начинаем новую последовательность? */
    if (*pos == 0) {
        /* Да => возвращается ненулевое значение для начала последовательности */
        return &counter;
    }

    /* Нет => это конец последовательности, возвращается NULL для завершения
считывания */
    *pos = 0;
    return NULL;
}

/* Эта функция вызывается после начала последовательности.
 * Ее вызов повторяется до возвращения значения NULL (затем последовательность
завершается).
 */
static void *my_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
    unsigned long *tmp_v = (unsigned long *)v;
    (*tmp_v)++;
    (*pos)++;
    return NULL;
}

/* Эта функция вызывается в конце последовательности. */
static void my_seq_stop(struct seq_file *s, void *v)
{
    /* Делать нечего, используем в start() статическое значение. */
}

/* Эта функция вызывается для каждого «шага» последовательности. */
static int my_seq_show(struct seq_file *s, void *v)
{
    loff_t *spos = (loff_t *)v;
```

```
    seq_printf(s, "%Ld\n", *spos);
    return 0;
}

/* Эта структура формирует "функцию" для управления последовательностью. */
static struct seq_operations my_seq_ops = {
    .start = my_seq_start,
    .next = my_seq_next,
    .stop = my_seq_stop,
    .show = my_seq_show,
};

/* Эта функция вызывается при открытии файла /proc. */
static int my_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &my_seq_ops);
};

/* Эта структура формирует “функцию”, управляющую файлом /proc. */
#ifdef HAVE_PROC_OPS
static const struct proc_ops my_file_ops = {
    .proc_open = my_open,
    .proc_read = seq_read,
    .proc_lseek = seq_lseek,
    .proc_release = seq_release,
};
#else
static const struct file_operations my_file_ops = {
    .open = my_open,
    .read = seq_read,
    .llseek = seq_lseek,
    .release = seq_release,
};
#endif

static int __init procfs4_init(void)
{
    struct proc_dir_entry *entry;

    entry = proc_create(PROC_NAME, 0, NULL, &my_file_ops);
    if (entry == NULL) {
        remove_proc_entry(PROC_NAME, NULL);
        pr_debug("Error: Could not initialize /proc/%s\n", PROC_NAME);
        return -ENOMEM;
    }

    return 0;
```

```
}

static void __exit procfs4_exit(void)
{
    remove_proc_entry(PROC_NAME, NULL);
    pr_debug("/proc/%s removed\n", PROC_NAME);
}

module_init(procfs4_init);
module_exit(procfs4_exit);


MODULE_LICENSE("GPL");
```

Если вас интересует дополнительная информация, рекомендую заглянуть на эти страницы:

- lwn.net/Articles/22355
- kernelnewbies.org/Documents/SeqFileHowTo

Также можете почитать код `fs/seq_file.c` в ядре.

8 sysfs: взаимодействие с модулем

`Sysfs` позволяет взаимодействовать с работающим ядром из пользовательского пространства, считывая или устанавливая переменные внутри модулей. Это может пригодиться в целях отладки или же в качестве интерфейса для приложений либо скриптов. Каталоги и файлы `sysfs` располагаются в */sys*:

```
ls -l /sys
```

Атрибуты для `kobjects` в этой файловой системе можно экспортировать в форме стандартных файлов. `Sysfs` перенаправляет файловые операции ввода-вывода в определенные для этих атрибутов методы, тем самым обеспечивая средства для считывания и записи атрибутов ядра.

Определение атрибута:

```
struct attribute {
    char *name;
    struct module *owner;
    umode_t mode;
```

```
};

int sysfs_create_file(struct kobject * kobj, const struct attribute * attr);
8void sysfs_remove_file(struct kobject * kobj, const struct attribute * attr);
```

К примеру, модель драйвера определяет struct device_attribute так:

```
struct device_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device *dev, struct device_attribute *attr,
                    char *buf);
    ssize_t (*store)(struct device *dev, struct device_attribute *attr,
                    const char *buf, size_t count);
};

int device_create_file(struct device *, const struct device_attribute *);
void device_remove_file(struct device *, const struct device_attribute *);
```

Чтобы иметь возможность читать и записывать атрибут, при его объявлении необходимо указать метод `show()` или `store()`.

Для распространенных случаев `include/linux/sysfs.h` предоставляет удобные макросы (`__ATTR`, `__ATTR_RO`, `__ATTR_WO`, и т.д.), упрощая определение атрибутов, а также позволяя сделать код более лаконичным и читаемым.

Вот пример модуля “Hello world”, который включает создание переменной, доступной через `sysfs`.

Код hello-sysfs.c:

```
/*
 * hello-sysfs.c - пример использования sysfs
 */
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/kobject.h>
#include <linux/module.h>
#include <linux/string.h>
#include <linux/sysfs.h>

static struct kobject *mymodule;

/* Переменная, которую нужно будет изменять. */
static int myvariable = 0;
```

```c
static ssize_t myvariable_show(struct kobject *kobj,
                               struct kobj_attribute *attr, char *buf)
{
    return sprintf(buf, "%d\n", myvariable);
}

static ssize_t myvariable_store(struct kobject *kobj,
                                struct kobj_attribute *attr, char *buf,
                                size_t count)
{
    sscanf(buf, "%du", &myvariable);
    return count;
}

static struct kobj_attribute myvariable_attribute =
    __ATTR(myvariable, 0660, myvariable_show, (void *)myvariable_store);

static int __init mymodule_init(void)
{
    int error = 0;

    pr_info("mymodule: initialised\n");

    mymodule = kobject_create_and_add("mymodule", kernel_kobj);
    if (!mymodule)
        return -ENOMEM;

    error = sysfs_create_file(mymodule, &myvariable_attribute.attr);
    if (error) {
        pr_info("failed to create the myvariable file "
                "in /sys/kernel/mymodule\n");
    }

    return error;
}

static void __exit mymodule_exit(void)
{
    pr_info("mymodule: Exit success\n");
    kobject_put(mymodule);
}

module_init(mymodule_init);
module_exit(mymodule_exit);


MODULE_LICENSE("GPL");
```

Компиляция и установка модуля:

```
make
sudo insmod hello-sysfs.ko
```

Убеждаемся в успешности операции:

```
sudo lsmod | grep hello_sysfs
```

Каково текущее значение `myvariable`?

```
cat /sys/kernel/mymodule/myvariable
```

Установка значения `myvariable` и проверка, изменилось ли оно:

```
echo "32" > /sys/kernel/mymodule/myvariable
cat /sys/kernel/mymodule/myvariable
```

Наконец, извлечение тестового модуля:

```
sudo rmmod hello_sysfs
```

В случае выше мы используем для создания каталога в `sysfs` и взаимодействия с его атрибутами простой `kobject`. Начиная с Linux v2.6.0, структура `kobject` постепенно обретала свой нынешний облик.

Изначально она подразумевалась как простой способ унификации кода ядра, управляющего объектами с подсчетом ссылок. Однако спустя некоторое время ее назначение расширилось, и теперь она связывает большую часть модели устройства и ее интерфейса `sysfs`.

Подробнее о `kobject` и `sysfs` читайте в Documentation/driver-api/driver-model/driver.rst и lwn.net/Articles/51437.

9. Взаимодействие с файлами устройств

Файлы устройств представляют физические устройства. Большинство таких устройств используются для вывода и ввода, а значит необходим некий механизм, который бы позволил их находящимся в ядре драйверам получать вывод от процессов для его перенаправления самим устройствам. Для этого файл открывается, и в него производится

запись, в точности аналогично стандартной операции записи в файл. В примере ниже это реализовано с помощью `device_write`.

Но этого не всегда оказывается достаточно. Представьте, что у вас к последовательному порту подключен модем (даже если модем внутренний, эта схема с точки зрения процессора все равно реализуется как модем, подключенный к последовательному порту, так что воображение особо напрягать не нужно).

Естественным решением здесь будет использовать файл устройства как для записи на модем (к примеру, команд или данных для отправки), так и для чтения с него (например, ответов на команды или полученных данных). Тем не менее остается вопрос о том, что же делать, когда нужно взаимодействовать с самим последовательным портом, например, для настройки скорости отправки/получения данных.

В Unix ответом будет использовать специальную функцию `ioctl` (сокращенно от Input Output ConTroL). Каждое устройство может иметь собственные команды `ioctl`, реализующие чтение (для отправки информации от процесса ядру), запись (для возвращения информации процессу), и то и другое, либо ни одно из этих действий. Имейте ввиду, что в `ioctl` роли чтения и записи снова реверсируются, то есть при чтении происходит отправка информации ядру, а при записи ее получение от ядра.

Вызывается функция `ioctl` с тремя параметрами: дескриптором соответствующего файла устройства, номером `ioctl` и параметром, имеющим тип `long`, чтобы можно было использовать приведение, позволяющее с его помощью передавать почти все, что захочется.

Таким способом не удастся передать структуру, но можно будет передать указатель на нее. Вот пример:

Код ioctl.c:

```
/*
 * ioctl.c
 */
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/ioctl.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/uaccess.h>

struct ioctl_arg {
    unsigned int val;
};
```

```c
/* Documentation/ioctl/ioctl-number.txt */
#define IOC_MAGIC '\x66'

#define IOCTL_VALSET _IOW(IOC_MAGIC, 0, struct ioctl_arg)
#define IOCTL_VALGET _IOR(IOC_MAGIC, 1, struct ioctl_arg)
#define IOCTL_VALGET_NUM _IOR(IOC_MAGIC, 2, int)
#define IOCTL_VALSET_NUM _IOW(IOC_MAGIC, 3, int)

#define IOCTL_VAL_MAXNR 3
#define DRIVER_NAME "ioctltest"

static unsigned int test_ioctl_major = 0;
static unsigned int num_of_dev = 1;
static struct cdev test_ioctl_cdev;
static int ioctl_num = 0;

struct test_ioctl_data {
    unsigned char val;
    rwlock_t lock;
};

static long test_ioctl_ioctl(struct file *filp, unsigned int cmd,
                             unsigned long arg)
{
    struct test_ioctl_data *ioctl_data = filp->private_data;
    int retval = 0;
    unsigned char val;
    struct ioctl_arg data;
    memset(&data, 0, sizeof(data));

    switch (cmd) {
    case IOCTL_VALSET:
        if (copy_from_user(&data, (int __user *)arg, sizeof(data))) {
            retval = -EFAULT;
            goto done;
        }

        pr_alert("IOCTL set val:%x .\n", data.val);
        write_lock(&ioctl_data->lock);
        ioctl_data->val = data.val;
        write_unlock(&ioctl_data->lock);
        break;

    case IOCTL_VALGET:
        read_lock(&ioctl_data->lock);
        val = ioctl_data->val;
        read_unlock(&ioctl_data->lock);
        data.val = val;
```

```
        if (copy_to_user((int __user *)arg, &data, sizeof(data))) {
            retval = -EFAULT;
            goto done;
        }

        break;

    case IOCTL_VALGET_NUM:
        retval = __put_user(ioctl_num, (int __user *)arg);
        break;

    case IOCTL_VALSET_NUM:
        ioctl_num = arg;
        break;

    default:
        retval = -ENOTTY;
    }

done:
    return retval;
}

static ssize_t test_ioctl_read(struct file *filp, char __user *buf,
                               size_t count, loff_t *f_pos)
{
    struct test_ioctl_data *ioctl_data = filp->private_data;
    unsigned char val;
    int retval;
    int i = 0;

    read_lock(&ioctl_data->lock);
    val = ioctl_data->val;
    read_unlock(&ioctl_data->lock);

    for (; i < count; i++) {
        if (copy_to_user(&buf[i], &val, 1)) {
            retval = -EFAULT;
            goto out;
        }
    }

    retval = count;
out:
    return retval;
}
```

```
static int test_ioctl_close(struct inode *inode, struct file *filp)
{
    pr_alert("%s call.\n", __func__);

    if (filp->private_data) {
        kfree(filp->private_data);
        filp->private_data = NULL;
    }

    return 0;
}

static int test_ioctl_open(struct inode *inode, struct file *filp)
{
    struct test_ioctl_data *ioctl_data;

    pr_alert("%s call.\n", __func__);
    ioctl_data = kmalloc(sizeof(struct test_ioctl_data), GFP_KERNEL);

    if (ioctl_data == NULL)
        return -ENOMEM;

    rwlock_init(&ioctl_data->lock);
    ioctl_data->val = 0xFF;
    filp->private_data = ioctl_data;

    return 0;
}

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = test_ioctl_open,
    .release = test_ioctl_close,
    .read = test_ioctl_read,
    .unlocked_ioctl = test_ioctl_ioctl,
};

static int ioctl_init(void)
{
    dev_t dev;
    int alloc_ret = -1;
    int cdev_ret = -1;
    alloc_ret = alloc_chrdev_region(&dev, 0, num_of_dev, DRIVER_NAME);

    if (alloc_ret)
        goto error;

    test_ioctl_major = MAJOR(dev);
```

```
    cdev_init(&test_ioctl_cdev, &fops);
    cdev_ret = cdev_add(&test_ioctl_cdev, dev, num_of_dev);

    if (cdev_ret)
        goto error;

    pr_alert("%s driver(major: %d) installed.\n", DRIVER_NAME,
             test_ioctl_major);
    return 0;
error:
    if (cdev_ret == 0)
        cdev_del(&test_ioctl_cdev);
    if (alloc_ret == 0)
        unregister_chrdev_region(dev, num_of_dev);
    return -1;
}

static void ioctl_exit(void)
{
    dev_t dev = MKDEV(test_ioctl_major, 0);

    cdev_del(&test_ioctl_cdev);
    unregister_chrdev_region(dev, num_of_dev);
    pr_alert("%s driver removed.\n", DRIVER_NAME);
}

module_init(ioctl_init);
module_exit(ioctl_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("This is test_ioctl module");
```

Вы можете заметить в функции `test_ioctl_ioctl()` аргумент `cmd`. Это номер `ioctl`. Он кодирует старший (major) номер устройства, тип `ioctl`, команду и тип параметра. Обычно этот номер создается вызовом макроса (`_IO`, `_IOR`, `_IOW` или `_IOWR` — в зависимости от типа) в заголовочном файле.

Этот заголовочный файл должен быть включен и в программы, которые будут использовать `ioctl` (чтобы они могли генерировать подходящие `ioctl`), и в модуль ядра (чтобы он мог эту функцию понимать).
В примере ниже заголовочным файлом является `chardev.h`, а использующей его программой `userspace_ioctl.c`.

Если вы хотите использовать `ioctl` в собственных модулях, то лучше будет получить для нее официальное назначение. Тогда, если у вас каким-то образом окажется чужая `ioctl`, то сразу станет понятно, что что-то не так.

Более подробную информацию можно получить в дереве исходного кода ядра на странице Documentation/userspace-api/ioctl/ioctl-number.rst

Кроме того, необходимо иметь ввиду, что конкурентное обращение к ресурсам приведет к состоянию гонки. Решением будет использовать атомарную инструкцию сравнения с обменом (CAS), которая упоминалась в разделе 6.5, чтобы организовать индивидуальный доступ.

Код chardev2.c:

```
/*
 * chardev2.c - создание символьного устройства ввода/вывода
 */

#include <linux/cdev.h>
#include <linux/delay.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/irq.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/poll.h>

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

enum {
    CDEV_NOT_USED = 0,
    CDEV_EXCLUSIVE_OPEN = 1,
};

/* Открыто ли сейчас устройство? Служит для предотвращения
 * конкурентного доступа к одному устройству.
 */
static atomic_t already_open = ATOMIC_INIT(CDEV_NOT_USED);

/* Сообщение, которое устройство будет выдавать при обращении. */
static char message[BUF_LEN];

static struct class *cls;

/* Вызывается, когда процесс пытается открыть файл устройства. */
static int device_open(struct inode *inode, struct file *file)
```

```
{
    pr_info("device_open(%p)\n", file);

    try_module_get(THIS_MODULE);
    return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
    pr_info("device_release(%p,%p)\n", inode, file);

    module_put(THIS_MODULE);
    return SUCCESS;
}

/* Эта функция вызывается, когда процесс, уже открывший файл,
 * пытается считать из него.
 */
static ssize_t device_read(struct file *file, /* см. include/linux/fs.h   */
                           char __user *buffer, /* Буфер для заполнения.  */
                           size_t length, /* Длина буфера.     */
                           loff_t *offset)
{
    /* Количество байтов, фактически записываемых в буфер. */
    int bytes_read = 0;
    /* Как далеко зашел процесс, считывающий
     * сообщение? Пригождается, когда сообщение больше размера буфера
     * в device_read.
     */
    const char *message_ptr = message;

    if (!*(message_ptr + *offset)) { /* Мы в конце сообщения. */
        *offset = 0; /* Сброс смещения. */
        return 0; /* Обозначение конца файла. */
    }

    message_ptr += *offset;

    /* Фактически помещает данные в буфер. */
    while (length && *message_ptr) {
        /* Поскольку буфер находится в пользовательском сегменте данных,
         * а не в сегменте ядра, присваивание не сработает. Вместо этого
         * нужно использовать put_user, которая скопирует данные из
         * сегмента ядра в сегмент пользователя.
         */
        put_user(*(message_ptr++), buffer++);
        length--;
        bytes_read++;
```

```
    }

    pr_info("Read %d bytes, %ld left\n", bytes_read, length);

    *offset += bytes_read;

    /* Функции чтения должны возвращать количество байтов, реально
     * вставляемых в буфер.
     */
    return bytes_read;
}

/* Вызывается, когда кто-то пытается произвести запись в файл устройства. */
static ssize_t device_write(struct file *file, const char __user *buffer,
                            size_t length, loff_t *offset)
{
    int i;

    pr_info("device_write(%p,%p,%ld)", file, buffer, length);

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

    /* Также возвращает количество использованных во вводе символов. */
    return i;
}

/* Эта функция вызывается, когда процесс пытается выполнить ioctl для
 * файла устройства. Мы получаем два дополнительных параметра
 * (дополнительных для структур inode и file, которые получают все
 * функции устройств): номер ioctl и параметр, заданный для этой ioctl.
 *
 * Если ioctl реализует запись или запись/чтение (то есть ее вывод
 * возвращается вызывающему процессу), вызов ioctl возвращает вывод
 * этой функции.
 */
static long
device_ioctl(struct file *file, /* То же самое. */
             unsigned int ioctl_num, /* Число и параметр для ioctl */
             unsigned long ioctl_param)
{
    int i;
    long ret = SUCCESS;

    /* Мы не хотим взаимодействовать с двумя процессами одновременно */
    if (atomic_cmpxchg(&already_open, CDEV_NOT_USED, CDEV_EXCLUSIVE_OPEN))
        return -EBUSY;
```

```
    /* Переключение согласно вызванной ioctl. */
    switch (ioctl_num) {
    case IOCTL_SET_MSG: {
        /* Получение указателя на сообщение (в пользовательском
         * пространстве) и установка его как сообщения устройства.
         * Получение параметра, передаваемого ioctl процессом.
         */
        char __user *tmp = (char __user *)ioctl_param;
        char ch;

        /* Определение длины сообщения. */
        get_user(ch, tmp);
        for (i = 0; ch && i < BUF_LEN; i++, tmp++)
            get_user(ch, tmp);

        device_write(file, (char __user *)ioctl_param, i, NULL);
        break;
    }
    case IOCTL_GET_MSG: {
        loff_t offset = 0;

        /* Передача текущего сообщения вызывающему процессу. Получаемый
         * параметр является указателем, который мы заполняем.
         */
        i = device_read(file, (char __user *)ioctl_param, 99, &offset);

        /* Помещаем в конец буфера нуль, чтобы он правильно завершился.
         */
        put_user('\0', (char __user *)ioctl_param + i);
        break;
    }
    case IOCTL_GET_NTH_BYTE:
        /* Эта ioctl является и вводом (ioctl_param), и выводом
         * (возвращаемым значением этой функции).
         */
        ret = (long)message[ioctl_param];
        break;
    }

    /* Теперь можно принимать следующий вызов. */
    atomic_set(&already_open, CDEV_NOT_USED);

    return ret;
}

/* Объявления модулей. */

/* Эта структура будет хранить функции, вызываемые при выполнении
```

```
 * процессом действий с созданным нами устройством. Поскольку указатель
 * на эту структуру содержится в таблице устройств, он не может быть
 * локальным для init_module. NULL для не реализованных функций.
 */
static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .unlocked_ioctl = device_ioctl,
    .open = device_open,
    .release = device_release, /* Аналогично закрытию. */
};

/* Инициализация модуля – регистрация символьного устройства. */
static int __init chardev2_init(void)
{
    /* Регистрация символьного устройства (по меньшей мере попытка). */
    int ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);

    /* Отрицательные значения означают ошибку. */
    if (ret_val < 0) {
        pr_alert("%s failed with %d\n",
                 "Sorry, registering the character device ", ret_val);
        return ret_val;
    }

    cls = class_create(THIS_MODULE, DEVICE_FILE_NAME);
    device_create(cls, NULL, MKDEV(MAJOR_NUM, 0), NULL, DEVICE_FILE_NAME);

    pr_info("Device created on /dev/%s\n", DEVICE_FILE_NAME);

    return 0;
}

/* Очистка – снятие регистрации соответствующего файла из /proc. */
static void __exit chardev2_exit(void)
{
    device_destroy(cls, MKDEV(MAJOR_NUM, 0));
    class_destroy(cls);

    /* Снятие регистрации устройства. */
    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
}

module_init(chardev2_init);
module_exit(chardev2_exit);


MODULE_LICENSE("GPL");
```

Код chardev.h:

```
/*
 * chardev.h - заголовочный файл с определениями ioctl.
 *
 * Объявления нужны в заголовочном файле, поскольку их должен знать
 * как модуль ядра (из chardev2.c), так и процесс, вызывающий ioctl()
 * (из userspace_ioctl.c).
 */

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/* Старший номер устройства. Мы больше не можем полагаться на
 * динамическую регистрацию, поскольку функции ioctl должны его знать.
 */
#define MAJOR_NUM 100

/* Установка сообщения драйвера устройства. */
#define IOCTL_SET_MSG _IOW(MAJOR_NUM, 0, char *)
/* _IOW означает, что мы создаем номер команды ioctl для передачи
 * информации от пользовательского процесса модулю ядра.
 *
 * Первый аргумент, MAJOR_NUM, это используемый старший номер устройства
 *
 * Второй аргумент - это номер команды (их может быть несколько с
 * разными смыслами).
 *
 * Третий аргумент - это тип, который мы хотим передать от процесса ядру
 */

/* Получение сообщения драйвера устройства. */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
/* Эта IOCTL используется для вывода с целью получить сообщение
 * драйвера устройства. Однако нам все равно нужен буфер для размещения
 * этого сообщения в качестве ввода при его передаче процессом.
 */

/* Получение n-ного байта сообщения. */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
/* Эта IOCTL используется как для ввода, так и для вывода. Она получает
 * от пользователя число, n, и возвращает message[n].
 */

/* Имя файла устройства. */
```

```
#define DEVICE_FILE_NAME "char_dev"
#define DEVICE_PATH "/dev/char_dev"


#endif
```

Код userspace_ioctl.c:

```
/*  userspace_ioctl.c - процесс, позволяющий контролировать модуль ядра
 *  с помощью ioctl.
 *
 *  До этого момента можно было использовать для ввода и вывода cat.
 *  Теперь необходимо использовать ioctl, для чего нужно написать свой
 *  процесс.
 */

/* Детали устройства, такие как номера ioctl и старший файл устройства.  */
#include "../chardev.h"

#include <stdio.h> /* Стандартный ввод-вывод. */
#include <fcntl.h> /* Открытие. */
#include <unistd.h> /* Закрытие. */
#include <stdlib.h> /* Выход. */
#include <sys/ioctl.h> /* ioctl */

/* Функции для вызовов ioctl. */

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

    ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

    if (ret_val < 0) {
        printf("ioctl_set_msg failed:%d\n", ret_val);
    }

    return ret_val;
}

int ioctl_get_msg(int file_desc)
{
    int ret_val;
    char message[100] = { 0 };

  /* Внимание! Это опасно, так как мы не сообщаем ядру, до куда
   * можно производить запись, то есть рискуем вызвать переполнение
```

```
   * буфера. В реальной программе мы бы использовали две ioctl - одну
   * для информирования ядра о длине буфера и вторую для предоставления
   * ему самого буфера под заполнение.
   */
    ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

    if (ret_val < 0) {
        printf("ioctl_get_msg failed:%d\n", ret_val);
    }
    printf("get_msg message:%s", message);

    return ret_val;
}

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

    printf("get_nth_byte message:");

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

        if (c < 0) {
            printf("\nioctl_get_nth_byte failed at the %d'th byte:\n", i);
            return c;
        }

        putchar(c);
    } while (c != 0);

    return 0;
}

/* Main - вызов функций ioctl. */
int main(void)
{
    int file_desc, ret_val;
    char *msg = "Message passed by ioctl\n";

    file_desc = open(DEVICE_PATH, O_RDWR);
    if (file_desc < 0) {
        printf("Can't open device file: %s, error:%d\n", DEVICE_PATH,
               file_desc);
        exit(EXIT_FAILURE);
    }
```

```
    ret_val = ioctl_set_msg(file_desc, msg);
    if (ret_val)
        goto error;
    ret_val = ioctl_get_nth_byte(file_desc);
    if (ret_val)
        goto error;
    ret_val = ioctl_get_msg(file_desc);
    if (ret_val)
        goto error;

    close(file_desc);
    return 0;
error:
    close(file_desc);
    exit(EXIT_FAILURE);
}
```

10. Системные вызовы

До этого момента мы просто использовали отлаженные механизмы ядра для регистрации файлов `/proc` и обработчиков устройств. И это отличное решение, когда вы хотите сделать нечто стандартное, что предполагали программисты ядра, например написать драйвер устройства. Но как быть, если вы планируете сделать что-то необычное, каким-то образом изменить поведение системы? Здесь вам придется действовать полностью самостоятельно.

И если вы пойдете рисковым путем, не задействовав виртуальную машину, то программирование ядра уже может стать весьма опасным занятием. В процессе написания примера, который вы увидите ниже, я убил системный вызов `open()` и в итоге не смог открывать какие-либо файлы, запускать программы и даже выключить систему. Пришлось перезапускать виртуальную машину. Никакие важные файлы не пострадали, но если бы я делал то же самое в реальной критически важной системе, то такой исход оказался бы вполне вероятен. Чтобы оградить себя от возможной потери каких-либо файлов, даже в рамках тестовой среды, рекомендую до выполнения `insmod` и `rmmod` выполнять `sync`.

Забудьте о файлах `/proc`, забудьте о файлах устройств – это лишь мелкие детали в необъятном пространстве вселенной. Реальным механизмом в контексте взаимодействия с ядром, тем, который используют все процессы, являются системные вызовы. Когда процесс запрашивает у ядра операцию (например, открытие файла, разветвление на новый процесс или выделение дополнительной памяти), используется именно этот механизм.

Если вы хотите изменить поведение ядра, то вмешательство производится как раз в него. Кстати, посмотреть, какие системные вызовы использует программа, можно так:

```
strace <arguments>
```

Как правило, процесс не должен иметь возможности обращаться к ядру, то есть он не может обращаться к его памяти и вызывать его функции. Это обусловлено аппаратной спецификой ЦПУ (именно поэтому мы говорим «защищенный режим» или «защита страниц»).

И системные вызовы являются в этом общем правиле исключением. Технически это происходит так – процесс заполняет регистр нужными значениями, после чего вызывает особую инструкцию, которая переключается на ранее определенную область ядра (естественно, эта область доступна пользовательским процессам только для чтения, но не для записи). В случае процессоров Intel это происходит посредством прерывания `0x80`. Аппаратное обеспечение знает, что при переходе в эту область вы выходите из ограниченного пользовательского режима, начиная действовать уже в юрисдикции ядра, в связи с чем для вас открываются все двери.

Область ядра, в которую может перескочить процесс, называется `system_call`. Процедура в этой области проверяет номер системного вызова, сообщая ядру о том, какое действие процесс запросил. Затем она просматривает таблицу системных вызовов (`sys_call_table`) в поиске адреса нужной функции ядра. Далее она эту функцию вызывает, и после того, как та возвращает результат, выполняет ряд системных проверок и делает возврат процессу (или к другому процессу, если время изначального истекло). Прописан весь этот код в файле `arch/$(architecture)/kernel/entry.S` после строки `ENTRY(system_call)`.

Итак, если мы хотим изменить способ работы определенного системного вызова, то нам нужно написать собственную функцию для его реализации (обычно для этого добавляется собственный код, после чего вызывается оригинальная функция), а затем перевести указатель в `sys_call_table` на нашу новую функцию. Поскольку позднее эта функция может быть удалена, важно, чтобы `cleanup_module` восстанавливала таблицу в исходное состояние.

Для изменения содержимого `sys_call_table` необходимо взять во внимание регистр управления. Это регистр процессора, который изменяет или управляет его общим функционированием. В архитектуре x86 у регистра `cr0` есть различные флаги управления, которые изменяют основные операции процессора. Среди них есть флаг `WP`, который отвечает за защиту от записи. Если он установлен, процессор запрещает выполнение записи в разделы только для чтения. Следовательно, прежде чем изменять `sys_call_table`, нам нужно этот флаг отключить.

Начиная с Linux v5.3, функцию `write_cr0` использовать нельзя из-за чувствительных бит `cr0`, представляющих угрозу безопасности. С их помощью атакующий может производить

запись в регистры управления, отключая защиты ЦПУ, например защиту от записи. В итоге для обхода этого ограничения необходимо предоставить собственную подпрограмму ассемблера.

Однако символ `sys_call_table` с целью предотвращения подобного трюка является неэкспортируемым. Но у нас все же есть пара способов для его получения, а именно ручной поиск символов и `kallsyms_lookup_name`. Здесь мы рассмотрим использование обоих, в зависимости от версии ядра.

В ядре используется механизм Control-Flow Integrity (CFI), предназначенный для предотвращения возможного перенаправления исполнения кода атакующим. Он позволяет гарантировать направление косвенных вызовов к ожидаемым адресам, а также неизменность возвращаемых адресов. Начиная с Linux v5.7, в ядре пропатчили ряд методов Control-Flow Enforcement (CET) для архитектуры x86, и некоторые конфигурации GCC, например GCC 9 и 10 версии, будут идти с CET (опция `-fcf-protection`) по умолчанию. Использование этого GCC для компиляции ядра при отключенном Retpoline может привести к активации в ядре функционала CET.

Проверить, включена ли опция `-fcf-protection`, можно следующей командой:

```
$ gcc -v -Q -O2 --help=target | grep protection
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
...
gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)
COLLECT_GCC_OPTIONS='-v' '-Q' '-O2' '--help=target' '-mtune=generic'
'-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/9/cc1 -v ... -fcf-protection ...
 GNU C17 (Ubuntu 9.3.0-17ubuntu1~20.04) version 9.3.0 (x86_64-linux-gnu)
...
```

Но CET не должен быть включен в ядре, поскольку это может нарушить работу Kprobes и bpf. По этой причине, начиная с v5.11, данный функционал был отключен. Поэтому, чтобы у нас гарантированно работал ручной поиск символов, мы используем версии ядра до v5.4.

К сожалению, начиная с Linux v5.7, `kallsyms_lookup_name` также не экспортируется, и получить адрес этой функции можно лишь обходным путем. Если включена опция `CONFIG_KPROBES`, можно извлечь этот адрес, используя Kprobes для динамического проникновения в определенную подпрограмму ядра. Kprobe вставляет точку останова на входе функции, заменяя первые байты просматриваемой инструкции. Когда ЦПУ достигает этой точки останова, регистры сохраняются, и управление переходит Kprobes. Он передает адреса сохраненных регистров и структуры Kprobe заданному нами обработчику, после чего его запускает. Kprobes можно зарегистрировать по имени

символа или адресу. При использовании имени символа адрес будет обрабатываться ядром.

В противном случае указать адрес `sys_call_table` из `/proc/kallsyms` и `/boot/System.map` в параметре `sym`. Вот пример использования `/proc/kallsyms`:

```
$ sudo grep sys_call_table /proc/kallsyms
ffffffff82000280 R x32_sys_call_table
ffffffff820013a0 R sys_call_table
ffffffff820023e0 R ia32_sys_call_table
$ sudo insmod syscall.ko sym=0xffffffff820013a0
```

При использовании адреса из `/boot/System.map` будьте внимательны к `KASLR` (рандомизация адресного пространства ядра). Этот механизм может рандомизировать адреса кода ядра и данных при каждой загрузке. К примеру, статический адрес, указанный в `/boot/System.map`, сдвигается на некоторое случайное значение.

Задача `KASLR` – защищать пространство ядра от атак. В случае отсутствия этого механизма атакующему было бы проще отыскать фиксированный целевой адрес, после чего с помощью возвратно-ориентированного программирования вставить вредоносный код для выполнения или получения нужных данных по фиктивному указателю.

`KASLR` противостоит подобным атакам, не позволяя атакующему прямиком узнать целевой адрес. Хотя метод подбора здесь все еще может сработать. Если адрес символа в `/proc/kallsyms` отличается от адреса в `/boot/System.map`, ядро активирует `KASLR`.

```
$ grep GRUB_CMDLINE_LINUX_DEFAULT /etc/default/grub
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
$ sudo grep sys_call_table /boot/System.map-$(uname -r)
ffffffff82000300 R sys_call_table
$ sudo grep sys_call_table /proc/kallsyms
ffffffff820013a0 R sys_call_table
# Перезагрузка
$ sudo grep sys_call_table /boot/System.map-$(uname -r)
ffffffff82000300 R sys_call_table
$ sudo grep sys_call_table /proc/kallsyms
ffffffff86400300 R sys_call_table
```

Если `KASLR` активна, то после каждой перезагрузки необходимо уделять внимание адресу из `/proc/kallsyms`.

Поэтому для использования адреса из `/boot/System.map` нужно будет убедиться, что `KASLR` отключена.

Отключить его можно, добавив при очередной загрузке параметр `nokaslr`:

```
$ grep GRUB_CMDLINE_LINUX_DEFAULT /etc/default/grub
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
$ sudo perl -i -pe 'm/quiet/ and s//quiet nokaslr/' /etc/default/grub
$ grep quiet /etc/default/grub
GRUB_CMDLINE_LINUX_DEFAULT="quiet nokaslr splash"
$ sudo update-grub
```

Дополнительно по этой теме читайте:

- Cook: Security things in Linux v5.3
- Unexporting the system call table
- Control-flow integrity for the kernel
- Unexporting kallsyms_lookup_name()
- Kernel Probes (Kprobes)
- Kernel address space layout randomization

Приведенный здесь исходный код является примером подобного модуля ядра. Мы хотим реализовать «слежку» за определенным пользователем и выводить (`pr_info()`) сообщение при каждом открытии им файла. С этой целью мы заменяем системный вызов `sys_open` собственной функцией `our_sys_open`. Эта функция проверяет uid текущего процесса, и если оно совпадает с uid, за которым мы следим, вызывает `pr_info()` для отображения имени открываемого файла. Затем она в любом случае вызывает оригинальную функцию `sys_open` с теми же параметрами, чтобы уже реально открыть файл.

Функция `init_module` заменяет нужную запись в `sys_call_table` и сохраняет оригинальный указатель в переменной.

В последствии функция `cleanup_module` использует эту переменную для восстановления исходного состояния таблицы. Это опасный подход, поскольку есть вероятность, что два модуля изменят один и тот же системный вызов.

Представьте, что у нас есть два модуля, *A* и *B*. У *A* системным вызовом для открытия файла будет `A_open`, а у *B* им будет `B_open`. Теперь, при внедрении *A* в ядро соответствующий системный вызов будет заменен на `A_open`, который по завершению вызовет `sys_open`. Далее в ядро внедряется *B*, заменяя системный вызов *A* на `B_open`, который по завершению вызовет тот самый `A_open`, являющийся для него оригинальным.

Теперь, если первым будет удален *B*, то все нормально – он просто восстановит системный вызов на `A_open`, который вызывает оригинал. Но вот если сначала удалить *A*, а затем *B*, то в системе произойдет сбой. Удаление *A* приведет к восстановлению исходного системного вызова, `sys_open`, исключив из процесса *B*.

Затем, когда будет удаляться *B*, он постарается восстановить системный вызов на тот, который считает исходным, то есть `A_open`, но его в памяти уже не окажется.

На первый взгляд эту проблему можно разрешить, проверив, совпадает ли системный вызов с нашей функцией открытия – если да, то просто его не менять (чтобы *B* не трогал системный вызов при удалении). Но это создаст еще большую проблему.

При удалении *A* увидит, что системный вызов был заменен на `B_open` и больше не указывает на `A_open`, а значит не станет восстанавливать его на `sys_open`, пока не будет удален из памяти.

К сожалению, в итоге *B* по-прежнему будет пытаться вызвать `A_Open`, которого больше нет, так что даже без удаления *B* система все равно даст сбой.

Заметьте, что все описанные проблемы делают перехват системных вызовов нецелесообразным для использования в продакшн-среде. И для того, чтобы оградить людей от совершения потенциально вредных действий, `sys_call_table` больше не экспортируется.

То есть, если вы хотите сделать что-то большее, нежели просто выполнить этот пример, то вам потребуется пропатчить ядро, чтобы вернуть поддержку экспорта `sys_call_table`.

Код syscall.c:

```
/*
 * syscall.c
 *
 * Пример перехвата системного вызова.
 *
 * Отключает защиту страниц на уровне процессора путем изменения
 * 16 бита в регистре cr0 (возможно, относится только к Intel).
 *
 * Основан на примере Питера Джея Зальцмана и
 * https://bbs.archlinux.org/viewtopic.php?id=139406
 */

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

```c
#include <linux/moduleparam.h> /* Будет содержать параметры. */
#include <linux/unistd.h> /* Список системных вызовов. */
#include <linux/version.h>

/* Для текущей структуры (процесса). Необходимы для понимания, кто
 * является текущим пользователем.
 */
#include <linux/sched.h>
#include <linux/uaccess.h>

/* По ходу изменения ядра изменяется и способ обращения к "sys_call_table"
 * - до v5.4       : ручной поиск символов
 * - с v5.5 по v5.6: использование kallsyms_lookup_name()
 * - v5.7+         : Kprobes либо определенный параметр модуля ядра
 */

/* В Linux v5.11+ внутренние вызовы ядра к ksys_close() были удалены.
 */
#if (LINUX_VERSION_CODE < KERNEL_VERSION(5, 7, 0))

#if LINUX_VERSION_CODE <= KERNEL_VERSION(5, 4, 0)
#define HAVE_KSYS_CLOSE 1
#include <linux/syscalls.h> /* Для ksys_close() */
#else
#include <linux/kallsyms.h> /* Для kallsyms_lookup_name */
#endif

#else

#if defined(CONFIG_KPROBES)
#define HAVE_KPROBES 1
#include <linux/kprobes.h>
#else
#define HAVE_PARAM 1
#include <linux/kallsyms.h> /* Для sprint_symbol */
/* Адрес sys_call_table, который можно получить поиском по
 * "/boot/System.map" или "/proc/kallsyms". В ядре v5.7+, когда
 * CONFIG_KPROBES отсутствует, можно вводить этот параметр, иначе модуль
 * будет производить поиск по всей памяти.
 */
static unsigned long sym = 0;
module_param(sym, ulong, 0644);
#endif /* CONFIG_KPROBES */

#endif /* Version < v5.7 */

static unsigned long **sys_call_table;
```

```c
/* UID, за которым мы хотим следить - будет заполняться из командной строки. */
static int uid;
module_param(uid, int, 0644);

/* Указатель на исходный системный вызов. Мы сохраняем его, а не
 * вызываем исходную функцию (sys_open), так как кто-то другой мог
 * заменить этот системный вызов до нас. Заметьте, что это не гарантирует
 * 100% безопасность, поскольку, если до этого sys_open уже был
 * заменен другим модулем, то внедрение нашего приведет
 * к вызову функции в том модуле - а он может быть уже удален.
 *
 * Еще одна причина в том, что мы не можем получить sys_open, поскольку
 * это статическая переменная, и она не экспортируется.
 */
static asmlinkage int (*original_call)(const char *, int, int);

/* Функция, которой мы заменяем sys_open. Для нахождения точного прототипа
 * с числом и типом аргументов сначала мы находим исходную функцию
 * (в fs/open.c).
 *
 * В теории это означает, что мы привязаны к текущей версии ядра. На
 * практике же системные вызовы почти никогда не меняются (это бы внесло
 * беспорядок и потребовало перекомпиляции программ, так как системные вызовы
 * являются интерфейсом между ядром и процессами).
 */
static asmlinkage int our_sys_open(const char *filename, int flags, int mode)
{
    int i = 0;
    char ch;

    /* В случае соответствия сообщить об открытом файле. */
    pr_info("Opened file by %d: ", uid);
    do {
        get_user(ch, (char __user *)filename + i);
        i++;
        pr_info("%c", ch);
    } while (ch != 0);
    pr_info("\n");

    /* Вызов исходной sys_open - иначе мы потеряем возможность открывать
     * файлы.
     */
    return original_call(filename, flags, mode);
}

static unsigned long **aquire_sys_call_table(void)
{
#ifdef HAVE_KSYS_CLOSE
```

```
    unsigned long int offset = PAGE_OFFSET;
    unsigned long **sct;

    while (offset < ULLONG_MAX) {
        sct = (unsigned long **)offset;

        if (sct[__NR_close] == (unsigned long *)ksys_close)
            return sct;

        offset += sizeof(void *);
    }

    return NULL;
#endif

#ifdef HAVE_PARAM
    const char sct_name[15] = "sys_call_table";
    char symbol[40] = { 0 };

    if (sym == 0) {
        pr_alert("For Linux v5.7+, Kprobes is the preferable way to get "
                 "symbol.\n");
        pr_info("If Kprobes is absent, you have to specify the address of "
                "sys_call_table symbol\n");
        pr_info("by /boot/System.map or /proc/kallsyms, which contains all the "
                "symbol addresses, into sym parameter.\n");
        return NULL;
    }
    sprint_symbol(symbol, sym);
    if (!strncmp(sct_name, symbol, sizeof(sct_name) - 1))
        return (unsigned long **)sym;

    return NULL;
#endif

#ifdef HAVE_KPROBES
    unsigned long (*kallsyms_lookup_name)(const char *name);
    struct kprobe kp = {
        .symbol_name = "kallsyms_lookup_name",
    };

    if (register_kprobe(&kp) < 0)
        return NULL;
    kallsyms_lookup_name = (unsigned long (*)(const char *name))kp.addr;
    unregister_kprobe(&kp);
#endif

    return (unsigned long **)kallsyms_lookup_name("sys_call_table");
```

```
}

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 3, 0)
static inline void __write_cr0(unsigned long cr0)
{
    asm volatile("mov %0,%%cr0" : "+r"(cr0) : : "memory");
}
#else
#define __write_cr0 write_cr0
#endif

static void enable_write_protection(void)
{
    unsigned long cr0 = read_cr0();
    set_bit(16, &cr0);
    __write_cr0(cr0);
}

static void disable_write_protection(void)
{
    unsigned long cr0 = read_cr0();
    clear_bit(16, &cr0);
    __write_cr0(cr0);
}

static int __init syscall_start(void)
{
    if (!(sys_call_table = aquire_sys_call_table()))
        return -1;

    disable_write_protection();

    /* Отслеживание исходной функции открытия. */
    original_call = (void *)sys_call_table[__NR_open];

    /* Использование вместо нее собственной функции. */
    sys_call_table[__NR_open] = (unsigned long *)our_sys_open;

    enable_write_protection();

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

    return 0;
}

static void __exit syscall_end(void)
{
    if (!sys_call_table)
```

```
        return;

    /* Восстановление исходного системного вызова. */
    if (sys_call_table[__NR_open] != (unsigned long *)our_sys_open) {
        pr_alert("Somebody else also played with the ");
        pr_alert("open system call\n");
        pr_alert("The system may be left in ");
        pr_alert("an unstable state.\n");
    }

    disable_write_protection();
    sys_call_table[__NR_open] = (unsigned long *)original_call;
    enable_write_protection();

    msleep(2000);
}

module_init(syscall_start);
module_exit(syscall_end);


MODULE_LICENSE("GPL");
```

11. Блокировка процессов и потоков

11.1 Ожидание

Что вы делаете, когда вас просят сделать что-то, чем пока вы заняться не можете? Как обычный человек, которого просит другой такой же человек, вы на это можете сказать лишь: «Я пока занят. Не мешай». Но если вы являетесь ядром, а обратился к вам процесс, то у вас есть другой вариант.

Вы можете поставить этот процесс в режим ожидания (sleep), пока не появится возможность его обслужить. По факту ядро постоянно отправляет процессы в ожидание и пробуждает их. Именно так реализовано одновременное выполнение множества процессов на одном ЦПУ.

И текущий модуль ядра является примером этого. Файл (с именем `/proc/sleep`) одновременно может быть открыт лишь одним процессом. Если он уже открыт, модуль вызывает `wait_event_interruptible`. Самый простой способ сохранять файл открытым – это использовать команду:

```
tail -f
```

Эта функция изменяет статус задачи (задача – это структура данных ядра, содержащая информацию о процессе и системном вызове, в котором он находится, если таковой присутствует) на `TASK_INTERRUPTIBLE`. Это означает, что выполнение задачи будет отложено до момента ее пробуждения, а пока она добавляется в `WaitQ`, то есть очередь задач, ожидающих возможности получить доступ к файлу. Затем эта функция вызывает планировщик для переключения контекста на другой процесс, которому нужен ЦПУ.

Когда процесс закончил работу с файлом, он его закрывает, и вызывается `module_close`. Эта функция пробуждает все процессы в очереди (не существует механизма для пробуждения их по одиночке), после чего делает возврат, и процесс, закрывший файл, может продолжать свое выполнение. Далее в свое время планировщик решает, что этот процесс уже достаточно поработал, и передает управление ЦПУ другому процессу из очереди. Свое выполнение этот процесс начинает с момента, следующего сразу за вызовом `module_interruptible_sleep_on`.

Это означает, что процесс все еще находится в режиме ядра – по имеющейся у него информации, он отправил системный вызов `open()`, который возврат еще не сделал. Процессу не известно, что большую часть времени между моментом отправки этого вызова и его возвратом ЦПУ использовался кем-то еще.

После этого он может установить глобальную переменную, указывающую всем другим процессам, что файл пока открыт, и продолжить выполнение. Когда другие процессы будут получать долю внимания ЦПУ, они будут видеть эту установленную переменную и возвращаться в режим ожидания.

Итак, мы используем `tail -f`, чтобы фоново удерживать файл в открытом состоянии при попытке получить к нему доступ другим процессом (также в фоновом режиме, чтобы не пришлось переключаться на другой VT). Как только первый фоновый процесс завершится командой `kill %1`, пробудится второй, который получит доступ к файлу, а затем также завершится.

При этом `module_close` не единственный, кто имеет право на пробуждение процессов, ожидающих доступа к файлу. Помимо этого, они могут пробуждаться сигналом *Ctrl+C* (`SIGINT`). Причина тому в использованной нами функции `module_interruptible_sleep_on`. Можно было задействовать `module_sleep_on`, но это бы сильно разозлило пользователей, чьи нажатия *Ctrl+C* тогда бы игнорировались. В этом случае нам нужно сразу же возвращать `-EINTR`. Это необходимо, чтобы пользователи могли, например, завершить процесс до получения им доступа к файлу.
Нужно помнить и еще кое-что. Иногда процессы не хотят спать, они хотят незамедлительно получить либо желаемое, либо ответ, что это действие выполнить нельзя. Подобные процессы используют при открытии файла флаг `O_NONBLOCK`.

На это ядро должно возвращать код ошибки `-EAGAIN` от операций, которые в противном случае должны были заблокироваться, к примеру, при открытии файла, как в нашем

примере. Для открытия файла с `O_NONBLOCK` можно использовать программу `cat_nonblock`, расположенную в каталоге `examples/other`.

```
$ sudo insmod sleep.ko
$ cat_nonblock /proc/sleep
Last input:
$ tail -f /proc/sleep &
Last input:
Last input:
Last input:
Last input:
Last input:
Last input:
Last input:
tail: /proc/sleep: file truncated
[1] 6540
$ cat_nonblock /proc/sleep
Open would block
$ kill %1
[1]+  Terminated              tail -f /proc/sleep
$ cat_nonblock /proc/sleep
Last input:
$
```

Код sleep.ko:

```
/*
 * sleep.c - создаем файл /proc, и если его одновременно будут пытаться
 * открыть несколько процессов, все их отправляем в ожидание.
 */

#include <linux/kernel.h> /* Для работы с ядром. */
#include <linux/module.h> /* Для модуля. */
#include <linux/proc_fs.h> /* Необходим для использования procfs */
#include <linux/sched.h> /* Для усыпления процессов и их пробуждения. */
#include <linux/uaccess.h> /* Для get_user и put_user. */
#include <linux/version.h>

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

/* Здесь мы храним последнее полученное сообщение, подтверждая возможность
 * обработки ввода.
 */
```

```c
#define MESSAGE_LENGTH 80
static char message[MESSAGE_LENGTH];

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

/* Так как мы используем структуру файловых операций, то не можем
 * задействовать специальную файловую систему proc и должны
 * использовать стандартную функцию чтения, которой эта функция и является.
 */
static ssize_t module_output(struct file *file, /* см. include/linux/fs.h   */
                             char __user *buf, /* Буфер для данных
                                       (в сегменте пользователя).    */
                             size_t len, /* Длина буфера. */
                             loff_t *offset)
{
    static int finished = 0;
    int i;
    char output_msg[MESSAGE_LENGTH + 30];

    /* Возвращаем 0, обозначая конец файла.
     */
    if (finished) {
        finished = 0;
        return 0;
    }

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

    finished = 1;
    return i; /* Возвращаем количество "считанных" байт. */
}

/* Эта функция получает ввод от пользователя, когда он производит запись
 * в файл /proc.
 */
static ssize_t module_input(struct file *file, /* Сам файл. */
                            const char __user *buf, /* Буфер с вводом. */
                            size_t length, /* Длина буфера. */
                            loff_t *offset) /* Смещение до файла — игнорируется.  */
{
    int i;

    /* Помещение ввода в Message, где позднее его сможет использовать
     * module_output.
     */
```

```
    for (i = 0; i < MESSAGE_LENGTH - 1 && i < length; i++)
        get_user(message[i], buf + i);
    /* Нам нужна стандартная строка, завершающаяся нулем. */
    message[i] = '\0';

    /* Нужно вернуть количество использованных во вводе символов. */
    return i;
}

/* 1, если файл сейчас уже кем-то открыт. */
static atomic_t already_open = ATOMIC_INIT(0);

/* Очередь процессов, ожидающих доступа к файлу. */
static DECLARE_WAIT_QUEUE_HEAD(waitq);

/* Вызывается при открытии файла /proc. */
static int module_open(struct inode *inode, struct file *file)
{
    /* Если флаги при открытии файла содержат O_NONBLOCK, значит процесс
     * не хочет ждать доступности этого файла. В таком случае, если файл
     * уже открыт, нужно будет не блокировать процесс, который
     * предпочитает оставаться открытым, а вернуть -EAGAIN, сообщив ему,
     * что попытку нужно повторить позже.
     */
    if ((file->f_flags & O_NONBLOCK) && atomic_read(&already_open))
        return -EAGAIN;

    /* Это подходящее место для try_module_get(THIS_MODULE), так как,
     * если процесс находится в цикле в модуле ядра, то этот модуль
     * извлекать нельзя.
     */
    try_module_get(THIS_MODULE);

    while (atomic_cmpxchg(&already_open, 0, 1)) {
        int i, is_sig = 0;

        /* Эта функция отправляет текущий процесс, включая любые системные
         * вызовы, например наши, в ожидание. Выполнение продолжится сразу
         * после вызова этой функции либо при вызове
         * wake_up(&waitq) (это делает только module_close при закрытии
         * файла), либо при отправке процессу сигнала вроде Ctrl+C.
         */
        wait_event_interruptible(waitq, !atomic_read(&already_open));

        /* Если пробуждение произошло из-за получения сигнала, который не
         * блокируется, вернуть -EINTR (провал системного вызова). Это
         * позволяет завершать или останавливать процессы.
         */
```

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

        if (is_sig) {
            /* Важно поместить module_put(THIS_MODULE) сюда, так как
             * для процессов, где окажется прервана операция open(),
             * соответствующей операции close() не будет. Если не
             * декрементировать счетчик использования здесь, у нас
             * останется в нем положительный счет, который мы никак уже
             * не приведем к нулю. В итоге у нас получится бессмертный
             * модуль, для извлечения которого потребуется перезагрузка.
             */
            module_put(THIS_MODULE);
            return -EINTR;
        }
    }

    return 0; /* Разрешение доступа. */
}

/* Вызывается при закрытии файла /proc. */
static int module_close(struct inode *inode, struct file *file)
{
    /* Устанавливаем already_open на нуль, чтобы один из процессов в waitq
     * мог установить already_open обратно на один и открыть файл. В итоге
     * остальные процессы при вызове будут видеть, что already_open
     * равен одному, в связи с чем возвращаться в ожидание.
     */
    atomic_set(&already_open, 0);

    /* Пробуждение всех процессов в waitq, чтобы очередной ожидающий мог
     * получить доступ к файлу.
     */
    wake_up(&waitq);

    module_put(THIS_MODULE);

    return 0; /* Успех. */
}

/* Структуры для регистрации в качестве файла /proc с указателями на все
 * связанные функции.
 */

/* Файловые операции нашего файла /proc. Здесь размещаются указатели на
 * все функции, вызываемые, когда кто-то пытается произвести действия с
 * файлом. NULL означает, что мы не хотим выполнять какое-то действие.
 */
```

```c
#ifdef HAVE_PROC_OPS
static const struct proc_ops file_ops_4_our_proc_file = {
    .proc_read = module_output, /* "Считывание" из файла. */
    .proc_write = module_input, /* "Запись" в файл. */
    .proc_open = module_open, /* Вызывается при открытии файла /proc */
    .proc_release = module_close, /* Вызывается при его закрытии. */
};
#else
static const struct file_operations file_ops_4_our_proc_file = {
    .read = module_output,
    .write = module_input,
    .open = module_open,
    .release = module_close,
};
#endif

/* Инициализация модуля – регистрация файла /proc. */
static int __init sleep_init(void)
{
    our_proc_file =
        proc_create(PROC_ENTRY_FILENAME, 0644, NULL, &file_ops_4_our_proc_file);
    if (our_proc_file == NULL) {
        remove_proc_entry(PROC_ENTRY_FILENAME, NULL);
        pr_debug("Error: Could not initialize /proc/%s\n", PROC_ENTRY_FILENAME);
        return -ENOMEM;
    }
    proc_set_size(our_proc_file, 80);
    proc_set_user(our_proc_file, GLOBAL_ROOT_UID, GLOBAL_ROOT_GID);

    pr_info("/proc/%s created\n", PROC_ENTRY_FILENAME);

    return 0;
}

/* Очистка – снятие регистрации файла из /proc.  Это может быть опасно,
 * если в waitq еще есть ожидающие процессы, потому что они находятся
 * внутри функции open(), которая будет выгружена. В 10 главе я объясняю,
 * как в подобном случае избежать извлечения модуля.
 */
static void __exit sleep_exit(void)
{
    remove_proc_entry(PROC_ENTRY_FILENAME, NULL);
    pr_debug("/proc/%s removed\n", PROC_ENTRY_FILENAME);
}

module_init(sleep_init);
module_exit(sleep_exit);
```

Код cat_nonblock.c:

```
MODULE_LICENSE("GPL");
/*
 *  cat_nonblock.c - открывает файл и отображает содержимое, но в случае
 *  необходимости ожидания ввода выходит.
 */
#include <errno.h> /* Для errno. */
#include <fcntl.h> /* Для открытия. */
#include <stdio.h> /* Стандартный ввод-вывод. */
#include <stdlib.h> /* Для выхода. */
#include <unistd.h> /* Для считывания.*/

#define MAX_BYTES 1024 * 4

int main(int argc, char *argv[])
{
    int fd; /* Дескриптор считываемого файла. */
    size_t bytes; /* Количество считываемых байт. */
    char buffer[MAX_BYTES]; /* Буфер для этих байт. */

    /* Использование. */
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        puts("Reads the content of a file, but doesn't wait for input");
        exit(-1);
    }

    /* Открытие файла для считывания в неблокирующемся режиме. */
    fd = open(argv[1], O_RDONLY | O_NONBLOCK);

    /* Если открытие провалилось. */
    if (fd == -1) {
        puts(errno == EAGAIN ? "Open would block" : "Open failed");
        exit(-1);
    }

    /* Считывание файла и вывод его содержимого. */
    do {
        /* Считывание символов из файла. */
        bytes = read(fd, buffer, MAX_BYTES);

        /* В случае ошибки сообщить о ней и завершиться. */
        if (bytes == -1) {
            if (errno == EAGAIN)
                puts("Normally I'd block, but you told me not to");
            else
```

```
                puts("Another read error");
            exit(-1);
        }

        /* Вывод символов. */
        if (bytes > 0) {
            for (int i = 0; i < bytes; i++)
                putchar(buffer[i]);
        }

        /* Пока нет ошибок, и файл не закончился. */
    } while (bytes > 0);

    return 0;
}
```

11.2 Завершение потоков

Иногда в модуле, имеющем несколько потоков, одно действие должно совершиться перед другим. И вместо использования команд `/bin/sleep` ядро реализует это другим способом, поддерживающим таймауты или прерывания.В примере ниже стартуют два потока, но один должен сработать раньше.

Код completion.c:

```
/*
 * completions.c
 */
#include <linux/completion.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/kthread.h>
#include <linux/module.h>

static struct {
    struct completion crank_comp;
    struct completion flywheel_comp;
} machine;

static int machine_crank_thread(void *arg)
{
    pr_info("Turn the crank\n");

    complete_all(&machine.crank_comp);
```

```
    complete_and_exit(&machine.crank_comp, 0);
}

static int machine_flywheel_spinup_thread(void *arg)
{
    wait_for_completion(&machine.crank_comp);

    pr_info("Flywheel spins up\n");

    complete_all(&machine.flywheel_comp);
    complete_and_exit(&machine.flywheel_comp, 0);
}

static int completions_init(void)
{
    struct task_struct *crank_thread;
    struct task_struct *flywheel_thread;

    pr_info("completions example\n");

    init_completion(&machine.crank_comp);
    init_completion(&machine.flywheel_comp);

    crank_thread = kthread_create(machine_crank_thread, NULL, "KThread Crank");
    if (IS_ERR(crank_thread))
        goto ERROR_THREAD_1;

    flywheel_thread = kthread_create(machine_flywheel_spinup_thread, NULL,
                                     "KThread Flywheel");
    if (IS_ERR(flywheel_thread))
        goto ERROR_THREAD_2;

    wake_up_process(flywheel_thread);
    wake_up_process(crank_thread);

    return 0;

ERROR_THREAD_2:
    kthread_stop(crank_thread);
ERROR_THREAD_1:

    return -1;
}

static void completions_exit(void)
{
    wait_for_completion(&machine.crank_comp);
    wait_for_completion(&machine.flywheel_comp);
```

```
    pr_info("completions exit\n");
}

module_init(completions_init);
module_exit(completions_exit);

MODULE_DESCRIPTION("Completions example");
MODULE_LICENSE("GPL");
```

Структура `machine` хранит состояния завершения для этих двух потоков. В точке выхода каждого из них обновляется соответствующее состояние.

При этом для потока `flywheel` используется `wait_for_completion`, чтобы он не запустился преждевременно.

Так что, хоть `flywheel_thread` и стартует первым, загрузив модуль и выполнив `dmesg`, вы должны заметить, что сначала всегда происходит поворот рычага (`crank`), потому что поток маховика (`flywheel`) ожидает его завершения.

У функции `wait_for_completion` есть и другие вариации, которые включают таймауты и прерывания, но этого базового механизма вполне достаточно для множества типичных ситуаций без добавления излишней сложности.

12. Избегание коллизий и взаимных блокировок

Если процессы, выполняющиеся на разных ядрах или в разных потоках, попытаются обратиться к одной и той же области памяти, то вполне могут случиться странности, либо система просто заблокируется.

Для избежания этого в ядре существуют специальные функции взаимного исключения (мьютексы).

Они показывают, «занят» или «свободен» в данный момент фрагмент кода, исключая тем самым одновременные попытки его выполнения.

12.1 Мьютексы

Используются мьютексы ядра аналогично тому, как они развертываются в пользовательской среде.

И в большинстве случаев для избежания коллизий этого вполне может оказаться достаточно.

Код example_mutex.c:

```c
/*
 * example_mutex.c
 */
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/mutex.h>

static DEFINE_MUTEX(mymutex);

static int example_mutex_init(void)
{
    int ret;

    pr_info("example_mutex init\n");

    ret = mutex_trylock(&mymutex);
    if (ret != 0) {
        pr_info("mutex is locked\n");

        if (mutex_is_locked(&mymutex) == 0)
            pr_info("The mutex failed to lock!\n");

        mutex_unlock(&mymutex);
        pr_info("mutex is unlocked\n");
    } else
        pr_info("Failed to lock\n");

    return 0;
}

static void example_mutex_exit(void)
{
    pr_info("example_mutex exit\n");
}

module_init(example_mutex_init);
module_exit(example_mutex_exit);

MODULE_DESCRIPTION("Mutex example");
MODULE_LICENSE("GPL");
```

12.2 Спин-блокировки

Спин-блокировки, или спинлоки, блокируют ЦПУ, на котором выполняется код, занимая 100% его ресурсов. В связи с этим механизм спинлоков желательно использовать только для кода, на выполнение которого требуется не более нескольких миллисекунд, чтобы с позиции пользователя не вызвать заметного замедления работы.

Примером в данном случае является ситуация *irq safe*, когда прерывания, происходящие во время блокировки, не забываются, а повторно активируются при ее снятии, используя переменную `flags` для сохранения своего состояния.

Код example_spinlock.c:

```
/*
 * example_spinlock.c
 */
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/spinlock.h>

static DEFINE_SPINLOCK(sl_static);
static spinlock_t sl_dynamic;

static void example_spinlock_static(void)
{
    unsigned long flags;

    spin_lock_irqsave(&sl_static, flags);
    pr_info("Locked static spinlock\n");

    /* Безопасное выполнение задачи. Поскольку задействуется 100% ЦПУ,
     * выполнение кода должно занимать не более нескольких миллисекунд.
     */

    spin_unlock_irqrestore(&sl_static, flags);
    pr_info("Unlocked static spinlock\n");
}

static void example_spinlock_dynamic(void)
{
    unsigned long flags;

    spin_lock_init(&sl_dynamic);
    spin_lock_irqsave(&sl_dynamic, flags);
```

```
    pr_info("Locked dynamic spinlock\n");

    /* Безопасное выполнение задачи. Поскольку задействуется 100% ЦПУ,
     * выполнение кода должно занимать не более нескольких миллисекунд.
     */

    spin_unlock_irqrestore(&sl_dynamic, flags);
    pr_info("Unlocked dynamic spinlock\n");
}

static int example_spinlock_init(void)
{
    pr_info("example spinlock started\n");

    example_spinlock_static();
    example_spinlock_dynamic();

    return 0;
}

static void example_spinlock_exit(void)
{
    pr_info("example spinlock exit\n");
}

module_init(example_spinlock_init);
module_exit(example_spinlock_exit);

MODULE_DESCRIPTION("Spinlock example");
MODULE_LICENSE("GPL");
```

12.3 Блокировки для чтения и записи

Блокировки для выполнения чтения и записи – это специализированные спинлоки, позволяющие эксклюзивно считывать или производить запись.

Подобно предыдущему примеру, код ниже показывает ситуацию *irq safe*, когда в случае активации аппаратными прерываниями других функций, которое также могут выполнять нужные вам чтение/запись, эти функции не нарушат текущую логику выполнения.

Как и прежде, будет правильным решением, устанавливать подобную блокировку для максимально коротких задач, чтобы они не подвешивали систему и не вызывали недовольство пользователей относительно тирании вашего модуля.

Код example_rwlock.c:

```c
/*
 * example_rwlock.c
 */
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/module.h>

static DEFINE_RWLOCK(myrwlock);

static void example_read_lock(void)
{
    unsigned long flags;

    read_lock_irqsave(&myrwlock, flags);
    pr_info("Read Locked\n");

   /* Считывание. */

    read_unlock_irqrestore(&myrwlock, flags);
    pr_info("Read Unlocked\n");
}

static void example_write_lock(void)
{
    unsigned long flags;

    write_lock_irqsave(&myrwlock, flags);
    pr_info("Write Locked\n");

    /* Запись. */

    write_unlock_irqrestore(&myrwlock, flags);
    pr_info("Write Unlocked\n");
}

static int example_rwlock_init(void)
{
    pr_info("example_rwlock started\n");

    example_read_lock();
    example_write_lock();

    return 0;
}
```

```
static void example_rwlock_exit(void)
{
    pr_info("example_rwlock exit\n");
}

module_init(example_rwlock_init);
module_exit(example_rwlock_exit);

MODULE_DESCRIPTION("Read/Write locks example");
MODULE_LICENSE("GPL");
```

Конечно же, если вы уверены, что аппаратные прерывания не активируют никакие функции, которые могли бы нарушить логику, то можете использовать более простые `read_lock(&myrwlock)` и `read_unlock(&myrwlock)` либо соответствующие функции записи.

12.4 Атомарные операции

Если вы выполняете простую арифметику: сложение, вычитание или побитовые операции, тогда многоядерный и гиперпоточный мир может предложить еще один способ, как не позволить другим компонентам системы вмешаться в ваше действо. С помощью атомарных операций вы можете обеспечить, чтобы ваше сложение, вычитание или инвертирование битов произошли успешно и не были перезаписаны какими-либо сторонними процессами. Вот пример:

Код example_atomic.c:

```
/*
 * example_atomic.c
 */
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/module.h>

#define BYTE_TO_BINARY_PATTERN "%c%c%c%c%c%c%c%c"
#define BYTE_TO_BINARY(byte)                                    \
    ((byte & 0x80) ? '1' : '0'), ((byte & 0x40) ? '1' : '0'),   \
        ((byte & 0x20) ? '1' : '0'), ((byte & 0x10) ? '1' : '0'),   \
        ((byte & 0x08) ? '1' : '0'), ((byte & 0x04) ? '1' : '0'),   \
        ((byte & 0x02) ? '1' : '0'), ((byte & 0x01) ? '1' : '0')

static void atomic_add_subtract(void)
{
    atomic_t debbie;
```

```
    atomic_t chris = ATOMIC_INIT(50);

    atomic_set(&debbie, 45);

    /* Вычитание единицы. */
    atomic_dec(&debbie);

    atomic_add(7, &debbie);

    /* Прибавление единицы. */
    atomic_inc(&debbie);

    pr_info("chris: %d, debbie: %d\n", atomic_read(&chris),
            atomic_read(&debbie));
}

static void atomic_bitwise(void)
{
    unsigned long word = 0;

    pr_info("Bits 0: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));
    set_bit(3, &word);
    set_bit(5, &word);
    pr_info("Bits 1: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));
    clear_bit(5, &word);
    pr_info("Bits 2: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));
    change_bit(3, &word);

    pr_info("Bits 3: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));
    if (test_and_set_bit(3, &word))
        pr_info("wrong\n");
    pr_info("Bits 4: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));

    word = 255;
    pr_info("Bits 5: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));
}

static int example_atomic_init(void)
{
    pr_info("example_atomic started\n");

    atomic_add_subtract();
    atomic_bitwise();

    return 0;
}

static void example_atomic_exit(void)
```

```
{
    pr_info("example_atomic exit\n");
}

module_init(example_atomic_init);
module_exit(example_atomic_exit);

MODULE_DESCRIPTION("Atomic operations example");
MODULE_LICENSE("GPL");
```

До того, как в стандарте C11 появились встроенные атомарные типы, ядро уже предоставляло небольшой их набор, которым можно было воспользоваться с помощью хитрого архитектурно-зависимого кода.

Реализация же атомарных типов в C11 позволяет ядру отказаться от этих специфичных команд, сделав его код более внятным для людей, которые данный стандарт понимают. Но есть здесь и кое-какие проблемы, например модель памяти ядра не соответствует модели, формируемой атомарными операциями в C11. Подробнее эта тема раскрыта в следующих ресурсах:

- Kernel documentation of atomic types
- Time to move to C11 atomics?
- Atomic usage patterns in the kernel

13. Замена макроса Print

13.1 Замена

В разделе 2 я говорил, что программировать модули ядра через X Window System не желательно. Это верно относительно именно разработки модулей, но при фактическом использовании нам нужна возможность отправлять сообщения на любой tty, с которого поступила команда на загрузку модуля.

Аббревиатура tty означает телетайп, устройство, которое в своем изначальном виде представляло совмещенную с принтером клавиатуру, используемую для взаимодействия с системой Unix.

В современном же представлении телетайп является абстракцией текстового потока, используемой программой Unix, будь то физический терминал, xterm на X-сервере, сетевое подключение через ssh или нечто аналогичное.

Реализуется это с помощью `current`, указателя на выполняемую в данный момент задачу, позволяющего получить tty-структуру этой задачи. Затем эта структура просматривается в поиске указателя на строковую функцию `write`, которая используется для записи строки в данный tty.

Код print_string.c:

```
/*
 * print_string.c - отправляет вывод на tty, с которого мы работаем,
 * будь то через X11, telnet и т.д. Для этого мы выводим строку на
 * tty, связанный с текущей задачей.
 */
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h> /* Для current. */
#include <linux/tty.h> /* Для объявлений tty. */

static void print_string(char *str)
{
    /* tty для текущей задачи. */
    struct tty_struct *my_tty = get_current_tty();

    /* Если my_tty равен NULL, значит у текущей задачи нет tty, куда
     * можно было бы произвести вывод (например, если это демон). В таком
     * случае ничего не поделаешь.
     */
    if (my_tty) {
        const struct tty_operations *ttyops = my_tty->driver->ops;
        /* my_tty->driver - это структура, где расположены функции tty,
         * одна из которых (write) используется для записи строк в tty.
         * С помощью нее можно извлекать строки из сегментов пространства памяти ядра
или
         * пользователя.
         *
         * Первый параметр этой функции устанавливает tty, куда нужно
         * производить запись, потому что одна и та же функция служит
         * для записи во все tty определенного типа.
         * Второй параметр - это указатель на строку.
         * Третий параметр устанавливает длину строки.
         *
         * Как вы увидите ниже, иногда необходимо использовать функционал
         * препроцессора, чтобы получить код, работающий для различных
         * версий ядра. Реализованный нами здесь наивный подход плохо
         * масштабируется. Правильный способ решения этой проблемы описан
         * в разделе 2 документации: linux/Documentation/SubmittingPatches
         */
```

```
        (ttyops->write)(my_tty, /* Сам tty. */
                        str, /* Строка. */
                        strlen(str)); /* Длина. */

        /* Изначально телетайпы были аппаратными и, как правило, строго
         * следовали стандарту ASCII. В ASCII для перехода на новую строку
         * необходимо два символа, возврат каретки и перевод строки. В
         * Unix перевод строки ASCII задействуется для того и другого,
         * поэтому нельзя просто использовать \n, так как возврата
         * каретки не произойдет, и следующая строка начнется в столбце,
         * идущим сразу за переводом строки.
         *
         * Именно поэтому в Unix и MS Windows текстовые файлы отличаются.
         * В CP/M и ее производных вроде MS-DOS и MS Windows текст строго
         * подчиняется стандарту ASCII, в связи с чем для перехода на
         * новую строку требуется и LF, и CR.
         */
        (ttyops->write)(my_tty, "\015\012", 2);
    }
}

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

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

module_init(print_string_init);
module_exit(print_string_exit);


MODULE_LICENSE("GPL");
```

13.2 Мигание светодиодами клавиатуры

В определенных условиях вы можете предпочесть более простой и непосредственный способ связи с внешним миром. Решением в таком случае может стать мигание светодиодами клавиатуры. Это прямой способ привлечь внимание или продемонстрировать некое состояние. Светодиоды есть у любой клавиатуры, они всегда на виду, не требуют настройки и очень просты в использовании, если сравнивать с записью в tty или файл.

В v4.14 и v.4.15 в API таймера произошел ряд изменений, нацеленных на повышение безопасности памяти. Переполнение буфера в области структуры `timer_list` может привести к перезаписи полей `function` и `data`, предоставив атакующему возможность с помощью возвратно-объектного программирование вызывать произвольные функции в ядре.

Кроме того, прототип функции обратного вызова, содержащий аргумент `unsigned long`, полностью исключит возможность проверки типов. Плюс такой прототип может помешать защитить косвенные переходы и вызовы (forward-edge) с помощью сохранения целостности потока управления (CFI).

Поэтому лучше использовать уникальный прототип, чтобы отделиться от кластера, который получает аргумент `unsigned long`. В обратный вызов таймера необходимо передавать не аргумент `unsigned long`, а указатель на структуру `timer_list`.

Тогда он объединит всю необходимую ему информацию, включая структуру `timer_list`, в более обширную структуру и сможет использовать вместо значения `unsigned long` макрос `container_of`.

Более развернуто эта тема описана в статье Improving the kernel timers API.

До Linux v4.14 инициализация таймеров производилась с помощью `setup_timer`, а структура `timer_list` выглядела так:

```
struct timer_list {
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
    u32 flags;
    /* ... */
};

void setup_timer(struct timer_list *timer, void (*callback)(unsigned long),
                 unsigned long data);
```

В Linux v4.14 появилась `timer_setup`, и ядро постепенно перестроилось с `setup_timer` на `timer_setup`. Одна из причин изменения API заключалась в потребности сосуществования с интерфейсом старых версий. Более того, по началу `timer_setup` реализовывалась через `setup_timer`:

```
void timer_setup(struct timer_list *timer,
                 void (*callback)(struct timer_list *), unsigned int flags);
```

Позднее в v4.15 `setup_timer` удалили, что также отразилось на облике структуры `timer_list`:

```
struct timer_list {
    unsigned long expires;
    void (*function)(struct timer_list *);
    u32 flags;
    /* ... */
};
```

Приведенный ниже код демонстрирует минимальный модуль ядра, который после загрузки начинает мигать светодиодами до тех пор, пока не будет выгружен.

Код kbleds.c:

```
/*
 * kbleds.c - мигает светодиодами клавиатуры, пока не будет выгружен.
 */

#include <linux/init.h>
#include <linux/kd.h> /* Для KDSETLED. */
#include <linux/module.h>
#include <linux/tty.h> /* Для tty_struct. */
#include <linux/vt.h> /* Для MAX_NR_CONSOLES. */
#include <linux/vt_kern.h> /* Для fg_console. */
#include <linux/console_struct.h> /* Для vc_cons. */

MODULE_DESCRIPTION("Example module illustrating the use of Keyboard LEDs.");

static struct timer_list my_timer;
static struct tty_driver *my_driver;
static unsigned long kbledstatus = 0;

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

/* Функция my_timer_func периодически мигает светодиодами,
 * вызывая для драйвера клавиатуры команду управления вводом-выводом
 * KDSETLED. Дополнительную информацию по командам ввода-вывода
 * смотрите в функции vt_ioctl() файла drivers/tty/vt/vt_ioctl.c.
 *
 * Аргумент KDSETLED попеременно устанавливается то на 7 (что приводит к
 * активации режима LED_SHOW_IOCTL и загоранию всех светодиодов), то на
 * 0xFF (любое значение выше 7 переключает режим обратно на
 * LED_SHOW_FLAGS, в результате чего светодиоды отображают фактический
```

```
 * статус клавиатуры). Подробности смотрите в функции setledstate() файла
 * drivers/tty/vt/keyboard.c.
  */
static void my_timer_func(struct timer_list *unused)
{
    struct tty_struct *t = vc_cons[fg_console].d->port.tty;

    if (kbledstatus == ALL_LEDS_ON)
        kbledstatus = RESTORE_LEDS;
    else
        kbledstatus = ALL_LEDS_ON;

    (my_driver->ops->ioctl)(t, KDSETLED, kbledstatus);

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

static int __init kbleds_init(void)
{
    int i;

    pr_info("kbleds: loading\n");
    pr_info("kbleds: fgconsole is %x\n", fg_console);
    for (i = 0; i < MAX_NR_CONSOLES; i++) {
        if (!vc_cons[i].d)
            break;
        pr_info("poet_atkm: console[%i/%i] #%i, tty %p\n", i, MAX_NR_CONSOLES,
                vc_cons[i].d->vc_num, (void *)vc_cons[i].d->port.tty);
    }
    pr_info("kbleds: finished scanning consoles\n");

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

    /* Первая настройка таймера мигания светодиодов. */
    timer_setup(&my_timer, my_timer_func, 0);
    my_timer.expires = jiffies + BLINK_DELAY;
    add_timer(&my_timer);

    return 0;
}

static void __exit kbleds_cleanup(void)
{
    pr_info("kbleds: unloading...\n");
    del_timer(&my_timer);
    (my_driver->ops->ioctl)(vc_cons[fg_console].d->port.tty, KDSETLED,
```

```
                            RESTORE_LEDS);
}

module_init(kbleds_init);
module_exit(kbleds_cleanup);


MODULE_LICENSE("GPL");
```

Если ни один из приведенных в этой главе примеров не подходит под ваши отладочные нужды, то наверняка есть другие решения. Не задумывались, для чего может быть полезна `CONFIG_LL_DEBUG` из `menu menuconfig`?

В случае ее активации вы получаете низкоуровневый доступ к последовательному порту. И хотя это может не показаться особо полезным, такой прием позволяет пропатчить `kernel/printk.c` или любой другой важный системный вызов для печати символов ASCII, делая возможным отслеживание практически всех действий кода на последовательном порту.

Если вы займетесь портированием ядра на новую, ранее неподдерживаемую архитектуру, то реализация этого решения должна идти одной из первых. Также можно рассмотреть вариант логирования через netconsole.

Несмотря на множество рассмотренных здесь отладочных приемов, кое-что нужно иметь ввиду. Отладка практически всегда оказывается интрузивной процедурой. Добавление отладочного кода может привести к тому, что ошибка, на первый взгляд, исчезнет. Поэтому такой код нужно минимизировать и следить, чтобы он не попал в продакшн-код.

14. Планирование задач

Есть два основных способа выполнения задач: тасклеты и очереди заданий. Тасклеты – это быстрый и простой способ планирования выполнения одной функции, например, при ее активации прерыванием.

А вот очереди заданий хоть и более сложны, но зато лучше подходят для выполнения последовательностей задач.

14.1 Тасклеты

Ниже показан пример модуля тасклета. Функция `tasklet_fn` выполняется несколько секунд. При этом выполнение функции `example_tasklet_init` может продолжаться до точки выхода, что будет зависеть от того, была ли она прервана `softirq`.

Код example_tasklet.c:

```
/*
 * example_tasklet.c
 */
#include <linux/delay.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/module.h>

/* Макрос DECLARE_TASKLET_OLD присутствует для совместимости.
 * См. https://lwn.net/Articles/830964/.
 */
#ifndef DECLARE_TASKLET_OLD
#define DECLARE_TASKLET_OLD(arg1, arg2) DECLARE_TASKLET(arg1, arg2, 0L)
#endif

static void tasklet_fn(unsigned long data)
{
    pr_info("Example tasklet starts\n");
    mdelay(5000);
    pr_info("Example tasklet ends\n");
}

static DECLARE_TASKLET_OLD(mytask, tasklet_fn);

static int example_tasklet_init(void)
{
    pr_info("tasklet example init\n");
    tasklet_schedule(&mytask);
    mdelay(200);
    pr_info("Example tasklet init continues...\n");
    return 0;
}

static void example_tasklet_exit(void)
{
    pr_info("tasklet example exit\n");
    tasklet_kill(&mytask);
}

module_init(example_tasklet_init);
module_exit(example_tasklet_exit);

MODULE_DESCRIPTION("Tasklet example");
MODULE_LICENSE("GPL");
```

После загрузки этого примера `dmesg` должна отобразить следующее:

```
tasklet example init
Example tasklet starts
Example tasklet init continues...
Example tasklet ends
```

И хотя использовать тасклеты легко, они имеют несколько недостатков, и в среде разработчиков обсуждается их возможное исключение из ядра. Обратный вызов тасклета выполняется в атомарном контексте внутри программного прерывания, то есть он не может входить в режим ожидания или получать доступ к данным пользовательского пространства, в результате чего в обработчике тасклетов не получится выполнить всю работу. Кроме того, ядро разрешает одновременно выполнять только один экземпляр любого конкретного тасклета. При этом несколько разных могут выполняться параллельно.

В последних версиях ядра появилась возможность заменить тасклеты очередями заданий, таймерами или прерываниями, выносимыми в отдельные потоки (threaded interrupts). Пока удаление тасклетов продолжает оставаться долгосрочной целью, в своем текущем виде ядро содержит более сотни случаев их использования. Сейчас разработчики продолжают вносить изменения в API, и для совместимости существует макрос `DECLARE_TASKLET_OLD`.

Подробнее читайте на странице https://lwn.net/Articles/830964/.

14.2 Очереди заданий

Добавлять задачи в планировщик можно через очередь заданий. Для выполнения прописанных в этой очереди задач ядро использует Completely Fair Scheduler (CFS).

Код sched.c:

```
/*
 * sched.c
 */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/workqueue.h>

static struct workqueue_struct *queue = NULL;
static struct work_struct work;

static void work_handler(struct work_struct *data)
```

```
{
    pr_info("work handler function.\n");
}

static int __init sched_init(void)
{
    queue = alloc_workqueue("HELLOWORLD", WQ_UNBOUND, 1);
    INIT_WORK(&work, work_handler);
    schedule_work(&work);
    return 0;
}

static void __exit sched_exit(void)
{
    destroy_workqueue(queue);
}

module_init(sched_init);
module_exit(sched_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Workqueue example");
```

15. Обработка прерываний

15.1 Обработчики прерываний

Во всех главах (за исключением предыдущей), мы реализовывали в ядре лишь ответы на запросы процессов, для чего-либо работали с особым файлом, либо отправляли `ioctl()` или системный вызов. Но работа ядра состоит не только в реагировании на запросы процессов. Ещё одной его немаловажной ответственностью является взаимодействие с подключённым к машине оборудованием.

Существует два типа взаимодействий между ЦПУ и остальным оборудованием компьютера. Первый – это когда ЦПУ отдаёт ему распоряжения. Распоряжение подразумевает, что оборудование должно сообщить что-либо процессору. Второй, называемый прерываниями, уже гораздо сложнее в реализации, поскольку обрабатывается, когда это нужно оборудованию, а не процессору. Как правило, аппаратные средства оснащены очень небольшим объёмом оперативной памяти, и если своевременно не считать предоставляемую ими информацию, она будет потеряна.

В Linux — аппаратные прерывания называются IRQ (Interrupt ReQuests). Существует два типа IRQ, короткие и длинные. Короткие прерывания, как и предполагает их имя, должны выполняться в краткий промежуток времени, во время которого остальная часть машины

будет заблокирована, и обработка никаких других прерываний производиться не будет. Длительные же IRQ выполняются продолжительно и не препятствуют выполнению других прерываний (за исключением IRQ от одного и того же устройства). По возможности желательно объявлять обработчики прерываний длительными.

Когда ЦПУ получает прерывание, он прекращает все свои текущие действия (если только не обрабатывает более важное прерывание; в таком случае сначала он заканчивает его), сохраняет определённые параметры в стеке и вызывает обработчик прерываний. Это означает, что определённые действия в самом обработчике недопустимы, поскольку система находится в неизвестном состоянии. Для решения этой проблемы ядро разделяет обработку прерываний на две части. Первая выполняется сразу же и маскирует линию прерываний.

Аппаратные прерывания должны обрабатываться быстро, и именно поэтому нам нужна вторая часть, выполняющая всю тяжёлую работу, отделённую от обработчика. По историческим причинам BH (аббревиатура для Нижних половин) статистически ведёт учёт этих отделённых функций. Начиная с Linux 2.3, на смену BH пришёл механизм Softirq и его более высокоуровневая абстракция Tasklet.

Реализуется этот механизм через вызов `request_irq()`, который при получении прерывания активизирует его обработчик.

На практике же обработка IRQ может представлять сложности. Зачастую аппаратные устройства реализуют в себе цепочку из двух контроллеров прерываний, чтобы всё IRQ, поступающие от контроллера B, каскадировались в определённое IRQ от контроллера A. Естественно, для этого ядру необходимо разобраться, какое в действительности это было прерывание, что накладывает дополнительную нагрузку. В других архитектурах предлагается особый вид менее нагружающих систему прерываний, называемых «fast IRQ», или FIQ.

Для их использования обработчики должны быть написаны на ассемблере, в связи с чем ядру они уже не подходят. Можно сделать так, чтобы эти обработчики работали аналогично другим, но тогда они утратят своё преимущество в скорости. Ядра с поддержкой SMP, работающие в системах с несколькими процессорами, должны решать множество и других проблем. Недостаточно просто знать о том, что произошло определённое прерывание, также важно понимать, для какого (или каких) ЦПУ оно предназначено.

Тем, кого интересуют дополнительные подробности, рекомендую обратиться к документации по “APIC” (Advanced Programmable Interrupt Controller — улучшенный программируемый обработчик прерываний).

Функция `request_irq` получает номер IRQ, имя функции, флаги, имя для `/proc/interrupts` и параметр, передаваемый в обработчик прерываний. Как правило, доступно определённое число IRQ, какое именно – зависит от оборудования. В качестве

флагов могут использоваться `SA_SHIRQ`, указывающий, что вы хотите поделиться этим IRQ с остальными обработчиками прерываний (обычно ввиду того, что ряд устройств сидят на одном IRQ) и `SA_INTERRUPT`, обозначающий быстрое прерывание. Эта функция сработает успешно, только если для данного прерывания ещё не установлен обработчик, или если вы также хотите им поделиться.

15.2 Обнаружение нажатий клавиш

Многие популярные одноплатные компьютеры, такие как Raspberry Pi или Beegleboards, оборудованы множеством контактов ввода-вывода. Подключение к этим выводам кнопок и настройка срабатывания их нажатий является классическим случаем, в котором могут потребоваться прерывания. Поэтому вместо того, чтобы тратить время процессора и энергию на опрос об изменении входного состояния, лучше настроить вход на активацию ЦПУ для последующего выполнения определённой функции обработки.

Вот пример, в котором кнопки подключены к выводам 17 и 18, а светодиод к выводу 4. При желании можете изменить номера выводов на своё усмотрение.

Код intrpt.c:

```
/*
 * intrpt.c – Обработка ввода-вывода с помощью прерываний.
 *
 * За основу взят пример RPi Стефана Вендлера (devnull@kaltpost.de)
 * из репозитория https://github.com/wendlers/rpi-kmod-samples
 *
 * При нажатии одной кнопки светодиод загорается, а при нажатии другой
 * гаснет.
 */

#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/module.h>

static int button_irqs[] = { -1, -1 };

/* Определение вводов-выводов для светодиодов.
 * Номера выводов можете изменить.
 */
static struct gpio leds[] = { { 4, GPIOF_OUT_INIT_LOW, "LED 1" } };

/* Определение вводов-выводов для BUTTONS.
 * Номера вводов-выводов можете изменить.
 */
```

```
static struct gpio buttons[] = { { 17, GPIOF_IN, "LED 1 ON BUTTON" },
                                 { 18, GPIOF_IN, "LED 1 OFF BUTTON" } };

/* Функция обработки прерываний, активируемая нажатием кнопки. */
static irqreturn_t button_isr(int irq, void *data)
{
    /* Первая кнопка. */
    if (irq == button_irqs[0] && !gpio_get_value(leds[0].gpio))
        gpio_set_value(leds[0].gpio, 1);
    /* Вторая кнопка. */
    else if (irq == button_irqs[1] && gpio_get_value(leds[0].gpio))
        gpio_set_value(leds[0].gpio, 0);

    return IRQ_HANDLED;
}

static int __init intrpt_init(void)
{
    int ret = 0;

    pr_info("%s\n", __func__);

    /* Регистрация вводов-выводов светодиодов. */
    ret = gpio_request_array(leds, ARRAY_SIZE(leds));

    if (ret) {
        pr_err("Unable to request GPIOs for LEDs: %d\n", ret);
        return ret;
    }

    /* Регистрация вводов-выводов для BUTTON. */
    ret = gpio_request_array(buttons, ARRAY_SIZE(buttons));

    if (ret) {
        pr_err("Unable to request GPIOs for BUTTONs: %d\n", ret);
        goto fail1;
    }

    pr_info("Current button1 value: %d\n", gpio_get_value(buttons[0].gpio));

    ret = gpio_to_irq(buttons[0].gpio);

    if (ret < 0) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    button_irqs[0] = ret;
```

```
    pr_info("Successfully requested BUTTON1 IRQ # %d\n", button_irqs[0]);

    ret = request_irq(button_irqs[0], button_isr,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "gpiomod#button1", NULL);

    if (ret) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    ret = gpio_to_irq(buttons[1].gpio);

    if (ret < 0) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    button_irqs[1] = ret;

    pr_info("Successfully requested BUTTON2 IRQ # %d\n", button_irqs[1]);

    ret = request_irq(button_irqs[1], button_isr,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "gpiomod#button2", NULL);

    if (ret) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail3;
    }

    return 0;

/* Удаление проделанных настроек. */
fail3:
    free_irq(button_irqs[0], NULL);

fail2:
    gpio_free_array(buttons, ARRAY_SIZE(leds));

fail1:
    gpio_free_array(leds, ARRAY_SIZE(leds));

    return ret;
}

static void __exit intrpt_exit(void)
```

```
{
    int i;

    pr_info("%s\n", __func__);

    /* Свободные прерывания. */
    free_irq(button_irqs[0], NULL);
    free_irq(button_irqs[1], NULL);

    /* Отключение всех светодиодов. */
    for (i = 0; i < ARRAY_SIZE(leds); i++)
        gpio_set_value(leds[i].gpio, 0);

    /* Снятие регистрации. */
    gpio_free_array(leds, ARRAY_SIZE(leds));
    gpio_free_array(buttons, ARRAY_SIZE(buttons));
}

module_init(intrpt_init);
module_exit(intrpt_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Handle some GPIO interrupts");
```

15.3 Нижняя половина

Предположим, вам нужно проделать ряд операций внутри подпрограммы прерывания. Стандартный способ реализовать это, не лишая прерывание доступности на долгое время, предполагает его совмещение с тасклетом. Так вы переложите основную работу на планировщик.

Ниже представлена изменённая версия предыдущего примера, в которой при срабатывании прерывания запускается дополнительная задача.

Код bottomhalf.c:

```
/*
 * bottomhalf.c – Обработка верхней и нижней частей прерывания.
 *
 * За основу взят пример RPi Стефана Вендлера (devnull@kaltpost.de)
 * из репозитория https://github.com/wendlers/rpi-kmod-samples
 *
 * При нажатии одной кнопки светодиод загорается, а при нажатии другой
 * гаснет.
 */
```

```c
#include <linux/delay.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/module.h>

/* Макрос DECLARE_TASKLET_OLD присутствует для совместимости.
 * См. https://lwn.net/Articles/830964/
 */
#ifndef DECLARE_TASKLET_OLD
#define DECLARE_TASKLET_OLD(arg1, arg2) DECLARE_TASKLET(arg1, arg2, 0L)
#endif

static int button_irqs[] = { -1, -1 };

/* Определение вводов-выводов для светодиодов.
 * Номера вводов-выводов можно изменить.
 */
static struct gpio leds[] = { { 4, GPIOF_OUT_INIT_LOW, "LED 1" } };

/* Определение вводов-выводов для BUTTONS.
 * Номера вводов-выводов можно изменить.
 */
static struct gpio buttons[] = {
    { 17, GPIOF_IN, "LED 1 ON BUTTON" },
    { 18, GPIOF_IN, "LED 1 OFF BUTTON" },
};

/* Тасклет, содержащий большой объём обработки. */
static void bottomhalf_tasklet_fn(unsigned long data)
{
    pr_info("Bottom half tasklet starts\n");
    /* Выполнение длительных действий. */
    mdelay(500);
    pr_info("Bottom half tasklet ends\n");
}

static DECLARE_TASKLET_OLD(buttontask, bottomhalf_tasklet_fn);

/* Функция прерывания, активизируемая при нажатии кнопки. */
static irqreturn_t button_isr(int irq, void *data)
{
    /* Быстрое выполнение действия прямо сейчас. */
    if (irq == button_irqs[0] && !gpio_get_value(leds[0].gpio))
        gpio_set_value(leds[0].gpio, 1);
    else if (irq == button_irqs[1] && gpio_get_value(leds[0].gpio))
        gpio_set_value(leds[0].gpio, 0);
```

```
    /* Неспешное выполнение остального через планировщик. */
    tasklet_schedule(&buttontask);

    return IRQ_HANDLED;
}

static int __init bottomhalf_init(void)
{
    int ret = 0;

    pr_info("%s\n", __func__);

    /* Регистрация вводов-выводов светодиодов. */
    ret = gpio_request_array(leds, ARRAY_SIZE(leds));

    if (ret) {
        pr_err("Unable to request GPIOs for LEDs: %d\n", ret);
        return ret;
    }

    /* Регистрация вводов-выводов BUTTONS. */
    ret = gpio_request_array(buttons, ARRAY_SIZE(buttons));

    if (ret) {
        pr_err("Unable to request GPIOs for BUTTONs: %d\n", ret);
        goto fail1;
    }

    pr_info("Current button1 value: %d\n", gpio_get_value(buttons[0].gpio));

    ret = gpio_to_irq(buttons[0].gpio);

    if (ret < 0) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    button_irqs[0] = ret;

    pr_info("Successfully requested BUTTON1 IRQ # %d\n", button_irqs[0]);

    ret = request_irq(button_irqs[0], button_isr,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "gpiomod#button1", NULL);

    if (ret) {
        pr_err("Unable to request IRQ: %d\n", ret);
```

```
        goto fail2;
    }

    ret = gpio_to_irq(buttons[1].gpio);

    if (ret < 0) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    button_irqs[1] = ret;

    pr_info("Successfully requested BUTTON2 IRQ # %d\n", button_irqs[1]);

    ret = request_irq(button_irqs[1], button_isr,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "gpiomod#button2", NULL);

    if (ret) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail3;
    }

    return 0;

/* Удаление проделанных настроек. */
fail3:
    free_irq(button_irqs[0], NULL);

fail2:
    gpio_free_array(buttons, ARRAY_SIZE(leds));

fail1:
    gpio_free_array(leds, ARRAY_SIZE(leds));

    return ret;
}

static void __exit bottomhalf_exit(void)
{
    int i;

    pr_info("%s\n", __func__);

    /* Освобождение прерываний. */
    free_irq(button_irqs[0], NULL);
    free_irq(button_irqs[1], NULL);
```

```
    /* Отключение всех светодиодов. */
    for (i = 0; i < ARRAY_SIZE(leds); i++)
        gpio_set_value(leds[i].gpio, 0);

    /* Отмена регистрации. */
    gpio_free_array(leds, ARRAY_SIZE(leds));
    gpio_free_array(buttons, ARRAY_SIZE(buttons));
}

module_init(bottomhalf_init);
module_exit(bottomhalf_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Interrupt with top and bottom half");
```

16. Криптография

На заре становления интернета все его пользователи полностью доверяли друг другу…но ничего хорошего из этого не вышло. Изначально это руководство писалось в безмятежную эру, в которой мало кого заботила криптография – по крайней мере, разработчиков ядра. Сегодня же времена совсем другие. Для обработки криптографии в ядре реализован собственный API, предоставляющий стандартные методы шифрования/дешифровании и ваши любимые хеш-функции.

16.1 Хеш-функции

Вычисление и проверка хешей является стандартной операцией. Ниже приведён пример вычисления хеша sha256 в модуле ядра.

Код cryptosha256.c:

```
/*
 * cryptosha256.c
 */
#include <crypto/internal/hash.h>
#include <linux/module.h>

#define SHA256_LENGTH 32

static void show_hash_result(char *plaintext, char *hash_sha256)
{
    int i;
    char str[SHA256_LENGTH * 2 + 1];
```

```
    pr_info("sha256 test for string: \"%s\"\n", plaintext);
    for (i = 0; i < SHA256_LENGTH; i++)
        sprintf(&str[i * 2], "%02x", (unsigned char)hash_sha256[i]);
    str[i * 2] = 0;
    pr_info("%s\n", str);
}

static int cryptosha256_init(void)
{
    char *plaintext = "This is a test";
    char hash_sha256[SHA256_LENGTH];
    struct crypto_shash *sha256;
    struct shash_desc *shash;

    sha256 = crypto_alloc_shash("sha256", 0, 0);
    if (IS_ERR(sha256))
        return -1;

    shash = kmalloc(sizeof(struct shash_desc) + crypto_shash_descsize(sha256),
                    GFP_KERNEL);
    if (!shash)
        return -ENOMEM;

    shash->tfm = sha256;

    if (crypto_shash_init(shash))
        return -1;

    if (crypto_shash_update(shash, plaintext, strlen(plaintext)))
        return -1;

    if (crypto_shash_final(shash, hash_sha256))
        return -1;

    kfree(shash);
    crypto_free_shash(sha256);

    show_hash_result(plaintext, hash_sha256);

    return 0;
}

static void cryptosha256_exit(void)
{
}

module_init(cryptosha256_init);
```

```
module_exit(cryptosha256_exit);

MODULE_DESCRIPTION("sha256 hash test");
MODULE_LICENSE("GPL");
```

Установите модуль:

```
sudo insmod cryptosha256.ko
sudo dmesg
```

И увидите, что для тестовой строки вычисляется хеш.

В завершение удалите тестовый модуль:

```
sudo rmmod cryptosha256
```

16.2 Шифрование с симметричным ключом

Вот пример симметричного шифрования строки с помощью алгоритма AES и пароля.

Код cryptosk.c:

```
/*
 * cryptosk.c
 */
#include <crypto/internal/skcipher.h>
#include <linux/crypto.h>
#include <linux/module.h>
#include <linux/random.h>
#include <linux/scatterlist.h>

#define SYMMETRIC_KEY_LENGTH 32
#define CIPHER_BLOCK_SIZE 16

struct tcrypt_result {
    struct completion completion;
    int err;
};

struct skcipher_def {
    struct scatterlist sg;
    struct crypto_skcipher *tfm;
    struct skcipher_request *req;
```

```
    struct tcrypt_result result;
    char *scratchpad;
    char *ciphertext;
    char *ivdata;
};

static struct skcipher_def sk;

static void test_skcipher_finish(struct skcipher_def *sk)
{
    if (sk->tfm)
        crypto_free_skcipher(sk->tfm);
    if (sk->req)
        skcipher_request_free(sk->req);
    if (sk->ivdata)
        kfree(sk->ivdata);
    if (sk->scratchpad)
        kfree(sk->scratchpad);
    if (sk->ciphertext)
        kfree(sk->ciphertext);
}

static int test_skcipher_result(struct skcipher_def *sk, int rc)
{
    switch (rc) {
    case 0:
        break;
    case -EINPROGRESS || -EBUSY:
        rc = wait_for_completion_interruptible(&sk->result.completion);
        if (!rc && !sk->result.err) {
            reinit_completion(&sk->result.completion);
            break;
        }
        pr_info("skcipher encrypt returned with %d result %d\n", rc,
                sk->result.err);
        break;
    default:
        pr_info("skcipher encrypt returned with %d result %d\n", rc,
                sk->result.err);
        break;
    }

    init_completion(&sk->result.completion);

    return rc;
}

static void test_skcipher_callback(struct crypto_async_request *req, int error)
```

```
{
    struct tcrypt_result *result = req->data;

    if (error == -EINPROGRESS)
        return;

    result->err = error;
    complete(&result->completion);
    pr_info("Encryption finished successfully\n");

    /* Расшифровка данных. */
#if 0
    memset((void*)sk.scratchpad, '-', CIPHER_BLOCK_SIZE);
    ret = crypto_skcipher_decrypt(sk.req);
    ret = test_skcipher_result(&sk, ret);
    if (ret)
        return;

    sg_copy_from_buffer(&sk.sg, 1, sk.scratchpad, CIPHER_BLOCK_SIZE);
    sk.scratchpad[CIPHER_BLOCK_SIZE-1] = 0;

    pr_info("Decryption request successful\n");
    pr_info("Decrypted: %s\n", sk.scratchpad);
#endif
}

static int test_skcipher_encrypt(char *plaintext, char *password,
                                 struct skcipher_def *sk)
{
    int ret = -EFAULT;
    unsigned char key[SYMMETRIC_KEY_LENGTH];

    if (!sk->tfm) {
        sk->tfm = crypto_alloc_skcipher("cbc-aes-aesni", 0, 0);
        if (IS_ERR(sk->tfm)) {
            pr_info("could not allocate skcipher handle\n");
            return PTR_ERR(sk->tfm);
        }
    }

    if (!sk->req) {
        sk->req = skcipher_request_alloc(sk->tfm, GFP_KERNEL);
        if (!sk->req) {
            pr_info("could not allocate skcipher request\n");
            ret = -ENOMEM;
            goto out;
        }
    }
```

```
    skcipher_request_set_callback(sk->req, CRYPTO_TFM_REQ_MAY_BACKLOG,
                                  test_skcipher_callback, &sk->result);

    /* Очистка ключа. */
    memset((void *)key, '\0', SYMMETRIC_KEY_LENGTH);

    /* Использование самого популярного в мире пароля. */
    sprintf((char *)key, "%s", password);

    /* AES 256 с заданным симметричным ключом. */
    if (crypto_skcipher_setkey(sk->tfm, key, SYMMETRIC_KEY_LENGTH)) {
        pr_info("key could not be set\n");
        ret = -EAGAIN;
        goto out;
    }
    pr_info("Symmetric key: %s\n", key);
    pr_info("Plaintext: %s\n", plaintext);

    if (!sk->ivdata) {
        /* См. https://en.wikipedia.org/wiki/Initialization_vector */
        sk->ivdata = kmalloc(CIPHER_BLOCK_SIZE, GFP_KERNEL);
        if (!sk->ivdata) {
            pr_info("could not allocate ivdata\n");
            goto out;
        }
        get_random_bytes(sk->ivdata, CIPHER_BLOCK_SIZE);
    }

    if (!sk->scratchpad) {
        /* Текст для шифрования. */
        sk->scratchpad = kmalloc(CIPHER_BLOCK_SIZE, GFP_KERNEL);
        if (!sk->scratchpad) {
            pr_info("could not allocate scratchpad\n");
            goto out;
        }
    }
    sprintf((char *)sk->scratchpad, "%s", plaintext);

    sg_init_one(&sk->sg, sk->scratchpad, CIPHER_BLOCK_SIZE);
    skcipher_request_set_crypt(sk->req, &sk->sg, &sk->sg, CIPHER_BLOCK_SIZE,
                               sk->ivdata);
    init_completion(&sk->result.completion);

    /* Шифрование данных. */
    ret = crypto_skcipher_encrypt(sk->req);
    ret = test_skcipher_result(sk, ret);
    if (ret)
```

```
        goto out;

    pr_info("Encryption request successful\n");

out:
    return ret;
}

static int cryptoapi_init(void)
{
    /* Самый популярный пароль в мире. */
    char *password = "password123";

    sk.tfm = NULL;
    sk.req = NULL;
    sk.scratchpad = NULL;
    sk.ciphertext = NULL;
    sk.ivdata = NULL;

    test_skcipher_encrypt("Testing", password, &sk);
    return 0;
}

static void cryptoapi_exit(void)
{
    test_skcipher_finish(&sk);
}

module_init(cryptoapi_init);
module_exit(cryptoapi_exit);

MODULE_DESCRIPTION("Symmetric key encryption example");
MODULE_LICENSE("GPL");
```

17. Драйвер виртуального устройства ввода

Драйвер устройства ввода – это модуль, обеспечивающий возможность взаимодействия с интерактивным устройством через события. Например, клавиатура может отправлять событие нажатия или отпускания клавиши, сообщая ядру наши намерения. Драйвер устройства ввода выделяет новую структуру ввода с помощью функции `input_allocate_device()`, настраивает в ней битовые поля, ID устройства, версию и прочее, после чего регистрирует его через вызов `input_register_device()`.
В качестве примера приведу `vinput` – API, обеспечивающий удобство разработки драйверов виртуальных устройств. Этот драйвер должен экспортировать

`vinput_device()`, содержащую имя виртуального устройства, и структуру `vinput_ops`, которая описывает:

- Функцию инициализации: `init()`
- Функцию внедрения события ввода: `send()`
- Функцию обратного чтения: `read()`

Далее с помощью `vinput_register_device()` и `vinput_unregister_device()` новое устройство добавляется в список поддерживаемых виртуальных устройств ввода:

```
int init(struct vinput *);
```

Этой функции передаётся `struct vinput`, уже инициализированная с помощью выделенной `struct input_dev`. Функция `init()` отвечает за инициализацию возможностей устройства ввода и его регистрацию:

```
int send(struct vinput *, char *, int);
```

Эта функция получает пользовательскую строку и внедряет соответствующее событие с помощью вызова `input_report_XXXX` или `input_event`. Данная строка уже скопирована от пользователя:

```
int read(struct vinput *, char *, int);
```

Эта функция используется для отладки и должна заполнять параметр буфера последним событием, отправленным в формате виртуального устройства ввода. После этого буфер копируется пользователю. Устройства `vinput` создаются и уничтожаются с помощью `sysfs`, а внедрение событий выполняется через узел `/dev`. Имя устройства используется пользовательским пространством для экспорта нового виртуального устройства ввода. Структура `class_attribute` аналогична другим типам атрибутов, о которых шла речь в разделе 8:

```
struct class_attribute {
    struct attribute attr;
    ssize_t (*show)(struct class *class, struct class_attribute *attr,
                    char *buf);
    ssize_t (*store)(struct class *class, struct class_attribute *attr,
                    const char *buf, size_t count);
};
```

В `vinput.c` макрос `CLASS_ATTR_WO(export/unexport)`, определённый в `include/linux/device.h` (в данном случае `device.h` включён в `include/linux/input.h`) сгенерирует структуры `class_attribute`, названные `class_attr_export/unexport`.

После этого он поместит их в массив `vinput_class_attrs`, и макрос `ATTRIBUTE_GROUPS(vinput_class)` сгенерирует `struct attribute_group vinput_class_group`, которую нужно будет присвоить в `vinput_class`. В завершении выполняется вызов `class_register(&vinput_class)` для создания атрибутов в `sysfs`.

Для создания записи `sysfs vinputX` и узла `/dev`:

```
echo "vkbd" | sudo tee /sys/class/vinput/export
```

Для обратного экспорта устройства нужно `echo` его ID в `unexport`.

```
echo "0" | sudo tee /sys/class/vinput/unexport
```

Код vinput.h:

```
/*
 * vinput.h
 */

#ifndef VINPUT_H
#define VINPUT_H

#include <linux/input.h>
#include <linux/spinlock.h>

#define VINPUT_MAX_LEN 128
#define MAX_VINPUT 32
#define VINPUT_MINORS MAX_VINPUT

#define dev_to_vinput(dev) container_of(dev, struct vinput, dev)

struct vinput_device;

struct vinput {
    long id;
    long devno;
    long last_entry;
    spinlock_t lock;

    void *priv_data;
```

```
    struct device dev;
    struct list_head list;
    struct input_dev *input;
    struct vinput_device *type;
};

struct vinput_ops {
    int (*init)(struct vinput *);
    int (*kill)(struct vinput *);
    int (*send)(struct vinput *, char *, int);
    int (*read)(struct vinput *, char *, int);
};

struct vinput_device {
    char name[16];
    struct list_head list;
    struct vinput_ops *ops;
};

int vinput_register(struct vinput_device *dev);
void vinput_unregister(struct vinput_device *dev);


#endif
```

Код vinput.c:

```
/*
 * vinput.c
 */

#include <linux/cdev.h>
#include <linux/input.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/spinlock.h>

#include <asm/uaccess.h>

#include "vinput.h"

#define DRIVER_NAME "vinput"

#define dev_to_vinput(dev) container_of(dev, struct vinput, dev)

static DECLARE_BITMAP(vinput_ids, VINPUT_MINORS);
```

```c
static LIST_HEAD(vinput_devices);
static LIST_HEAD(vinput_vdevices);

static int vinput_dev;
static struct spinlock vinput_lock;
static struct class vinput_class;

/* Поиск имени устройства vinput в связанном списке vinput_devices,
 * добавленном в vinput_register().
 */
static struct vinput_device *vinput_get_device_by_type(const char *type)
{
    int found = 0;
    struct vinput_device *vinput;
    struct list_head *curr;

    spin_lock(&vinput_lock);
    list_for_each (curr, &vinput_devices) {
        vinput = list_entry(curr, struct vinput_device, list);
        if (vinput && strncmp(type, vinput->name, strlen(vinput->name)) == 0) {
            found = 1;
            break;
        }
    }
    spin_unlock(&vinput_lock);

    if (found)
        return vinput;
    return ERR_PTR(-ENODEV);
}

/* Поиск ID виртуального устройства в связанном списке vinput_vdevices,
 * добавленном в vinput_alloc_vdevice().
 */
static struct vinput *vinput_get_vdevice_by_id(long id)
{
    struct vinput *vinput = NULL;
    struct list_head *curr;

    spin_lock(&vinput_lock);
    list_for_each (curr, &vinput_vdevices) {
        vinput = list_entry(curr, struct vinput, list);
        if (vinput && vinput->id == id)
            break;
    }
    spin_unlock(&vinput_lock);
```

```
    if (vinput && vinput->id == id)
        return vinput;
    return ERR_PTR(-ENODEV);
}

static int vinput_open(struct inode *inode, struct file *file)
{
    int err = 0;
    struct vinput *vinput = NULL;

    vinput = vinput_get_vdevice_by_id(iminor(inode));

    if (IS_ERR(vinput))
        err = PTR_ERR(vinput);
    else
        file->private_data = vinput;

    return err;
}

static int vinput_release(struct inode *inode, struct file *file)
{
    return 0;
}

static ssize_t vinput_read(struct file *file, char __user *buffer, size_t count,
                           loff_t *offset)
{
    int len;
    char buff[VINPUT_MAX_LEN + 1];
    struct vinput *vinput = file->private_data;

    len = vinput->type->ops->read(vinput, buff, count);

    if (*offset > len)
        count = 0;
    else if (count + *offset > VINPUT_MAX_LEN)
        count = len - *offset;

    if (raw_copy_to_user(buffer, buff + *offset, count))
        count = -EFAULT;

    *offset += count;

    return count;
}

static ssize_t vinput_write(struct file *file, const char __user *buffer,
```

```
                                size_t count, loff_t *offset)
{
    char buff[VINPUT_MAX_LEN + 1];
    struct vinput *vinput = file->private_data;

    memset(buff, 0, sizeof(char) * (VINPUT_MAX_LEN + 1));

    if (count > VINPUT_MAX_LEN) {
        dev_warn(&vinput->dev, "Too long. %d bytes allowed\n", VINPUT_MAX_LEN);
        return -EINVAL;
    }

    if (raw_copy_from_user(buff, buffer, count))
        return -EFAULT;

    return vinput->type->ops->send(vinput, buff, count);
}

static const struct file_operations vinput_fops = {
    .owner = THIS_MODULE,
    .open = vinput_open,
    .release = vinput_release,
    .read = vinput_read,
    .write = vinput_write,
};

static void vinput_unregister_vdevice(struct vinput *vinput)
{
    input_unregister_device(vinput->input);
    if (vinput->type->ops->kill)
        vinput->type->ops->kill(vinput);
}

static void vinput_destroy_vdevice(struct vinput *vinput)
{
    /* Сначала удаление из списка. */
    spin_lock(&vinput_lock);
    list_del(&vinput->list);
    clear_bit(vinput->id, vinput_ids);
    spin_unlock(&vinput_lock);

    module_put(THIS_MODULE);

    kfree(vinput);
}

static void vinput_release_dev(struct device *dev)
{
```

```
    struct vinput *vinput = dev_to_vinput(dev);
    int id = vinput->id;

    vinput_destroy_vdevice(vinput);

    pr_debug("released vinput%d.\n", id);
}

static struct vinput *vinput_alloc_vdevice(void)
{
    int err;
    struct vinput *vinput = kzalloc(sizeof(struct vinput), GFP_KERNEL);

    try_module_get(THIS_MODULE);

    memset(vinput, 0, sizeof(struct vinput));

    spin_lock_init(&vinput->lock);

    spin_lock(&vinput_lock);
    vinput->id = find_first_zero_bit(vinput_ids, VINPUT_MINORS);
    if (vinput->id >= VINPUT_MINORS) {
        err = -ENOBUFS;
        goto fail_id;
    }
    set_bit(vinput->id, vinput_ids);
    list_add(&vinput->list, &vinput_vdevices);
    spin_unlock(&vinput_lock);

    /* Выделение устройства ввода. */
    vinput->input = input_allocate_device();
    if (vinput->input == NULL) {
        pr_err("vinput: Cannot allocate vinput input device\n");
        err = -ENOMEM;
        goto fail_input_dev;
    }

    /* Инициализация устройства. */
    vinput->dev.class = &vinput_class;
    vinput->dev.release = vinput_release_dev;
    vinput->dev.devt = MKDEV(vinput_dev, vinput->id);
    dev_set_name(&vinput->dev, DRIVER_NAME "%lu", vinput->id);

    return vinput;

fail_input_dev:
    spin_lock(&vinput_lock);
    list_del(&vinput->list);
```

```
fail_id:
    spin_unlock(&vinput_lock);
    module_put(THIS_MODULE);
    kfree(vinput);

    return ERR_PTR(err);
}

static int vinput_register_vdevice(struct vinput *vinput)
{
    int err = 0;

    /* Регистрация устройства ввода. */
    vinput->input->name = vinput->type->name;
    vinput->input->phys = "vinput";
    vinput->input->dev.parent = &vinput->dev;

    vinput->input->id.bustype = BUS_VIRTUAL;
    vinput->input->id.product = 0x0000;
    vinput->input->id.vendor = 0x0000;
    vinput->input->id.version = 0x0000;

    err = vinput->type->ops->init(vinput);

    if (err == 0)
        dev_info(&vinput->dev, "Registered virtual input %s %ld\n",
                 vinput->type->name, vinput->id);

    return err;
}

static ssize_t export_store(struct class *class, struct class_attribute *attr,
                            const char *buf, size_t len)
{
    int err;
    struct vinput *vinput;
    struct vinput_device *device;

    device = vinput_get_device_by_type(buf);
    if (IS_ERR(device)) {
        pr_info("vinput: This virtual device isn't registered\n");
        err = PTR_ERR(device);
        goto fail;
    }

    vinput = vinput_alloc_vdevice();
    if (IS_ERR(vinput)) {
        err = PTR_ERR(vinput);
```

```
        goto fail;
    }

    vinput->type = device;
    err = device_register(&vinput->dev);
    if (err < 0)
        goto fail_register;

    err = vinput_register_vdevice(vinput);
    if (err < 0)
        goto fail_register_vinput;

    return len;

fail_register_vinput:
    device_unregister(&vinput->dev);
fail_register:
    vinput_destroy_vdevice(vinput);
fail:
    return err;
}
/* Этот макрос генерирует структуру class_attr_export и export_store() */
static CLASS_ATTR_WO(export);

static ssize_t unexport_store(struct class *class, struct class_attribute *attr,
                              const char *buf, size_t len)
{
    int err;
    unsigned long id;
    struct vinput *vinput;

    err = kstrtol(buf, 10, &id);
    if (err) {
        err = -EINVAL;
        goto failed;
    }

    vinput = vinput_get_vdevice_by_id(id);
    if (IS_ERR(vinput)) {
        pr_err("vinput: No such vinput device %ld\n", id);
        err = PTR_ERR(vinput);
        goto failed;
    }

    vinput_unregister_vdevice(vinput);
    device_unregister(&vinput->dev);

    return len;
```

```
failed:
    return err;
}
/* Этот макрос генерирует структуру class_attr_unexport
 * и unexport_store().
 */
static CLASS_ATTR_WO(unexport);

static struct attribute *vinput_class_attrs[] = {
    &class_attr_export.attr,
    &class_attr_unexport.attr,
    NULL,
};

/* Этот макрос генерирует структуру vinput_class_groups. */
ATTRIBUTE_GROUPS(vinput_class);

static struct class vinput_class = {
    .name = "vinput",
    .owner = THIS_MODULE,
    .class_groups = vinput_class_groups,
};

int vinput_register(struct vinput_device *dev)
{
    spin_lock(&vinput_lock);
    list_add(&dev->list, &vinput_devices);
    spin_unlock(&vinput_lock);

    pr_info("vinput: registered new virtual input device '%s'\n", dev->name);

    return 0;
}
EXPORT_SYMBOL(vinput_register);

void vinput_unregister(struct vinput_device *dev)
{
    struct list_head *curr, *next;

    /* Сначала удаление из списка. */
    spin_lock(&vinput_lock);
    list_del(&dev->list);
    spin_unlock(&vinput_lock);

    /* Снятие регистрации всех устройств этого типа. */
    list_for_each_safe (curr, next, &vinput_vdevices) {
        struct vinput *vinput = list_entry(curr, struct vinput, list);
        if (vinput && vinput->type == dev) {
```

```
            vinput_unregister_vdevice(vinput);
            device_unregister(&vinput->dev);
        }
    }

    pr_info("vinput: unregistered virtual input device '%s'\n", dev->name);
}
EXPORT_SYMBOL(vinput_unregister);

static int __init vinput_init(void)
{
    int err = 0;

    pr_info("vinput: Loading virtual input driver\n");

    vinput_dev = register_chrdev(0, DRIVER_NAME, &vinput_fops);
    if (vinput_dev < 0) {
        pr_err("vinput: Unable to allocate char dev region\n");
        goto failed_alloc;
    }

    spin_lock_init(&vinput_lock);

    err = class_register(&vinput_class);
    if (err < 0) {
        pr_err("vinput: Unable to register vinput class\n");
        goto failed_class;
    }

    return 0;
failed_class:
    class_unregister(&vinput_class);
failed_alloc:
    return err;
}

static void __exit vinput_end(void)
{
    pr_info("vinput: Unloading virtual input driver\n");

    unregister_chrdev(vinput_dev, DRIVER_NAME);
    class_unregister(&vinput_class);
}

module_init(vinput_init);
module_exit(vinput_end);

MODULE_LICENSE("GPL");
```

```
MODULE_DESCRIPTION("Emulate input events");
```

Здесь мы рассматриваем виртуальную клавиатуру как один из примеров использования `vinput`. Она поддерживает все коды клавиш `KEY_MAX`. Внедрение производится в формате `KEY_CODE`, как это определено в `include/linux/input.h`.
Положительное значение означает `KEY_PRESS`, а отрицательное `KEY_RELEASE`. Эта клавиатура поддерживает повторение ввода, когда клавиша остаётся нажатой длительное время. Код ниже демонстрирует работу данной симуляции.

Симулирует нажатие «g» (`KEY_G = 34`):

```
echo "+34" | sudo tee /dev/vinput0
```

Симулирует отпускание «g» (`KEY_G = 34`):

```
echo "-34" | sudo tee /dev/vinput0
```

Код vkbd.c:

```
/*
 * vkbd.c
 */

#include <linux/init.h>
#include <linux/input.h>
#include <linux/module.h>
#include <linux/spinlock.h>

#include "vinput.h"

#define VINPUT_KBD "vkbd"
#define VINPUT_RELEASE 0
#define VINPUT_PRESS 1

static unsigned short vkeymap[KEY_MAX];

static int vinput_vkbd_init(struct vinput *vinput)
{
    int i;

    /* Устанавливает битовое поле ввода. */
    vinput->input->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REP);
    vinput->input->keycodesize = sizeof(unsigned short);
    vinput->input->keycodemax = KEY_MAX;
    vinput->input->keycode = vkeymap;
```

```
    for (i = 0; i < KEY_MAX; i++)
        set_bit(vkeymap[i], vinput->input->keybit);

    /* vinput поможет выделить новую структуру устройства ввода через
     * input_allocate_device(), что позволит с лёгкостью его
     * зарегистрировать.
     */
    return input_register_device(vinput->input);
}

static int vinput_vkbd_read(struct vinput *vinput, char *buff, int len)
{
    spin_lock(&vinput->lock);
    len = snprintf(buff, len, "%+ld\n", vinput->last_entry);
    spin_unlock(&vinput->lock);

    return len;
}

static int vinput_vkbd_send(struct vinput *vinput, char *buff, int len)
{
    int ret;
    long key = 0;
    short type = VINPUT_PRESS;

    /* Определяем, какое было получено событие
     * (нажатие или отпускание) и сохраняем это состояние.
     */
    if (buff[0] == '+')
        ret = kstrtol(buff + 1, 10, &key);
    else
        ret = kstrtol(buff, 10, &key);
    if (ret)
        dev_err(&vinput->dev, "error during kstrtol: -%d\n", ret);
    spin_lock(&vinput->lock);
    vinput->last_entry = key;
    spin_unlock(&vinput->lock);

    if (key < 0) {
        type = VINPUT_RELEASE;
        key = -key;
    }

    dev_info(&vinput->dev, "Event %s code %ld\n",
             (type == VINPUT_RELEASE) ? "VINPUT_RELEASE" : "VINPUT_PRESS", key);

    /* Передаём полученное состояние подсистеме ввода. */
```

```
    input_report_key(vinput->input, key, type);
    /* Сообщаем подсистеме ввода, что передача закончена. */
    input_sync(vinput->input);

    return len;
}

static struct vinput_ops vkbd_ops = {
    .init = vinput_vkbd_init,
    .send = vinput_vkbd_send,
    .read = vinput_vkbd_read,
};

static struct vinput_device vkbd_dev = {
    .name = VINPUT_KBD,
    .ops = &vkbd_ops,
};

static int __init vkbd_init(void)
{
    int i;

    for (i = 0; i < KEY_MAX; i++)
        vkeymap[i] = i;
    return vinput_register(&vkbd_dev);
}

static void __exit vkbd_end(void)
{
    vinput_unregister(&vkbd_dev);
}

module_init(vkbd_init);
module_exit(vkbd_end);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Emulate keyboard input events through /dev/vinput");
```

18. Стандартизация интерфейсов: модель устройства

К этому моменту мы рассмотрели все виды модулей, выполняющие всевозможные задачи, но в их интерфейсах не было согласованности с остальной частью ядра. Для внесения согласованности, которая бы обеспечила как минимум стандартизированный способ запускать, приостанавливать и возобновлять работу устройства, была добавлена модель устройства.

Ниже показан её пример, который вы можете использовать в качестве шаблона для добавления собственных функций приостановки, возобновления и прочего.

Код devicemodel.c:

```c
/*
 * devicemodel.c
 */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/platform_device.h>

struct devicemodel_data {
    char *greeting;
    int number;
};

static int devicemodel_probe(struct platform_device *dev)
{
    struct devicemodel_data *pd =
        (struct devicemodel_data *)(dev->dev.platform_data);

    pr_info("devicemodel probe\n");
    pr_info("devicemodel greeting: %s; %d\n", pd->greeting, pd->number);

    /* Код инициализации устройства. */

    return 0;
}

static int devicemodel_remove(struct platform_device *dev)
{
    pr_info("devicemodel example removed\n");

    /* Код удаления устройства. */

    return 0;
}

static int devicemodel_suspend(struct device *dev)
{
    pr_info("devicemodel example suspend\n");

    /* Код приостановки устройства. */

    return 0;
```

```
}

static int devicemodel_resume(struct device *dev)
{
    pr_info("devicemodel example resume\n");

    /* Код возобновления работы устройства. */

    return 0;
}

static const struct dev_pm_ops devicemodel_pm_ops = {
    .suspend = devicemodel_suspend,
    .resume = devicemodel_resume,
    .poweroff = devicemodel_suspend,
    .freeze = devicemodel_suspend,
    .thaw = devicemodel_resume,
    .restore = devicemodel_resume,
};

static struct platform_driver devicemodel_driver = {
    .driver =
        {
            .name = "devicemodel_example",
            .owner = THIS_MODULE,
            .pm = &devicemodel_pm_ops,
        },
    .probe = devicemodel_probe,
    .remove = devicemodel_remove,
};

static int devicemodel_init(void)
{
    int ret;

    pr_info("devicemodel init\n");

    ret = platform_driver_register(&devicemodel_driver);

    if (ret) {
        pr_err("Unable to register driver\n");
        return ret;
    }

    return 0;
}

static void devicemodel_exit(void)
```

```
{
    pr_info("devicemodel exit\n");
    platform_driver_unregister(&devicemodel_driver);
}

module_init(devicemodel_init);
module_exit(devicemodel_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Linux Device Model example");
```

19. Оптимизации

19.1 Условия likely и unlikely

Иногда вам может потребоваться максимально быстрое выполнение кода, особенно если он обрабатывает прерывание или выполняет нечто, способное вызвать значительную задержку. Если ваш код содержит логические условия, и вы знаете, что эти условия практически всегда оцениваются как `true` либо `false`, тогда можете позволить компилятору выполнить соответствующую оптимизацию с помощью макросов `likely` и `unlikely`.

К примеру, при выделении памяти вы практически всегда ожидаете успешного завершения операции:

```
bvl = bvec_alloc(gfp_mask, nr_iovecs, &idx);
if (unlikely(!bvl)) {
    mempool_free(bio, bio_pool);
    bio = NULL;
    goto out;
}
```

Когда используется макрос `unlikely`, компилятор изменяет вывод машинной инструкции, чтобы код продолжал выполнение по ветке `false` и делал переход, только когда условие `true`. Это позволяет избежать очистки конвейера процессора. При использовании макроса `likely` происходит противоположное.

20. Важные нюансы

20.1 Использование стандартных библиотек

Этого делать нельзя. В модуле ядра допустимо использовать исключительно функции ядра, которые вы можете найти в `/proc/kallsyms`.

20.2 Отключение прерываний

Вам может потребоваться делать это ненадолго, что вполне нормально. Если же вы впоследствии их не включите, то система зависнет, и её придётся отключить.

21. Дальнейшие шаги

Для тех, кто серьёзно заинтересован в освоении программирования ядра, рекомендую ознакомиться с ресурсом kernelnewbies.org и поддиректорией Documentation в исходном коде, которая даёт неплохие базовые понятия для дальнейшего изучения темы, хотя местами могла бы быть написана и получше. Кроме того, как сказал сам Линус Торвальдс, лучший способ изучить ядро – это самостоятельно читать его исходный код.

Если вы желаете внести свой вклад в данное пособие или заметили в нём какие-либо серьёзные недочёты, создайте по этой теме запрос на https://github.com/sysprog21/lkmpg. Будем признательны за ваши пул-реквесты.

Успехов!

Эпилог

Перевод подготовлен редакционной командой компании RUVDS.com. Первоначально руководство выходило по частям - в блоге компании на Хабр:

- Часть 1
- Часть 2
- Часть 3
- Часть 4
- Часть 5
- Часть 6
- Часть 7

Особая благодарность в подготовке электронной версии и перевода:

переводчик: Bright_Translate

редактор: JohurN

линуксоид и руководитель спецпроектов RUVDS: Dlinyj

Если у вас возникнут какие-либо замечания по качеству перевода, либо вы найдёте какие-либо опечатки, пожалуйста сообщите нам по почте dlinyj@gmail.com