Практикум: модули ядра Linux

Конспект с примерами и упражнения с задачами

Автор: Олег Цилюрик

Редакция **6.245**

18.03.2015 г.

Оглавление

Содержание

Введение

Эта практикум является конспектом курса практических занятий по написанию кодов ядра Linux. Занятия организовывались компанией Global Logic (http://www.globallogic.com/) для сотрудников (программистов-разработчиков) украинских отделений (http://globallogic.com.ua) компании. Этот курс практических занятий основывается на моих же материалах проводимых ранее тренингов «Программирование модулей ядра Linux», текст и примеры кодов которых можно найти, например, здесь: http://mylinuxprog.blogspot.com/2015/01/linux.html. Зачем, при наличии этих предыдущих текстов, готовить новый материал, в чём отличия? Отличия в том, что:

- Данный курс рассчитан на слушателей, которые **уже имеют начальные навыки** программирования для ядра Linux, и их целью является только совершенствование в этом предмете.

- Основной целью является не показ иллюстрирующих кодов, а формулирование задач для самостоятельной проработки, и последующее обсуждение их решений

Материалы данной книги (сам текст, сопутствующие его примеры, файлы содержащие эти примеры), как и предмет её рассмотрения — задумывались и являются свободно распространяемыми. На них автором накладываются условия свободной лицензии (http://legalfoto.ru/licenzii/) **Creative Commons Attribution ShareAlike**: допускается копирование, коммерческое использование произведения, создание его производных при чётком указании источника, но при том единственном ограничении, что при использовании или переработке разрешается применять результат **только на условиях аналогичной лицензии**.

Требуемый начальный уровень

Курс рассчитана на **опытных** разработчиков системного программного обеспечения. Предполагается некоторый минимальный опыт в программировании для ядра Linux (модули, драйвера), на уровне компиляции, сборки, и использования таких модулей в системе.

Совершенно естественно, что от читателя требуется квалифицированное знание языка C — единственного необходимого и достаточного языка системного программирования (из числа компилирующих) в Linux (хоть в пространстве ядра, хоть в пользовательском пространстве).

Естественно, я предполагаю, что вы «на дружеской ноге» с основными UNIX/POSIX консольными утилитами, такими, как: `ls`, `rm`, `grep`, `tar` и другие. Это необходимо для тестирования и организации работы с проектируемыми компонентами. В Linux используются, наибольшим образом, GNU (FSF) реализации таких утилит, которые набором опций часто отличаются (чаще в сторону расширения) от предписаний стандарта POSIX, и отличаются, порой, от своих собратьев в других операционных системах (Solaris, QNX, ...). Но эти отличия не столь значительны, я думаю, чтобы вызвать какие-либо затруднения.

Последовательность изложения

Представленный порядок тем (глав) вызывал несколько раз обсуждения и возражения: сначала обсуждается общая структура драйверов устройств и сетевого стека, а только затем — реализация внутренних механизмов ядра, используемые в коде этих драйверов.

Прежде всего, любой раздел из области программирования не описывается в линейном изложении, и требует рекурсивного изложения, когда приходится оперировать терминами и понятиями, которые будут определены только позже. Это хорошо известно из теории языков программирования.

Во-вторых, автор предпочитает рассмотрение сверху-вниз альтернативному снизу-вверх, что также много обсуждалось в теории программирования и не нуждается в разъяснениях.

Далее, представленный материал, как было сказано, ориентирован на достаточно опытного практика, которому встречаемые ним механизмы низкого уровня будут интуитивно понятны и без дополнительных объяснений, хотя бы по аналогиям из POSIX, которым они достаточно близки.

И, наконец, самое главное: целевая направленность текста на сопровождение упражнений и задач для самостоятельной проработки слушателей. А без возможности создания работающих и тестируемых модулей невозможно написание кода, отрабатывающего какой-то из механизмов. Таким образом, прежде рассмотрения общей структуры программ модулей, любое детальное описание механизмов становится голословным рассказом.

Если некоторые термины и механизмы ядра, используемые в примерах, покажутся вам совершенно незнакомыми, рассмотрите их забежав вперёд, в части, касающейся обсуждения внутренних программных механизмов ядра.

Каждый раздел завершается формулировками нескольких задач относительно материала раздела. Задачами они названы достаточно условно, диапазон их широк: от относительно несложных **вопросов**, до **задач**, требующих создания законченных мини-проектов. Но даже вопросы, которые кажутся элементарными, отбирались из числа тех, которые «с подковыркой», и требуют обстоятельного **понимания** материала. Задачи, естественно, предназначены для самостоятельной проработки. На них не даются ответы в тексте. Но по всем задачам, требующих написания программного кода (а таких подавляющее большинство), показаны варианты решения «от автора» в архиве примеров, сопровождающем текст.

Соглашения принятые в тексте

Этот текст ориентировался, в первую очередь, не столько для чтения или подробных разъяснений, сколько в качестве справочника при решении практических задач. Это накладывает отпечаток на текст:

- Перечисления альтернатив, например, символьных констант для выражения значений некоторого параметра, в случае их многочисленности приводится **не полностью** — разъясняются только наиболее употребимые, акцент делается на понимании (всё остальное может быть найдено в заголовочных файлах ядра — вместо путанного перечисления 30-ти альтернатив лучше указать две, использующиеся в 95% случаев).
- Обязательно указываются те места для поиска (имена заголовочных файлов, файлы описаний) где можно (попытаться) разыскать ту же информацию, но актуальную в требуемой вами версии ядра; это связано с постоянными изменениями, происходящими от версии к версии.
- Обсуждаемые задачи — примеры законченные, исполнимые и проверенные. Примеры оформлены как небольшие проекты, которые собираются достаточно тривиально (многие проекты содержат файл `*.hist` — это протокол с терминала тех действий, которые выполнялись по сборке и тестированию данного проекта, это многое разъясняет: зачем сделан этот пример и что он должен показать).

Для ясности чтения текста, он размечен шрифтами по функциональному назначению, в принципе, такая разметка уже употребляется практически повсеместно. Для выделения фрагментов текста по назначению используется разметка:

- Некоторые ключевые понятия и термины в тексте, на которые нужно обратить особое внимание, будут выделены **жирным шрифтом**.
- Таким же `моноширинным шрифтом` (прямо в тексте) будут выделяться написание: имён команд, программ, файлов ... — всех тех терминов, которые должны оставаться неизменяемыми, например: `/proc, mkdir, ./myprog,` ...
- Ввод пользователя в консольных командах (сами выполняемые команды, или ответы в диалоге), кроме того, выделены **`жирным моноширинным шрифтом`**, чтобы отличать их от ответного вывода системы

в диалогах.

- Имена файлов программных листингов (как они придаются в архиве примеров) записаны 1-й строкой, предшествующей листингу, и выделены ***жирным подчёркнутым курсивом***.

В показанных примерах команд изображение символа приглашения, без дополнительных напоминаний, будет показывать уровень прав, которые требует данная команда: `#` - будет означать команду с правами `root`, `$` - команду в правами ординарного пользователя.

В задачах, перечисляемых после каждого раздела, некоторые из них отмечены знаком `(*)`. Так отмечены задачи, как это часто и принято, повышенной сложности. Это не задачи, требующие большой трудоёмкости для своего решения (а таких немало), а требующие нестандартных решений и находок, выходящих за рамки материала, освещаемого в тексте.

Код примеров и замеченные опечатки

Все листинги, приводимые в качестве примеров, были опробованы и испытаны. Архивы (вида `*.tgz`), содержащие листинги, представлены на едином общедоступном ресурсе, там же, где и сам текст рукописи. В тексте, где обсуждаются коды примеров, везде, по возможности, будет указано в скобках имя архива в этом источнике, например: (архив `export.tgz`, или это может быть каталог `export`). В зависимости от того, в каком виде (свёрнутом или развёрнутом) вам достались файлы примеров, то, что названо по тексту «архив», может быть представлен на самом деле каталогом, содержащим файлы этого примера, поэтому, относительно «целеуказания» примеров, термины **архив** и **каталог** будут употребляться как синонимы. Один и тот же архив может упоминаться несколько раз в самых разных главах описания — это не ошибка: в одной теме он может иллюстрировать структуру, в другой — конкретные механизмы ввода/вывода, в третьей — связывание внешних имён объектных файлов и так далее. Листинги, поэтому, специально не нумерованы (нумерация могла бы «сползти» при правках), но указаны архивы, где их можно найти в полном виде. В архивах примеров **могут** содержаться файлы вида `*.hist` (расширение от history) — это текстовые файлы протоколов выполнения примеров: порядок запуска приложений, и какие результаты следует ожидать, и на что обратить внимание..., в тех случаях, когда сборка (`make`) примеров требует каких-то специальных приёмов, протокол сборки также может быть тоже показан в этом файле.

И ещё одно немаловажное замечание относительно программных кодов. Я отлаживал и проверял эти примеры на не менее полтора десятка различных инсталляций Linux (реальных и виртуальных, 32 и 64 разрядные архитектуры, установленных из различных дистрибутивов, и разных семейств дистрибутивов: Fedora, CentOS, Debian, Ubuntu, ...). И в некоторых случаях **давно работающий** пример перестаёт компилироваться с совершенно дикими сообщениями вида:

```
/home/olej/Kexamples.BOOK/IRQ/mod_workqueue.c:27:3: ошибка: неявная декларация функции ‘kfree’
/home/olej/Kexamples.BOOK/IRQ/mod_workqueue.c:37:7: ошибка: неявная декларация функции ‘kmalloc’
```

Пусть вас это не смущает! В разных инсталляциях может нарушаться порядок взаимных ссылок заголовочных файлов, и какой-то из заголовочных файлов выпадает из определений. В таких случаях вам остаётся только разыскать недостающий файл (а по тексту для всех групп API я указываю файлы их определений), и включить его явным указанием. Вот, например, показанные выше сообщения об ошибках устраняются включением строки:

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

Но такое же включение для других инсталляций (дистрибутивов и версий) будет избыточным (но не навредит).

Конечно, при самой тщательной выверке и вычитке, не исключены недосмотры и опечатки в таком объёмном тексте, могут проскочить мало внятные стилистические обороты и подобное. О замеченных таких дефектах я прошу сообщать по электронной почте olej@front.ru, и я был бы признателен за любые указанные недостатки книги, замеченные ошибки, или высказанные пожелания по её доработке.

Замечания о дистрибутивах и версиях ядра

Этот текст, в нынешнем его виде, накапливался, подготавливался, писался, изменялся и дополнялся на протяжении достаточно продолжительного времени (более 3-х лет). К нему отбирались, писались, отрабатывались и совершенствовались примеры реализации кода. За всё это время «в обиходе» сменились версии ядра от 2.6.32 до 3.17 (на момент написания). Примеры и команды, показываемые в тексте, отрабатывались и проверялись на всём этом диапазоне, и, в дополнение, на нескольких различных дистрибутивах Linux: Fedora, CentOS, Debian, Ubuntu. Помимо этого, в качестве основы для некоторых примеров брался код из реальных проектов автора прошлых лет ещё в системе CentOS 5.2 (для проверки переносимости, стабильности):

```
$ uname -r
2.6.18-92.el5
```

Кроме того, разнообразие вносит и то, что примеры отрабатывались: а). на 32-бит и 64-бит инсталляциях и б). на реальных инсталляциях и нескольких виртуальных машинах под управлением Oracle VirtualBox. Как легко видеть, для проверок и сравнений были использованы варианты по возможности широкого спектра различий. Хотя выверить **все** примеры и на **всех** вариантах установки — это за гранью человеческих возможностей. Ещё на иных дистрибутивах Linux могут быть какие-то отличия, особенно в путевых именах файлов, но они не должны быть особо существенными.

К версии (ядра) нужно подходить с очень большой осторожностью: ядро — это не пользовательский уровень, и разработчики не особенно обременяют себя ограничениями совместимости снизу вверх (в отличие от пользовательских API, POSIX). Источники информации и обсуждения, в множестве разбросанные по Интернет, чаще всего относятся к устаревшим версиям ядра, и абсолютно не соответствуют текущему положению дел. Очень показательно это проявилось, например, в отношении макросов подсчёта ссылок использования модулей, которые до версий 2.4.X использовались: `MOD_INC_USE_COUNT` и `MOD_DEC_USE_COUNT`, но их нет в 2.6.X и далее, но они продолжают фигурировать во множестве описаний.

Использованные источники информации

Литература по программированию модулей ядра Linux хоть и малочисленна, но она есть. В конце книги приведено достаточно много обстоятельных источников информации по этому предмету. Основные использовавшиеся источники информации (которые показались мне самыми полезными) перечислены в конце текста, в разделе «Источники информации». В некоторых случаях это только указание выходных данных книг. Там где существуют изданные русскоязычные их переводы — я старался указать и переводы тоже. По некоторым источникам, авторы которых решили сделать их публично доступными, показаны ссылки на них в сети. Для статей, которые взяты из сети, я указываю URL и, по возможности, авторов публикации, но далеко не по всем материалам, разбросанным и кочующим по Интернет, удаётся установить их авторство.

1. Специфика программирования в ядре

Программирование в ядре имеет определённые отличия от программирования пользовательских приложений и накладывает определённые сложности и ограничения. И прежде чем переходить к конкретике программирования в ядре, разумно изучить те сложности, которые будут подстерегать и подготовиться к ним.

Различия в техниках программирования

1. В ядре **недоступны никакие библиотеки**, привычные из прикладного программирования, и известные как POSIX API. Причин на то много, и их обсуждение не входит в наши планы.

А как следствие, ядро оперирует со своим собственным набором API (kernel API), отличающимся от POSIX API (отличающихся набором функций, их наименованиями). Это видно на примере идентичных по смыслу, но различающихся вызовах `printk()` и `printf()`. И если мы и будем встречаться довольно часто с **якобы идентичными** функциями (`sprint(), strlen(), strcat()` и многие другие), то это только внешняя **видимость совпадения**. Эти функции реализуют ту же функциональность, но это **дубликатная** реализация: подобный код реализуется и размещается **в разных местах**, для POSIX API в составе библиотек, а для модулей — непосредственно в составе ядра.

Возьмём на заметку, что у этих двух эквивалентных реализаций будет и различная авторская (если можно так сказать) принадлежность, и время обновления. Реализация в составе библиотеки `libc.so`, изготавливается сообществом GNU/FSF в комплексе проекта компилятора GCC, и новая версия библиотеки (и её заголовочные файлы в `/usr/include`) устанавливаются, когда вы обновляете версию **компилятора**. А реализация версии той же функции в ядре по авторству принадлежит разработчикам ядра Linux, и будет обновляться когда вы обновляете **ядро**, будь то из репозитария используемого вами дистрибутива, или собирая его самостоятельно из исходных кодов. А эти обновления (**компилятора и ядра**), как понятно, являются не коррелированными и не синхронизированными. Это не очевидная и часто опускаемая из виду особенность!

Косвенным следствием из сказанного будет то, что программный код процесса и модуля в качестве каталогов для файлов определений (`.h`) по умолчанию (`#include <...>`) будут использовать совершенно разные источники: `/usr/lib/include` и `/lib/modules/`uname -r`/build/include`, соответственно. Но об этом мы поговорим подробно, когда будем детально разбирать варианты сборки модулей.

Побочным следствием дублирования реализации подобных функций может оказаться и то, что в некоторых деталях поведения будут различаться идентичные по наименованию функции API ядра и API POSIX.

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

Иногда очень помогает отслеживание аналогичных вызовов пространств пользователя и ядра, примеры только некоторых из них собраны в табл.1.

Таблица 1: примеры вызовов API POSIX и ядра, несущих эквивалентную функциональность.

API процессов (POSIX)	**API ядра**
`strcpy(), strncpy(), strcat(), strncat(), strcmp (), strncmp(), strchr (), strlen(), strnlen(), strstr(), strrchr()`	`strcpy (), strncpy (), strcat(), strncat(), strcmp(), strncmp(), strchr(), strlen(), strnlen(), strstr(), strrchr()`
`printf()`	`printk()`
`execl(), execlp(), execle(), execv(), execvp(), execve()`	`call_usermodehelper()`
`malloc(), calloc(), alloca()`	`kmalloc(), vmalloc()`
`kill(), sigqueue()`	`send_sig()`
`open(), lseek(), read(), write(), close()`	`filp_open(), kernel_read(), kernel_write(), vfs_llseek(), vfs_read(), vfs_write(), filp_close()`
`atol(), sscanf()`	`simple_strtoul(), sscanf()`
`pthread_create()`	`kernel_thread()`
`pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_unlock()`	`rt_mutex_lock(), rt_mutex_trylock(), rt_mutex_unlock()`

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

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

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

```
#include <linux/version.h>
...
#if (LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 24))
...
#elif (LINUX_VERSION_CODE < KERNEL_VERSION(3, 17, 0))
...
#else
...
#endif
```

Вы не имеете права предполагать в какой версии ядра будет компилировать ваш модуль его пользователь!

5. **Уникальными ресурсами**, позволяющем изучить и сравнить исходный код ядра для различных **версий** ядра и аппаратных **платформ**, являются ресурсы построенные на базе проекта LXR, например, такие как (даю несколько альтернатив, потому что каждая из них может в какое-то время быть недоступной):

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

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

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

http://lxr.missinglinkelectronics.com/linux (missing link elektronics)

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

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

Отсутствие защиты памяти. Если обычная программа предпринимает попытку некорректного обращения к памяти, ядро аварийно завершит процесс, послав ему сигнал `SIGSEGV`. Если ядро предпримет попытку некорректного обращения к памяти, результаты могут быть менее контролируемыми. К тому же ядро не использует замещение страниц: каждый байт, используемый в ядре – это один байт физической памяти.

7. Как следствие высокой вероятности краха системы даже в итоге незначительной ошибки кода является то, что **каждый** тестовый запуск может требовать перезагрузки системы, а это **в разы** замедляет темп проекта, в сравнении с ориентирами пользовательского пространства.

Отличным решением (по крайней мере, когда не требуется какая-то аппаратная специфика) является решение вести разработку в среде **виртуальной машины** Linux, на начальном этапе проекта и до до 90% общего времени разработки. Для этого не очень пригодны развитые системы виртуализации типа XEN и подобные, но очень подходят достаточно простые гипервизоры Oracle VirtualBox, или QEMU и KVM (когда нужна архитектура отличная от x86).

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

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

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

Создание среды разработки

Первично установленная по умолчанию из пакетного дистрибутива система Linux для сборки ядра модулей ядра **непригодна**. В ней отсутствуют некоторые специфические компоненты, такие как хэдеры ядра и

другие подобные. Это и естественно, так как рядовому пользователю нет нужды собирать модули ядра, и незачем ему загружать файловую систему достаточно объёмными бесполезными данными. Но вам необходимо создать среду сборки, если вы не сделали этого ранее. Для этого нужно установить некоторые дополнительные пакеты из репозитария...

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

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

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

```
# apt-get install gcc make
Чтение списков пакетов… Готово
Построение дерева зависимостей
Чтение информации о состоянии… Готово
Будут установлены следующие дополнительные пакеты:
  binfmt-support binutils cpp-4.6 g++-4.6 gcc-4.6 gcc-4.6-base gcc-4.7 libc-dev-bin libc6-dev
  libffi-dev libitm1 libllvm3.0 libstdc++6-4.6-dev linux-libc-dev
Предлагаемые пакеты:
  binutils-doc gcc-4.6-locales g++-4.6-multilib gcc-4.6-doc libstdc++6-4.6-dbg gcc-multilib
  autoconf automake1.9 libtool flex bison gdb gcc-doc gcc-4.6-multilib libmudflap0-4.6-dev
  libgcc1-dbg libgomp1-dbg libquadmath0-dbg libmudflap0-dbg binutils-gold gcc-4.7-multilib
  libmudflap0-4.7-dev gcc-4.7-doc gcc-4.7-locales libitm1-dbg libcloog-ppl0 libppl-c2 libppl7
  glibc-doc libstdc++6-4.6-doc
...
$ gcc --version
gcc (Debian 4.7.2-5) 4.7.2
...
$ make --version
GNU Make 3.81
...
```

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

```
# yum install gcc make
...
```

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

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

```
$ yum list all kernel*
...
0 packages excluded due to repository protections
Установленные пакеты
kernel.i686                     3.12.7-300.fc20                 @fedora-updates/$releasever
kernel.i686                     3.12.10-300.fc20                @updates
kernel-headers.i686             3.12.10-300.fc20                @updates
kernel-modules-extra.i686       3.12.7-300.fc20                 @fedora-updates/$releasever
kernel-modules-extra.i686       3.12.10-300.fc20                @updates
Доступные пакеты
kernel.i686                     3.13.6-200.fc20                 updates
```

```
...
kernel-devel.i686                 3.13.6-200.fc20                 updates
...
kernel-headers.i686               3.13.6-200.fc20                 updates
kernel-modules-extra.i686         3.13.6-200.fc20                 updates
...
```

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

```
$ sudo yum update kernel*
...
$ sudo yum install kernel-devel.i686
...
Объем загрузки: 8.5 M
Объем изменений: 31 M
Is this ok [y/d/N]: y
...
```

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

```
$ ls /usr/lib/modules
3.12.10-300.fc20.i686  3.12.7-300.fc20.i686  3.13.6-200.fc20.i686
$ tree -L 2 /usr/src/kernels/
/usr/src/kernels/
└── 3.13.6-200.fc20.i686
    ├── arch
    ├── block
    ├── crypto
    ├── drivers
    ├── firmware
    ├── fs
    ├── include
    ├── init
    ├── ipc
    ├── Kconfig
    ├── kernel
    ├── lib
    ├── Makefile
    ├── mm
    ├── Module.symvers
    ├── net
    ├── samples
    ├── scripts
    ├── security
    ├── sound
    ├── System.map
    ├── tools
    ├── usr
    ├── virt
    └── vmlinux.id

21 directories, 5 files
```

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

```
$ cat /etc/debian_version
7.2
```

```
$ aptitude show linux-headers
Нет в наличии или подходящей версии для linux-headers
Пакет: linux-headers
Состояние: не реальный пакет
Предоставляется: linux-headers-3.2.0-4-486, linux-headers-3.2.0-4-686-pae,
linux-headers-3.2.0-4-amd64, linux-headers-3.2.0-4-rt-686-pae, linux-headers-486,
linux-headers-686-pae, linux-headers-amd64, linux-headers-rt-686-pae
...
$ sudo aptitude install linux-headers-3.2.0-4-486
...
Настраивается пакет linux-headers-3.2.0-4-486 (3.2.51-1) …
$ ls /lib/modules/3.2.0-4-486 -l
итого 3000
lrwxrwxrwx 1 root root     34 Сен 20 17:26 build -> /usr/src/linux-headers-3.2.0-4-486
drwxr-xr-x 9 root root   4096 Окт 13 19:38 kernel
-rw-r--r-- 1 root root 723786 Окт 13 19:51 modules.alias
-rw-r--r-- 1 root root 705194 Окт 13 19:51 modules.alias.bin
-rw-r--r-- 1 root root   2954 Сен 20 17:26 modules.builtin
-rw-r--r-- 1 root root   3960 Окт 13 19:51 modules.builtin.bin
-rw-r--r-- 1 root root 353853 Окт 13 19:51 modules.dep
-rw-r--r-- 1 root root 491462 Окт 13 19:51 modules.dep.bin
-rw-r--r-- 1 root root    325 Окт 13 19:51 modules.devname
-rw-r--r-- 1 root root 118244 Сен 20 17:26 modules.order
-rw-r--r-- 1 root root    131 Окт 13 19:51 modules.softdep
-rw-r--r-- 1 root root 286536 Окт 13 19:51 modules.symbols
-rw-r--r-- 1 root root 364265 Окт 13 19:51 modules.symbols.bin
lrwxrwxrwx 1 root root     37 Сен 20 17:26 source -> /usr/src/linux-headers-3.2.0-4-common
```

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

```
$ ls -d -w 100 /lib/modules/`uname -r`/build/*
/lib/modules/3.13.6-200.fc20.i686/build/arch
/lib/modules/3.13.6-200.fc20.i686/build/block
/lib/modules/3.13.6-200.fc20.i686/build/crypto
/lib/modules/3.13.6-200.fc20.i686/build/drivers
/lib/modules/3.13.6-200.fc20.i686/build/firmware
/lib/modules/3.13.6-200.fc20.i686/build/fs
/lib/modules/3.13.6-200.fc20.i686/build/include
/lib/modules/3.13.6-200.fc20.i686/build/init
/lib/modules/3.13.6-200.fc20.i686/build/ipc
/lib/modules/3.13.6-200.fc20.i686/build/Kconfig
/lib/modules/3.13.6-200.fc20.i686/build/kernel
/lib/modules/3.13.6-200.fc20.i686/build/lib
/lib/modules/3.13.6-200.fc20.i686/build/Makefile
/lib/modules/3.13.6-200.fc20.i686/build/mm
/lib/modules/3.13.6-200.fc20.i686/build/Module.symvers
/lib/modules/3.13.6-200.fc20.i686/build/net
/lib/modules/3.13.6-200.fc20.i686/build/samples
/lib/modules/3.13.6-200.fc20.i686/build/scripts
/lib/modules/3.13.6-200.fc20.i686/build/security
/lib/modules/3.13.6-200.fc20.i686/build/sound
/lib/modules/3.13.6-200.fc20.i686/build/System.map
/lib/modules/3.13.6-200.fc20.i686/build/tools
/lib/modules/3.13.6-200.fc20.i686/build/usr
/lib/modules/3.13.6-200.fc20.i686/build/virt
/lib/modules/3.13.6-200.fc20.i686/build/vmlinux.id
```

Задачи

1. Найдите и выпишите **как можно больше** (оцениваем по числу существенных позиций) отличий в программировании приложений пользователя и модулей ядра. В тексте названы далеко не все различия, который могут «выплыть» в программировании модулей ядра. Вернитесь к этому вопросу после проработки всего материала.

2. Проверьте в своей системе (и, возможно, архитектуре) сборку и загрузку простейшего модуля ядра (любого из приводимых в архиве примеров). Это необходимо для работы со всем последующим материалом.

2. Вспоминаем: архитектура, ядро, модули...

«Эти правила, язык и грамматика Игры, представляют собой некую разновидность высокоразвитого тайного языка, в котором участвуют самые разные науки и искусства …, и который способен выразить и соотнести содержание и выводы чуть ли не всех наук.»
Герман Гессе «Игра в бисер».

Монолитное ядро Linux

Исторически все операционные системы, начиная от самых ранних (или считая даже начиная считать от самых рудиментарных исполняющих систем, которые с большой натяжкой вообще можно назвать операционной системой) делятся на самом верхнем уровне на два класса, различающихся в принципе:

- монолитное ядро (исторически более ранний класс), называемые ещё: моноядро, макроядро; к этому классу, из числа самых известных, относятся, например (хронологически): OS/360, RSX-11M+, VAX-VMS, MS-DOS, Windows (все модификации), OS/2, Linux, все клоны BSD (FreeBSD, NetBSD, OpenBSD), Solaris — почти все широко звучащие имена операционных систем.
- микроядро (архитектура появившаяся позже), известный также как клиент-серверные операционные системы и системы с обменом сообщениями; к этому классу относятся, например: QNX, MINIX 3, HURD, ядро Darwin MacOS, семейство ядер L4.

В микроядерной архитектуре все услуги для прикладного приложения система (микроядро) обеспечивает отсылая сообщения (запросы) соответствующим сервисам (драйверам, серверам, ...) — другим **процессам**, которые, что самое важное, выполняются не в пространстве ядра (в пользовательском кольце защиты). В этом случае не возникает никаких проблем с динамической реконфигурацией системы и добавлением к ней новых функциональностей (например, драйверов проприетарных устройств) «на лету».

В макроядерной архитектуре все услуги для прикладного приложения выполняют отдельные ветки кода внутри ядра (в пространстве ядра). В большинстве ОС с монолитным ядром обслуживание запросов пользовательского пространства выполняется **последовательно**, на время выполнения системного вызова доступ к ядру блокируется (выполняется монопольно). В Linux, начиная с ядер 2.6.X, введена **многопоточность** в ядре, т.е. несколько запросов к ядру могут выполняться параллельно. Главное что это позволяет — эффективно использовать несколько процессоров в SMP-архитектурах (на многоядерных процессорах).

Расширение функциональности кода ядра

До некоторого времени в развитии такой системы, и так было и в ранних версиях ядра Linux, всякое расширение функциональности достигалось пересборкой (перекомпиляцией) ядра. Для системы промышленного уровня это недопустимо. Поэтому, рано или поздно, любая монолитная операционная система начинает включать в себя ту или иную технологию динамической реконфигурации (что сразу же открывает дыру в её безопасности и устойчивости). Для Linux это — технология модулей ядра, которая появляется начиная с ядра 0.99 (1992г.) благодаря Питеру Мак-Дональду (Peter MacDonald).

Модульность в ядре решает многие проблемы, например, сохраняя объём ядра в памяти в некоторых разумных пределах при требовании возможности поддержки сотен и тысяч различных видов оборудования. Но нас будет интересовать модульность только в контексте **возможности расширения функциональности** ядра собственным кодом.

Из сказанного понятно, что расширение функциональности ядра может достигаться 2-мя различными способами:

1. **Статически.** Мы вносим собственные изменения (возможно, техникой наложения патчей) в основное дерево исходных кодов ядра Linux, после чего производится пересборка ядра. В итоге мы получаем некоторую проприетарную модификацию ядра. Преимуществом этого способа является то, что при этом в добавляемом коде доступны все имена объектов ядра (вызовы функций, экземпляры данных). Таким способом сборщики (мантейнеры) дистрибутивных образов патчат официальное (ванильное) ядро, включая туда дополнения, которые они находят нужными — ядро практически любого пакетного дистрибутива отличается от официального.

2. **Динамически.** Собственный код оформляется в формате динамически подгружаемого модуля. При выполнении системы модуль динамически загружается и становится **неотъемлемой частью** кода ядра. Всё связывание кода модуля производится по абсолютным адресам имён в ядре, поэтому модуль может быть собран только под конкретную версию и модификацию ядра (о чём будет подробно далее). Ограничением этого способа является то, что коду модуля доступны не все имена (объекты) ядра, а только те, которые явно объявлены как **экспортируемые** (число экспортируемых ядром имён составляет не более 10% от общего числа имён). Таким способом предоставляются драйверы самого разнообразного оборудования, как включённого в состав дистрибутива, так и от сторонних производителей.

Траектория системного вызова

Основным предназначением ядра всякой операционной системы, вне всякого сомнения, является обслуживание **системных вызовов** из выполняющихся в системе процессов (операционная система занимается, скажем 99.999% своего времени жизни, и только на оставшуюся часть приходится вся остальная экзотика, которой и посвящена эта книга: обработка прерываний, обслуживание таймеров, диспетчеризация потоков и подобные «мелочи»). Поэтому вопросы взаимосвязей и взаимодействий в операционной системе всегда нужно начинать с отчётливого понимания той цепочки, по которой проходит системный вызов.

В любой операционной системе системный вызов (запрос обслуживания со стороны системы) выполняется некоторой процессорной инструкцией прерывающей последовательное выполнение команд, и передающий управление коду режима супервизора. Это обычно некоторая команда программного прерывания, в зависимости от архитектуры процессора исторически это были команды с мнемониками подобными: `svc`, `emt`, `trap`, `int` и подобными. Если для конкретики проследить архитектуру Intel x86, то это традиционно команда программного прерывания с различным вектором, интересно сравнить, как это делают самые разнородные системы:

	Операционная система				
	MS-DOS	Windows	Linux	QNX	MINIX 3
Дескриптор прерывания для системного вызова	21h	2Eh	80h	21h	21h

Я специально добавил в таблицу две микроядерные операционные системы, которые принципиально по-другому строят обработку системных запросов: основной тип запроса обслуживания здесь — это требование отправки синхронного сообщения микроядра другому компоненту пользовательского пространства (драйверу, серверу). Но даже эта отличная модель только скрывает за фасадом то, что выполнение системных запросов, например, в QNX: `MsgSend()` или `MsgReply()` - ничего более на «аппаратном языке», в конечном итоге, чем процессорная команда `int 21h` с соответственно заполненными регистрами-параметрами.

Примечание: Начиная с некоторого времени (утверждается, что это примерно относится к началу 2008 года, или к времени версии Windows XP Service Pack 2) многие операционные системы (Windows, Linux) при выполнении системного вызова от использования программного прерывания `int` перешли к реализации системного вызова (и возврата из него) через новые

команды процессора `sysenter` (`sysexit`). Это было связано с заметной потерей производительности Pentium IV при классическом способе системного вызова, и желанием из коммерческих побуждений эту производительность восстановить любой ценой. Но принципиально нового ничего не произошло: ключевые параметры перехода (CS, SP, IP) теперь загружаются не из памяти, а из специальных внутренних регистров MSR (Model Specific Registers) с предопределёнными (0x174, 0x175, 0x176) номерами (из большого их общего числа), куда предварительно эти значения записываются, опять же, специальной новой командой процессора `wmsr`... В деталях это громоздко, реализационно — производительно, а по сущности происходит то, что назвали: «вектор прерывания теперь забит в железо, и процессор помогает нам быстрее перейти с одного уровня привилегий на другой».

Другие процессорные платформы, множество которых поддерживает Linux, используют некоторый подобный механизм перехода в привилегированный режим (режим супервизора), например, для сравнения:

- PowerPC обладает специальной процессорной инструкцией `sc` (system call), регистр `r3` загружается номером системного вызова, а параметры загружаются последовательно в регистры от `r4` по `r8`;

- ARM64 платформа также имеет специальную инструкцию `syscall` для осуществления системного вызова, номер системного вызова загружается в `raw` регистр, а параметры в `rdi, rsi, rdx, r10, r8` и `r9`;

Этих примеров достаточно, чтобы представить, что на любой интересующей вас платформа картина сохраняется качественно **одинаковая**.

Библиотечный и системный вызов из процесса

Теперь мы готовы коротко рассмотреть прохождение системного вызова в Linux (будем основываться на классической реализации через команды `int 80h / iret`, потому что реализация через `sysenter` / `sysexit` ничего принципиально нового не вносит).

Начнём рассмотрение с того как любой прикладной процесс запрашивает «библиотечную» услугу... Прикладной процесс вызывает требуемые ему услуги посредством библиотечного вызова ко множеству библиотек а). `*.so` — динамического связывания, или б). `*.a` — статического связывания. Примером такой библиотеки является стандартная C-библиотека (DLL — Dynamic Linked Library):

```
$ ls -l /lib/libc.*
lrwxrwxrwx 1 root root 14 Map 13  2010 /lib/libc.so.6 -> libc-2.11.1.so
$ ls -l /lib/libc-*.*
-rwxr-xr-x 1 root root 2403884 Янв  4  2010 /lib/libc-2.11.1.so
```

Часть (значительная) вызовов обслуживается непосредственно **внутри** библиотеки, вообще не требуя никакого вмешательства ядра, пример тому: `sprintf()` (или все строковые POSIX функции вида `str*()`). Другая часть потребует дальнейшего обслуживания со стороны ядра системы, например, вызов `printf()` (предельно близкий синтаксически к `sprintf()`). Тем не менее, **все** такие вызовы API классифицируются как **библиотечные вызовы**. Linux чётко регламентирует группы вызовов, относя библиотечные API к **секции 3** руководств `man`. Хорошим примером тому есть целая группа функций для запуска дочернего процесса `execl(), execlp(), execle(), execv(), execvp()`:

```
$ man 3 exec
NAME
       execl, execlp, execle, execv, execvp - execute a file
 SYNOPSIS
       #include <unistd.h>
...
```

Хотя ни один из всех этих **библиотечных** вызовов не запускает никаким образом дочерний процесс, а ретранслируют вызов к единственному **системному** вызову `execve()` :

```
$ man 2 execve
...
```

Описания системных вызовов (в отличие от библиотечных) отнесены к **секции 2** руководств `man`.

Системные вызовы далее преобразовываются в вызов ядра функцией `syscall()`, 1-м параметром которого будет номер требуемого системного вызова, например `NR_execve`. Для конкретности, ещё один пример: вызов `printf(string)`, где: `char *string` — будет трансфороваться в `write(1, string, strlen(string))`, который далее — в `sys_call(__NR_write,...)`, и ещё далее — в `int 0x80` (полный код такого примера будет показан страницей ниже).

Детально о самом `syscall()` можно посмотреть :

```
$ man syscall
ИМЯ
     syscall - непрямой системный вызов
     ОБЗОР
          #include <sys/syscall.h>
          #include <unistd.h>
          int syscall(int number, ...)
     ОПИСАНИЕ
          syscall() выполняет системный вызов, номер которого задаётся значением number и с
          заданными аргументами. Символьные константы для системных вызовов можно найти в
          заголовочном файле ⟨sys/syscall.h⟩.
...
```

Образцы констант некоторых хорошо известных системных вызовов (начало таблицы, в качестве примера):

```
$ head -n20 /usr/include/asm/unistd_32.h
...
#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
...
```

Системные вызовы `syscall()` в Linux **на процессоре x86** выполняются через прерывание `int 0x80`. Соглашение о **системных** вызовах в Linux отличается от общепринятого в UNIX (через стек) и соответствует соглашению «fastcall». Согласно ему, программа помещает в регистр **eax** номер системного вызова, входные аргументы размещаются в других регистрах процессора (таким образом, системному вызову может быть передано до 6 аргументов последовательно через регистры **`ebx, ecx, edx, esi, edi`** и **ebp**), после чего вызывается инструкция `int 0x80`. В тех относительно редких случаях, когда системному вызову необходимо передать **большее количество** аргументов (например, `mmap`), то они размещаются в структуре, адрес на которую передается в качестве первого аргумента (**ebx**). Результат возвращается в регистре **eax**, а стек вообще не используется. Системный вызов `syscall()`, попав в ядро, всегда попадает в таблицу `sys_call_table`, и далее переадресовывается по индексу (смещению) в этой таблице на величину 1-го параметра вызова `syscall()` - номера требуемого системного вызова.

В любой другой поддерживаемой Linux/GCC аппаратной платформе (из многих) результат будет аналогичный: системные вызов `syscall()` будет «доведен» до команды программного прерывания (вызова ядра), применяемой на данной платформе, команд: `emt`, `trap` или нечто подобное.

Конкретный **вид и размер** таблицы системных вызовов зависит от **процессорной архитектуры**, под которую компилируется ядро. Естественно, эта таблица определена в ассемблерной части кода ядра, но даже **имена** и структура файлов при этом отличаются. Сравним, для подтверждения этого, сравнительные определения некоторых (подобных между собой) системных вызовов (последних в таблице системных вызовов) для нескольких разных архитектур (в исходных кодах ядра 3.0):

Архитектура 32-бит Intel x86 (http://lxr.free-electrons.com/source/arch/?v=3.0) (самое окончание таблицы):

```
...
.long sys_prlimit64             /* 340 */
.long sys_name_to_handle_at
```

```
        .long sys_open_by_handle_at
        .long sys_clock_adjtime
        .long sys_syncfs
        .long sys_sendmmsg              /* 345 */
        .long sys_setns
```

То же окончание таблицы (http://lxr.free-electrons.com/source/arch/x86/include/asm/unistd_64.h?v=3.0) для близкой архитектуры AMD 64-бит:

```
...
#define __NR_syncfs                     306
__SYSCALL(__NR_syncfs, sys_syncfs)
#define __NR_sendmmsg                   307
__SYSCALL(__NR_sendmmsg, sys_sendmmsg)
#define __NR_setns                      308
__SYSCALL(__NR_setns, sys_setns)
```

И то же окончание таблицы (http://lxr.free-electrons.com/source/arch/arm/kernel/?v=3.0) для архитектуры ARM:

```
                ...
 /* 370 */      CALL(sys_name_to_handle_at)
                CALL(sys_open_by_handle_at)
                CALL(sys_clock_adjtime)
                CALL(sys_syncfs)
                CALL(sys_sendmmsg)
 /* 375 */      CALL(sys_setns)
```

Последний (по номеру) системный вызов **для этой версии ядра** (3.0) с именем `sys_setns`, имеет номер (а это и полный размер таблицы, и **число** системных вызовов в архитектуре), соответственно: 346 для 32-бит архитектуры Intel x86, 308 для 64-бит архитектуры AMD/Intel, и 375 для архитектуры ARM. И так мы можем проследить размер и состав таблицы системных вызовов (селектора) для всех архитектур, поддерживаемых Linux. Кстати, здесь же мы можем посмотреть какие архитектуры поддерживает Linux:

```
$ ls /usr/src/linux-2.6.37.3/arch
alpha  avr32     cris  h8300  m68k   microblaze  mn10300  powerpc  score  sparc  um   xtensa
arm    blackfin  frv   ia64   m32r   m68knommu   mips     parisc   s390   sh     tile   x86
$ ls /usr/src/linux-2.6.37.3/arch | wc -w
25
```

(Этот перечень изменяется между версиями ядра, и актуальное его состояние для различных ядер можно смотреть здесь: http://lxr.free-electrons.com/source/arch/).

Возьмите на заметку ещё и то, что для некоторых архитектур в этом списке имя — только родовое название, которое объединяет много частных технических реализаций, часто несовместимых друг с другом. Например, для ARM:

```
$ ls /usr/src/linux-2.6.37.3/arch/arm
boot        mach-bcmring    mach-integrator  mach-loki     mach-ns9xxx  mach-s3c2412  mach-sa1100    mach-versatile  plat-orion
common      mach-clps711x   mach-iop13xx     mach-lpc32xx  mach-nuc93x  mach-s3c2416  mach-shark     mach-vexpress   plat-pxa
configs     mach-cns3xxx    mach-iop32x      mach-mmp      mach-omap1   mach-s3c2440  mach-shmobile  mach-w90x900    plat-s3c24xx
include     mach-davinci    mach-iop33x      mach-msm      mach-omap2   mach-s3c2443  mach-spear3xx  Makefile        plat-s5p
Kconfig     mach-dove       mach-ixp2000     mach-mv78xx0  mach-orion5x mach-s3c24a0  mach-spear6xx  mm              plat-samsung
Kconfig.debug  mach-ebsa110 mach-ixp23xx     mach-mx25     mach-pnx4008 mach-s3c64xx  mach-stmp378x  nwfpe           plat-spear
Kconfig-nommu  mach-ep93xx  mach-ixp4xx      mach-mx3      mach-pxa     mach-s5p6442  mach-stmp37xx  oprofile        plat-stmp3xxx
kernel      mach-footbridge mach-kirkwood    mach-mx5      mach-realview  mach-s5p64x0 mach-tcc8k    plat-iop        plat-tcc
lib         mach-gemini     mach-ks8695      mach-mxc91231 mach-rpc      mach-s5pc100  mach-tegra    plat-mxc      plat-versatile
mach-aaec2000  mach-h720x   mach-l7200       mach-netx     mach-s3c2400  mach-s5pv210  mach-u300     plat-nomadik    tools
mach-at91   mach-imx        mach-lh7a40x     mach-nomadik  mach-s3c2410  mach-s5pv310  mach-ux500    plat-omap       vfp
```

Здесь мы обзорно взглянули на то **единственное** место в Linux, в котором выполнение «расслаивается» для разных аппаратных платформ, и собирается снова в единый код как только управление попадает на селекторную таблицу `sys_call_table`. Именно такая компактность фрагментов кода, зависящего от платформы, позволяет Linux с одинаковым успехом поддерживать десятки процессорных платформ.

Выполнение системного вызова

Но возвратимся к технике осуществления системного вызова. Рассмотрим пример прямой реализации системного вызова из пользовательского процесса на архитектуре x86 (архив `int80.tgz`), который многое проясняет. Первый пример этого архива (файл `mp.c`) демонстрирует **пользовательский** процесс, выполняющий последовательно системные вызовы, эквивалентные библиотечным: `getpid()`, `write()`, `mknod()`, причём `write()` именно на дискриптор 1, то есть `printf()`:

mp.c :

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <linux/kdev_t.h>
#include <sys/syscall.h>

int write_call( int fd, const char* str, int len ) {
   long __res;
   __asm__ volatile ( "int $0x80":
      "=a" (__res):"0"(__NR_write),"b"((long)(fd)),"c"((long)(str)),"d"((long)(len)) );
   return (int) __res;
}

void do_write( void ) {
   char *str = "эталонная строка для вывода!\n";
   int len = strlen( str ) + 1, n;
   printf( "string for write length = %d\n", len );
   n = write_call( 1, str, len );
   printf( "write return : %d\n", n );
}

int mknod_call( const char *pathname, mode_t mode, dev_t dev ) {
   long __res;
   __asm__ volatile ( "int $0x80":
      "=a" (__res):
      "a"(__NR_mknod),"b"((long)(pathname)),"c"((long)(mode)),"d"((long)(dev))
   );
   return (int) __res;
};

void do_mknod( void ) {
   char *nam = "ZZZ";
   int n = mknod_call( nam, S_IFCHR | S_IRUSR | S_IWUSR, MKDEV( 247, 0 ) );
   printf( "mknod return : %d\n", n );
}

int getpid_call( void ) {
   long __res;
   __asm__ volatile ( "int $0x80":"=a" (__res):"a"(__NR_getpid) );
   return (int) __res;
};

void do_getpid( void ) {
   int n = getpid_call();
   printf( "getpid return : %d\n", n );
}

int main( int argc, char *argv[] ) {
```

```
    do_getpid();
    do_write();
    do_mknod();
    return EXIT_SUCCESS;
};
```

Пример написан с использованием инлайновых ассемблерных вставок компилятора GCC, но пример так прост, легко читается и интуитивно понятен: последовательно загружаются регистры значениями из переменных C кода, и вызывается программное прерывание системного вызова (`int 80h`). Выполняем полученное приложение:

```
$ ./mp
getpid return : 14180
string for write length = 54
эталонная строка для вывода!
write return : 54
mknod return : -1
```

Всё достаточно пристойно, за исключением одного вызова `mknod()`, но здесь мы можем вспомнить, что одноимённая консольная команда требует прав `root`:

```
$ sudo ./mp
getpid return : 14182
string for write length = 54
эталонная строка для вывода!
write return : 54
mknod return : 0
$ ls -l ZZZ
crw------- 1 root root 247, 0 Дек 20 22:00 ZZZ
$ rm ZZZ
rm: удалить защищенный от записи знаковый специальный файл `ZZZ'? y
```

Мы успешно создали **именованное символьное устройство**, да ещё и не в каталоге `/dev`, где ему и место, а в текущем рабочем каталоге (чудить так чудить!). Не забудьте удалить это имя, что и показано последней командой.

Примечание: Этот пример, помимо прочего, наглядно показывает замечательным образом как обеспечивается единообразная работа операционной системы Linux на десятках самых разнородных аппаратных платформ, системный вызов — это то «узкое горлышко» передачи управления ядру, которое единственно и будет принципиально меняться от платформы к платформе.

Альтернативные реализации

Предыдущий пример показан только в качестве иллюстрации того, как осуществляется системный вызов Linux. Можно ли **записать** то же, но в более приемлемой реализации? Конечно да (показана только реализация функций, без обрамления):

mpsys.c :

```
void do_write( void ) {
   char *str = "эталонная строка для вывода!\n";
   int len = strlen( str ) + 1, n;
   printf( "string for write length = %d\n", len );
   n = syscall( __NR_write, 1, str, len );
   if( n >= 0 ) printf( "write return : %d\n", n );
   else printf( "write error : %m\n" );
}

void do_mknod( void ) {
   char *nam = "ZZZ";
   int n = syscall( __NR_mknod, nam, S_IFCHR | S_IRUSR | S_IWUSR, MKDEV( 247, 0 ) );
```

```
   if( n >= 0 ) printf( "mknod return : %d\n", n );
   else printf( "mknod error : %m\n" );
}

void do_getpid( void ) {
   int n = syscall( __NR_getpid );
   if( n >= 0 ) printf( "getpid return : %d\n", n );
   else printf( "getpid error : %m\n" );
}
```

Это абсолютно та же реализация, но через **непрямой системный вызов** : `syscall()`. Чем такая реализация много лучше предыдущей? Тем, в первую очередь, что такая запись **не зависит** от процессорной платформы. В инлайновых ассемблерных вставках запись и обозначения регистров, например, будет отличаться на разных платформах. Более того, они отличаются даже между Intel платформой i686 и IA-32 и AMD x86_64. Запись, использующая `syscall()`, будет корректно компилироваться в соответствии указанной целевой платформы.

Вот протокол выполнения этого приложения в 64-бит системе:

```
$ uname -a
Linux modules.localdomain 3.17.8-200.fc20.x86_64 #1 SMP Thu Jan 8 23:26:57 UTC 2015 x86_64 x86_64
x86_64 GNU/Linux
$ make mpsys
gcc -Wall     mpsys.c   -o mpsys
mpsys.c: В функции «do_write»:
mpsys.c:7:4: предупреждение: неявная декларация функции «syscall» [-Wimplicit-function-
declaration]
    n = syscall( __NR_write, 1, str, len );
    ^
$ ./mpsys
getpid return : 13896
string for write length = 54
эталонная строка для вывода!
write return : 54
mknod return : -1
$ sudo ./mpsys
[sudo] password for Olej:
getpid return : 13904
string for write length = 54
эталонная строка для вывода!
write return : 54
mknod return : 0
$ ls Z*
ZZZ
```

В чём недостаток такой записи? В том, что в ней полностью **исключён контроль** параметров вызова `syscall()` как по числу, так и по типам: прототип `syscall()` описан как функция с переменным числом параметров. Стандартная библиотека GCC вводит взаимно однозначное соответствие каждого `syscall()` библиотечному системному вызову. В такой терминологии эквивалентная реализация будет выглядеть так:

mplib.c :

```
void do_write( void ) {
   char *str = "эталонная строка для вывода!\n";
   int len = strlen( str ) + 1, n;
   printf( "string for write length = %d\n", len );
   n = write( 1, str, len );
   if( n >= 0 ) printf( "write return : %d\n", n );
   else printf( "write error : %m\n" );
}
```

```
void do_mknod( void ) {
   char *nam = "ZZZ";
   int n = mknod( nam, S_IFCHR | S_IRUSR | S_IWUSR, MKDEV( 247, 0 ) );
   if( n >= 0 ) printf( "mknod return : %d\n", n );
   else printf( "mknod error : %m\n" );
}

void do_getpid( void ) {
   int n = getpid();
   if( n >= 0 ) printf( "getpid return : %d\n", n );
   else printf( "getpid error : %m\n" );
}
```

Вот такое сравнительное рассмотрение системных вызовов между уровнями их представления в Linux является крайне поучительным для создания ясного представления о пути разрешения системных вызовов.

Отслеживание системного вызова в процессе

Где размещён код, являющийся интерфейсом к системному вызову Linux? Особенно, если выполняемый процесс скомпилирован из языка, отличного от C... Подобные особенности легко отследить, если собрать два сравнительных приложения (на C и C++) по принципу «проще не бывает» (архив `prog_sys_call.tgz`):

prog_c.c :

```
#include <stdio.h>

int main( int argc, char *argv[] ) {
   printf( "Hello, world!\n" );
   return 0;
};
```

prog_cc.cc :

```
#include <iostream>

using std::cout;
using std::endl;

int main( int argc, char *argv[] ) {
   cout << "Hello, world!" << endl;
   return 0;
};
```

Смотреть какие библиотеки (динамические) использует собранное приложение можем так:

```
$ ldd ./prog_c
        linux-gate.so.1 =>  (0x001de000)
        libc.so.6 => /lib/libc.so.6 (0x007ff000)
        /lib/ld-linux.so.2 (0x007dc000)
$ ldd ./prog_cc
        linux-gate.so.1 =>  (0x0048f000)
        libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x03927000)
        libm.so.6 => /lib/libm.so.6 (0x0098f000)
        libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x0054c000)
        libc.so.6 => /lib/libc.so.6 (0x007ff000)
        /lib/ld-linux.so.2 (0x007dc000)
```

Как легко видеть, эквивалент программы на языке C++ использует библиотеку API языка C (`libc.so.6`) в равной степени, как и программа, исходно написанная на C. Эта библиотека и содержит интерфейс к системным вызовам Linux.

Прослеживать, какие системные вызовы и в какой последовательности выполняет запущенный процесс, мы можем специальной формой команды запуска (через команду `strace`) такого процесса:

```
$ strace ./prog_c
...
write(1, "Hello, world!\n", 14Hello, world!
) = 14
...
$ strace ./prog_cc
...
write(1, "Hello, world!\n", 14Hello, world!
) = 14
...
```

Такой вывод объёмный и громоздкий, но он позволяет отследить в сложных случаях какие системные вызовы выполнял процесс. В показанном случае, пример показывает, помимо того, как две совершенно различные синтаксически конструкции (вызовы), из разных языков записи кода, разрешаются в один и тот же системный вызов `write()`.

Возможен ли системный вызов из модуля?

Такой вопрос мне нередко задавали мне в обсуждениях. Зададимся вопросом, предположительно достаточно безумным: а нельзя ли выполнить эти (а значит и другие) системные вызовы из кода модуля ядра, то есть изнутри ядра? Оформим практически тот же показанный выше код, только в форме модуля ядра... , но поскольку я хотел бы написать два почти идентичных примера (`mdu.c` и `mdc.c`), то этот общий для них код я помещу в общий включаемый файл (`syscall.h`):

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

int write_call( int fd, const char* str, int len ) {
   long __res;
   __asm__ volatile ( "int $0x80":
      "=a" (__res):"0"(__NR_write),"b"((long)(fd)),"c"((long)(str)),"d"((long)(len)) );
   return (int) __res;
}

void do_write( void ) {
   char *str = "=== эталонная строка для вывода!\n";
   int len = strlen( str ) + 1, n;
   printk( "=== string for write length = %d\n", len );
   n = write_call( 1, str, len );
   printk( "=== write return : %d\n", n );
}

int mknod_call( const char *pathname, mode_t mode, dev_t dev ) {
   long __res;
   __asm__ volatile ( "int $0x80":
      "=a" (__res):
      "a"(__NR_mknod),"b"((long)(pathname)),"c"((long)(mode)),"d"((long)(dev))
   );
   return (int) __res;
};

void do_mknod( void ) {
   char *nam = "ZZZ";
   int n = mknod_call( nam, S_IFCHR | S_IRUGO, MKDEV( 247, 0 ) );
   printk( KERN_INFO "=== mknod return : %d\n", n );
}

int getpid_call( void ) {
```

```
    long __res;
    __asm__ volatile ( "int $0x80":"=a" (__res):"a"(__NR_getpid) );
    return (int) __res;
};

void do_getpid( void ) {
    int n = getpid_call();
    printk( "=== getpid return : %d\n", n );
}
```

А вот и первый из модулей (`mdu.c`), который практически полностью повторяет код выполнявшегося выше процесса:

```
#include "syscall.h"

static int __init x80_init( void ) {
    do_write();
    do_mknod();
    do_getpid();
    return -1;
}

module_init( x80_init );
```

Примечание: Функция инициализации модуля `x80_init()` в этом примере сознательно возвращает ненулевой код возврата — такой модуль заведомо никогда не может быть установлен, но его функция инициализации будет выполнена, и будет выполнена в пространстве ядра. Это эквивалентно обычному выполнению пользовательской программы (от `main()` и далее...), но в адресном пространстве ядра. Фактически, можно с некоторой условностью считать, что таким образом мы не устанавливаем модуль в ядро, а просто выполняем некоторый процесс (свой код), но уже в адресном пространстве ядра. Такой трюк будет неоднократно использоваться далее, и там же, в вопросах отладки ядерного кода, будет обсуждаться подробнее.

Его загрузка:

```
$ sudo insmod mdu.ko
insmod: error inserting 'mdu.ko': -1 Operation not permitted
$ dmesg | tail -n30 | grep ===
=== string for write length = 58
=== write return : -14
=== mknod return : -14
=== getpid return : 14217
$ ps -A | grep 14217
$
```

В общем, всё совершенно ожидаемо (ошибки выполнения) ... кроме вызова `getpid()`, который навевает некоторые смутные подозрения, но об этом позже. Главная цель достигнута: мы видим тонкое различие между пользовательским процессом и ядром, состоящее в том, что при выполнении системного вызова из любого процесса, код обработчика системного вызова (в ядре!) должен копировать **данные** параметров вызова из адресного пространства процесса в пространство ядра, а после выполнения копировать **данные** результата обратно в адресного пространства процесса. А при попытке вызвать системный обработчик из контекста ядра (модуля), что мы только что сделали — нет адресного пространства процесса, нет такого процесса! Но ведь `getpid()` выполнился? И чей PID он показал? Да, выполнился, потому, что этот системный вызов не получает параметров и не копирует результатов (он возвращает значение в регистре). А возвратил он PID того процесса, в **контексте** которого выполнялся системный вызов, а это процесс `insmod`. Но всё таки системный вызов выполнен из модуля!

Перепишем (файл `mdc.c`) незначительно предыдущий пример:

```
#include <linux/uaccess.h>
#include "syscall.h"
```

```
static int __init x80_init( void ) {
   mm_segment_t fs = get_fs();
   set_fs( get_ds() );
   do_write();
   do_mknod();
   do_getpid();
   set_fs( fs );
   return -1;
}

module_init( x80_init );
```

(вызовы `set_fs()`, `get_ds()` - это вещи тонкие, это всего лишь смена признака сегмента данных, мы поговорим об этом как-то позже).

А теперь держитесь за стул! :

```
$ sudo insmod mdc.ko
=== эталонная строка для вывода!
insmod: error inserting 'mdc.ko': -1 Operation not permitted
$ dmesg | tail -n30 | grep ===
=== string for write length = 58
=== write return : 58
=== mknod return : 0
=== getpid return : 14248
 $ ls -l ZZZ
cr--r--r-- 1 root root 0, 63232 Дек 20 22:04 ZZZ
```

Вам говорили, что модуль не может выполнить `printf()` и осуществлять вывод на графический терминал? А как же та вопиющая строка, которая **предшествует** инсталляционному сообщению? На какой управляющий терминал произошёл вывод? Хороший вопрос... Конечно, на терминал запускающей программы `insmod`, и сделать подобное можно **только** из функции инициализации модуля. Но главное не это. Главное то, что в этом примере все системные вызовы успешно выполнились! А значит выполнится и **любой** системный вызов пользовательского пространства.

Мы ещё раз, детально и с примерами, вернёмся к вопросам программирования системных вызовов к концу нашего рассмотрения, не скоро, когда мы будем готовы рассмотреть возможности создания новых системных вызовов под собственные потребности.

Примечание: Теперь краткие замечания относительно структуры `mm_segment_t`, о которой было обещано выше. Это всего лишь такой трюк разработчиков ядра, и относящийся только к архитектуре x86 (хотя это и основная архитектура), и состоящий в том, что для **сегментов** памяти пространства ядра в поле границы сегментов в таблицах LDT или GDT записывается значение `__PAGE_OFFSET`, имеющее для **32-бит** значение 0xC0000000, что соответствует пределу пространства логических адресов до 3Gb. Сегменты же пространства ядра имеют в поле границы максимально возможное значение значение -1UL (и адреса пространства ядра имеют значения в диапазоне 0xC0000000 — 0xFFFFFFFF, 3Gb — 4Gb ... но это не имеет прямого отношения к текущему рассмотрению). Это поле границы сегмента используется **только** для **контроля** принадлежности **сегмента** памяти пространству пользователя при выполнении системных вызовов. Но этот контроль можно **отменить** как показано выше (а позже восстановить). Мы будем неоднократно использовать этот трюк, но не будем больше возвращаться к его объяснению. Проследить всю цепочку деталей этого механизма можно по заголовочным файлам:

```
<arch/x86/include/asm /thread_info.h>
struct thread_info {
...
        mm_segment_t            addr_limit;
...
<arch/x86/include/asm /processor.h>
...
typedef struct {
        unsigned long           seg;
```

```
} mm_segment_t;
...
#ifdef CONFIG_X86_32
/*
 * User space process size: 3GB (default).
 */
#define TASK_SIZE               PAGE_OFFSET
...
#else
/*
 * User space process size. 47bits minus one guard page.
 */
#define TASK_SIZE_MAX   ((1UL << 47) — PAGE_SIZE)
...
<arch/x86/include/asm /page_types.h >
...
#define PAGE_OFFSET             ((unsigned long)__PAGE_OFFSET)
...
<arch/x86/include/asm /page_32_types.h>
...
 * A __PAGE_OFFSET of 0xC0000000 means that the kernel has
 * a virtual address space of one gigabyte, which limits the
 * amount of physical memory you can use to about 950MB.
 *
#define __PAGE_OFFSET           _AC(CONFIG_PAGE_OFFSET, UL)
...
<linux/autoconf.h>
...
#define CONFIG_PAGE_OFFSET 0xC0000000
...
<arch/x86/include/asm /uaccess.h>
#define MAKE_MM_SEG(s)  ((mm_segment_t) { (s) })
#define KERNEL_DS       MAKE_MM_SEG(-1UL)
#define USER_DS         MAKE_MM_SEG(TASK_SIZE_MAX)
#define get_ds()        (KERNEL_DS)
#define get_fs()        (current_thread_info()->addr_limit)
#define set_fs(x)       (current_thread_info()->addr_limit = (x))
```

Задачи

1. Попытайтесь сформулировать все (как можно больше) преимущества и недостатки (табличка с отметками «+» и «-») микроядерной архитектуры (например QNX) по сравнению с моноядерной (например Linux или Solaris).

2. Для своей процессорной архитектуры (или хотя бы для AMD x86_64, за неимением другой) напишите выполнение системного вызова непосредственным **вызовом программного прерывания**. Покажите все слои, через которые проходит системный вызов в этой архитектуре.

3. В тексте ранее показано, что компилированный C++ код, например, помимо того, что он использует свою библиотеку, использует стандартную библиотеку C/POSIX как **интерфейс системных вызовов** к ядру. Рассмотрите и **покажите** (выполнением команд Linux), как эту же библиотеку используют:

    - Виртуальная машина Java (JVM) ... проделайте это, по возможности, для различных реализаций JVM: а). Oracle JDK, б). Open JDK, стоящем в Linux по умолчанию;

- То же для интерпретатора Python (python или python3).

4. (*) Используют ли какие либо (из известных вам) языковых технологий программирования интерфейс к системным вызовам (к операционной системе), не использующий стандартную библиотеку C `libc.so`. Если да, то подтвердите это не голословными утверждениями, а командами Linux.

5. Может ли при написании модулей ядра Linux использоваться другой язык программирования, чем C. Может ли для этих целей использоваться другой компилятор с C, отличный от GCC, например Clang.

3. Техника модулей ядра

Все мы умеем, и имеем больший или меньший опыт написания программ в Linux[1], которые все, при всём их многообразии, имеют абсолютно идентичную единую структуру:

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

Такую структуру в коде будут неизменно иметь все приложения-программы, будь то тривиальная показанная «Hello, world!», или «навороченная» среда разработки IDE Eclipse[2]. Это — в подавляющем большинстве встречаемый случай: пользовательское приложение начинающееся с `main()` и завершающееся по `exit()`.

Ещё один встречающийся (но гораздо реже) в UNIX случай — это демоны: программы, стартующие с `main()`, но никогда не завершающие своей работы (чаще всего это сервера различных служб). В этом случае для того, чтобы стать сервером, всё тот же пользовательский процесс должен выполнить некоторую фиксированную последовательность действий [19][20], называемую демонизацией, который состоит в том, чтобы (опуская некоторые детали для упрощения):

- создать свой собственный клон вызовом `fork()` и завершить родительский процесс;
- создать новую сессию вызовом `setsid()`, при этом процесс становится лидером сессии и открепляется (теряет связь) от управляющего терминала;
- позакрывать все ненужные файловые дескрипторы (унаследованные).

Но и в этом случае процесс выполняется в пользовательском адресном пространстве (отдельном для **каждого** процесса) со всеми ограничениями пользовательского режима: запрет на использование привилегированных команд, невозможность обработки прерываний, запрет (без особых ухищрений) операций ввода-вывода и многих других тонких деталей.

Возникает вопрос: а может ли пользователь написать и выполнить собственный код, выполняющийся в режиме супервизора, а, значит, имеющий полномочия расширять (или даже изменять) функциональность ядра Linux? Да, может! И эта техника программирования называется программированием модулей ядра. И именно она позволяет, в частности, создавать драйверы нестандартного оборудования[3].

Примечание: Как мы будем неоднократно видеть далее, установка (запуск) модуля выполняется посредством специальных команд установки, например, командой:

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

После чего в модуле начинает выполняться функция инициализации. Возникает вопрос: а можно ли (при необходимости) создать пользовательское приложение, стартующее, как обычно, с точки `main()`, а далее присваивающее себе требуемые привилегии, и выполняющееся в супервизорном режиме (в пространстве ядра)? Да, можно! Для этого изучите исходный код

1 Замечание здесь о Linux не есть оговоркой, а означает, что вышесказанное верно только для операционных систем, языком программирования для которых (самих систем) является классический язык C, точнее говорить даже в этом контексте не о системе Linux, а о любых UNIX-like или POSIX системах.

2 В качестве примера Eclipse указан также не случайно: а). это один из инструментов, который может эффективно использоваться в разработках модулей, и особенно если речь зайдёт о клоне Android на базе ядра Linux, и б). даже несмотря на то, что сам Eclipse писан на Java, а вовсе не на C - всё равно структура приложения сохранится, так как с вызова `main()` будет начинаться выполнение интерпретатора JVM, который далее будет выполнять Java байт-код. То же относится и к приложениям, написанным на таких интерпретирующих языках как Perl, Python, или даже на языке командного интерпретатора shell: точно ту же структуру приложения будет воспроизводить само интерпретирующее приложение, которое будет загружаться прежде интерпретируемого кода.

3 Это (написание драйверов) - самое важное, но не единственное предназначение модулей в Linux: «всякий драйвер является модулем, но не всякий модуль является драйвером».

утилиты `insmod` (а Linux — система с абсолютно открытым кодом всех компонент и подсистем), а утилита эта является ничем более, чем заурядным пользовательским приложением, выполните в своём коде те манипуляции с привилегиями, которые проделывает `insmod`, и вы получите желаемое приложение. Естественно, что всё это потребует от приложения привилегий `root` при запуске, но это всё то же минимальное требование, которое вообще обязательно при любой работе с модулями ядра.

Простейший модуль для анализа

> *««Hello, world!»— программа, результатом работы которой является вывод на экран или иное устройство фразы «Hello, world!»... Обычно это первый пример программы...»*
> *Википедия: http://ru.wikipedia.org/wiki/Hello,_World!*

Для начала знакомства с техникой написания модулей ядра Linux проще не вдаваться в пространные объяснения, но создать простейший модуль (код такого модуля интуитивно понятен всякому программисту), собрать его, наблюдать исполнение и т.д. Совершенно не важно, что конкретно будет выполнять этот модуль. И только потом, ознакомившись с некоторыми основополагающими принципами и приёмами работы из мира модулей, перейти к построению целевых примеров.

Вот с такого образца простейшего модуля ядра (архив `first_hello.tgz`) мы и начнём наш экскурс:

hello_printk.c :

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

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

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

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

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

Сборка модуля

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

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

Makefile :

```
CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)
DEST = /lib/modules/$(CURRENT)/misc

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

default:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	@rm -f *.o .*.cmd .*.flags *.mod.c *.order
	@rm -f .*.*.cmd *.symvers *~ *.*~ TODO.*
	@rm -fR .tmp*
	@rm -rf .tmp_versions
```

Цель сборки `clean` — присутствует в таком и неизменном виде практически во всех далее приводимых в архиве файлах сценариев сборки (`Makefile`), и не будет далее показываться в тексте.

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

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

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

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

На этом модуль создан. Начиная с ядер 2.6 расширение файлов модулей сменено с `*.o` на `*.ko`:

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

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

Загрузка и исполнение

Наш модуль при загрузке/выгрузке выводит сообщение посредством вызова `printk()`. Этот вывод направляется на **текстовую консоль**. При работе в **терминале** (в графической системе X11) вывод не попадает

в терминал, и его можно видеть **только** в лог файле `/var/log/messages`. Но и чисто в текстовую консоль (при отработке в текстовом режиме) вывод направляется не непосредственно, а через демон системного журнала, а выводится на экран только если демон конфигурирован для вывода такого уровня сообщений. Изучаем полученный файл модуля:

```
$ modinfo ./hello_printk.ko
filename:       hello_printk.ko
author:         Oleg Tsiliuric <olej@front.ru>
license:        GPL
srcversion:     83915F228EC39FFCBAF99FD
depends:
vermagic:       2.6.32.9-70.fc12.i686.PAE SMP mod_unload 686
```

Загружаем модуль и исследуем его выполнение несколькими часто используемыми командами:

```
$ sudo insmod ./hello_printk.ko
$ lsmod | head -n2
Module                  Size  Used by
hello_printk             557  0
$ sudo rmmod hello_printk
$ lsmod | head -n2
Module                  Size  Used by
vfat                    6740  2
$ dmesg | tail -n2
Hello, world!
Goodbye, world!
$ sudo cat /var/log/messages | tail -n3
Mar  8 01:44:14 notebook ntpd[1735]: synchronized to 193.33.236.211, stratum 2
Mar  8 02:18:54 notebook kernel: Hello, world!
Mar  8 02:19:13 notebook kernel: Goodbye, world!
```

Последними 2-мя командами показаны 2 основных метода визуализации сообщений ядра (занесенных в системный журнал): утилита `dmesg` и прямое чтение файла журнала `/var/log/messages`. Они имеют несколько отличающийся формат: файл журнала содержит метки времени поступления сообщений, что иногда бывает нужно. Кроме того, прямое чтение файла журнала требует, в некоторых дистрибутивах, наличия прав `root`.

Точки входа и завершения

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

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

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

Функция завершения, совершенно симметрично, имеет прототип, и объявляется макросом `module_exit()`, как было показано:

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

```
module_exit( hello_exit );
```

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

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

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

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

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

Внутренний формат модуля

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

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

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

```
$ objdump
Usage: objdump <option(s)> <file(s)>
 Display information from object <file(s)>.
....
```

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

```
$ objdump -h hello_printk.ko
hello_printk.ko:     file format elf32-i386
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
...
  1 .text         00000000  00000000  00000000  00000058  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .exit.text    00000015  00000000  00000000  00000058  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  3 .init.text    00000011  00000000  00000000  0000006d  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
...
```

```
  5 .modinfo      0000009b  00000000  00000000  000000a0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .data         00000000  00000000  00000000  0000013c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
...
  8 .bss          00000000  00000000  00000000  000002a4  2**2
                  ALLOC
...
```

Здесь секции:

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

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

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

Здесь хорошо видны имена (функций) описанных в коде нашего модуля, с ними вместе указывается имя секции, в которой находятся эти имена.

Ещё один альтернативный инструмент детального анализа объектной структуры модуля (он даёт несколько иные срезы информации), хорошо видна сфера видимости имён (колонка Bind : LOCAL — имя не экспортируется за пределы кода модуля):

```
$ readelf -s hello_printk.ko
Symbol table '.symtab' contains 35 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
    22: 00000000    21 FUNC    LOCAL  DEFAULT    3 hello_exit
    23: 00000000    17 FUNC    LOCAL  DEFAULT    5 hello_init
    24: 00000000    38 OBJECT  LOCAL  DEFAULT    8 __mod_author5
    25: 00000028    12 OBJECT  LOCAL  DEFAULT    8 __mod_license4
...
```

Примечание: Здесь самое время отвлечься и рассмотреть вопрос, чтобы к нему больше не обращаться — чем формат модуля `*.ko` **отличается** от обыкновенного объектного формата `*.o` (тем более, что последний появляется в процессе сборки модуля как промежуточный результат):

```
$ ls -l *.o *.ko
-rw-rw-r-- 1 olej olej 92209 Июн 13 22:51 hello_printk.ko
-rw-rw-r-- 1 olej olej 46396 Июн 13 22:51 hello_printk.mod.o
-rw-rw-r-- 1 olej olej 46956 Июн 13 22:51 hello_printk.o
$ modinfo hello_printk.o
filename:       hello_printk.o
author:         Oleg Tsiliuric <olej@front.ru>
```

```
license:        GPL
$ modinfo hello_printk.ko
filename:       hello_printk.ko
author:         Oleg Tsiliuric <olej@front.ru>
license:        GPL
srcversion:     83915F228EC39FFCBAF99FD
depends:
vermagic:       2.6.32.9-70.fc12.i686.PAE SMP mod_unload 686
```

Легко видеть, что при сборке к файлу модуля добавлено несколько внешних имён, значения которых используются системой для **контроля** возможности корректной **загрузки** модуля.

Вывод диагностики модуля

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

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

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

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

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

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

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

Вызов `printk()` не производит непосредственно какой-либо вывод, а направляет выводимую строку демону системного журнала, который уже перезаписывает полученную строку: а). на **текстовую консоль** и б). в системный журнал. При работе в графической системе X11, вывод `printk()` в терминал `xterm` (или другой) не попадает (графический терминал **не является** текстовой консолью!), поэтому остаётся только в системном журнале. Это имеет, помимо прочего, тонкое следствие, которое часто упускается из виду: независимо от того, завершается или нет строка, формируемая `printk()`, переводом строки (`'\n'`), «решать» переводить или нет строку будет демон системного журнала (`klogd` или `rsyslogd`), и разные демоны, похоже, решают это по-разному. Таким образом, попытка конкатенации строк:

```
printk( "string1" );
printk( " + string2" );
```

```
printk( " + string3\n" );
```

- в любом из показанных вариантов окажется неудачной: в системе Fedora 12 (ядро 2.6.32 с `rsyslog`) будет выведено 3 строки, а в CentOS 5.2 (ядро 2.6.18 с `klogd`) это будет единая строка: `string1 + <4>string2 + <4>string3`, но это наверняка не то, что вы намеревались получить... А что же делать, если нужно конкатенировать вывод в зависимости от условий? Нужно формировать весь нужный вывод в строку с помощью `sprintf()`, а потом выводить всю эту строку посредством `printk()` (это вообще хороший способ для модуля, чтобы не дёргать по много раз ядро и демон системного журнала многократными `printk()`).

Вывод системного журнала направляется, как уже сказано, и отображается в текстовой консоли, но не отображается в графических терминалах X11. Вы всегда можете оперативно переключаться между графическим экраном X11 и несколькими (обычно 6, зависит от конфигурации) текстовыми консолями, делается это клавишной комбинацией: <Ctrl><Alt><Fi>, где Fi - «функциональная клавиша», а i — номер консоли. Но вот распределение экранов по i может быть разным (в зависимости от способа конфигурации дистрибутива Linux, на какой консоли запускается система X11), я встречал:

- в Fedora 12 : <Ctrl><Alt><F1> - X11, а <Ctrl><Alt><F2>...<F7> - текстовые консоли;

- в CentOS 5.2 : <Ctrl><Alt><F1>...<F6> - текстовые консоли, а <Ctrl><Alt><F7> - X11;

Большой неожиданностью может стать отсутствие вывода `printk()` в текстовую консоль. Но такой вывод обеспечивается демоном системного журнала, и он выводит только сообщения **выше порога**, установленного ему при запуске. Для снижения порога вывода диагностики демон системного журнала может быть придётся перезапустить с другими параметрами. Детали вы найдёте в описании конкретного демона системного журнала, используемого в вашей системе (а это могут быть: `syslogd + klogd`, `rsyslogd`, `Journal` и др.).

Основные ошибки модуля

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

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

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

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

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

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

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

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

- в коде модуля отсутствует макрос определения лицензии GPL вида:

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

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

Ошибка (сообщение) может возникнуть и при попытке **выгрузить** модуль (но реже). Более того, обратите внимание, что прототип функции выгрузки модуля `void module_exit( void )` – не имеет возможности вернуть код причины неудачного завершения: все сообщения могут поступать только от подсистемы управления модулями операционной системы. Наиболее часто получаемые ошибки при неудачной попытке выгрузить модуль:

`ERROR: Removing 'params': Device or resource busy` – счётчик ссылок модуля ненулевой, в системе есть (возможно) модули, зависимые от данного. Но не исключено и то, что вы в самом своём коде инкрементировали счётчик ссылок, не декрементировав его назад. Такая же ситуация может возникать после аварийного сообщения ядра Ooops... после загрузки модуля (ошибка в коде модуля в ходе отработки). Вот протокол реальной ситуации после такой ошибки:

```
$ sudo rmmod mod_ser
ERROR: Module mod_ser is in use
$ echo $?
1
$ lsmod | head -n2
Module                  Size  Used by
mod_ser                 2277  1
```

Здесь счётчик ссылок модуля не нулевой, но нет имени модулей, ссылающихся на данный. Что можно предпринять в подобных случаях? Только перезагрузка системы!

`ERROR: Removing 'params': Operation not permitted` — самая частая причина такого сообщения — у вас просто нет прав `root` на выполнение операции `rmmod`. Более экзотический случай появления такого сообщения: не забыли ли вы в коде модуля вообще прописать функцию выгрузки (`module_exit()`)? В этом случае в списке модулей можно видеть довольно редкий квалификатор `permanent` (в этом случае вы создали не выгружаемый модуль, поможет только перезагрузка системы) :

```
$ /sbin/lsmod | head -n2
Module                  Size  Used by
params                  6412  0 [permanent]
...
```

Интерфейсы модуля

Модуль ядра является некоторым согласующим звеном потребностей, возникающих в пространстве пользователя, с механизмами, реализующими эти потребности в пространстве ядра. Таким образом модуль (код модуля) имеет (и пользуется ними) набор предоставляемых интерфейсов как в сторону взаимодействия с монолитным ядром Linux (с кодом ядра, с API ядра, структурами данных...), так и в сторону взаимодействия с пользовательским пространством (пользовательскими приложениями, пространством файловых имён, реальным оборудованием, каналами обмена данными...). Поэтому удобно раздельно рассматривать механизмы

взаимодействия модуля в направлении пользователя (**в наружу**) и в направлении механизмов ядра (**во внутрь**).

Взаимодействие модуля с уровнем пользователя

Если с интерфейсом модуля в сторону ядра всё относительно единообразно — это API предоставляемый со стороны ядра, то вот с уровнем пользователя (командами, приложениями, системными файлами, внешними устройствами и другое разное — то, что заметно пользователю) у модуля есть много разных по своему назначению способов взаимодействия.

1. **Диагностика из модуля** (в системный журнал) `printk()` :

– осуществляет вывод в **текстовую консоль** (не графический терминал!);

– осуществляет вывод в файл журнала `/var/log/messages` ;

– содержимое файла журнала можно дополнительно посмотреть командой `dmesg`;

– это основное средство диагностики, а часто и отладки.

Конечно, сам вызов `printk()` это такой же вызов API ядра (интерфейс вовнутрь), как, скажем, `strlen()`, к примеру. Но эффект, результат, производимый таким вызовом, **наблюдается в пространстве пользователя** (вовне). Кроме того, это настолько значимый интерфейс к модулю, что мы должны его здесь выделить из числа прочих вызовов API ядра и отметить как отдельный специфический интерфейс.

2. **Копирование данных** в программном коде между пользовательским адресным пространством и пространством ядра (выполняется только по инициативе модуля). Это самая важная часть взаимодействия модуля с пользователем. Конечно, опять же, это всё те же вызовы из числа API ядра, но эти несколько вызовов предназначены для узко утилитарных целей: обмен данными между адресным пространством ядра и пространством пользователя. Вот эти вызовы (мы их получим динамически из таблицы имён ядра, а только после этого посмотрим определения в заголовочных файлах):

```
$ cat /proc/kallsyms | grep T | grep copy_to_user
...
c0679ba0 T copy_to_user
...
$ cat /proc/kallsyms | grep T | grep copy_from_user
c04fd1d0 T iov_iter_copy_from_user
c04fd250 T iov_iter_copy_from_user_atomic
c06796b0 T copy_from_user_nmi
...
```

Реализованные `inline` вызовы `copy_to_user()` и `copy_from_user()` являются вызовами API ядра для данных **произвольного размера**. Для скалярных данных (фиксированного размера: `u8` — `u64`) они просто копируют требуемый элемент данных (1-8 байт) между пространствами пользователя и ядра (в том или другом направлении). Для данных отличающегося размера (произвольных данных) эти вызовы, после требуемой проверки **доступности страницы данных**, вызывают обычный `memcpy()` между областями[4].

```
$ cat /proc/kallsyms | grep T | grep put_user
c0679368 T __put_user_1
c067937c T __put_user_2
c0679394 T __put_user_4
c06793ac T __put_user_8
$ cat /proc/kallsyms | grep T | grep __get_user
c043f680 T __get_user_pages_fast
c0522630 T __get_user_pages
c06784b4 T __get_user_1
c06784c8 T __get_user_2
c06784e0 T __get_user_4
```

4 Сказанное относится к процессорной архитектуре x86, в другой процессорной архитектуре конкретная реализация `copy_*_user()` может быть совершенно другой, на что и указывает каталог размещения этих определений: `asm/uaccess.h`.

Вызовы `put_user()` и `get_user()` - **это макросы**, используемые для обмена простыми скалярными типами данных (байт, слово, ...), которые пытаются определить размер пересылаемой порции данных (1, 2, 4 байта - для `get_user()`, 1, 2, 4, 8 байт - для `put_user()`[5]) - именно то, что они реализуются как макросы, это даёт им возможность определить размер передаваемой порции данных без явного указания (по типу данных имени переменной). Почему мы не видим `put_user()` и `get_user()` среди экспортируемых символов ядра в `/proc/kallsyms`? Именно потому, что это макросы, подстановочно (при компиляции) разрешающиеся в имена с подчёркиванием, перечисленные выше. Вызовы `put_user()` и `get_user()` вызывают, в итоге, `copy_to_user()` и `copy_from_user()` для соответствующего числа байт, по типу:

```
static inline int __put_user_fn( size_t size, void __user *ptr, void *x ) {
   size = __copy_to_user(ptr, x, size);
   return size ? -EFAULT : size;
}
```

Определения всех API этой группы можно найти в `<asm/uaccess.h>` (http://lxr.free-electrons.com/source/include/asm-generic/uaccess.h), все они реализованы `inline` и доступны для рассмотрения. Прототипы имеют вид (для макросов `put_user()` и `get_user()` восстановлен вид, как он имел бы для функциональных вызовов - с типизированными параметрами):

```
long copy_from_user( void *to, const void __user * from, unsigned long n );
long copy_to_user( void __user *to, const void *from, unsigned long n );
int put_user( const void *x, void __user *ptr );
int get_user( void *x, const void __user *ptr );
```

Каждый вызовов этой группы API возвращает нулевое значение в качестве признака успешной операции, либо исходно переданное значение параметра `n` (положительное целое), как признак недоступности области для копирования.

Выше обсуждены наиболее часто упоминаемые вызовы обмена пространств ядра и пользователя. Там же (`<asm/uaccess.h>`) определяются ещё ряд менее употребляемых вызовов (их конкретный перечень может меняться от версии ядра). Они предназначены для работы с нуль-терминальными символьными строками, учитывая, что в значительной части случаев обмениваемые данные — это именно символьные строки:

```
long strncpy_from_user( char *dst, const char __user *src, long count );
long strnlen_user( const char __user *src, long n );
long strlen_user( const char __user *src)
unsigned long clear_user( void __user *to, unsigned long n )
```

Для строчных функций `strncpy_from_user()`, `strnlen_user()`, `strlen_user()` очень важное отличие (отличающее их от похожих по наименованию функций POSIX), состоит в том, что длина в этих вызовах трактуется **включая терминальный ноль** (завершающий строку), поскольку при копировании этот нулевой байт тоже подлежит передаче.

Особенностью **всех** функций копирования данных между пространствами пользователя и ядра состоит в том, что реальному физическому копированию **всегда** предшествует проверка доступности страницы памяти, к которой принадлежат данные в пространстве пользователя. Поскольку мы используем виртуальный (логический адрес) эта страница на момент вызова может отсутствовать в памяти, быть выгруженной на диск. В таком случае потребуется загрузка страницы с диска. А это приведёт к блокированному состоянию (ожиданию) породившего ситуацию вызова `copy_*_user()`. А раз так, то **все** вызовы этой группы недопустимы из обработчика аппаратного прерывания, **из контекста прерывания**, о чём будет говориться далее. Такой вызов, если он произведён из контекста прерывания, чреват полной аварийной остановкой операционной системы.

3. Интерфейс взаимодействия посредством создания **символьных имён устройств**, вида `/dev/XXX`. Модуль может обеспечивать поддержку стандартных операций ввода-вывода на устройстве (как символьном, там и блочном). Это **основной** интерфейс модуля к пользовательскому уровню. И основной механизм построения **драйверов** периферийных устройств. Модуль должен обеспечить, кроме механизма **создания** символьного имени устройства (потокового или блочного), и весь минимум типовых операций для такого устройства, например, возможность из пользовательского процесса выполнить:

```
int fd = open( /dev/XXX, O_RDWR );
```

5 Почему такая асимметрия я не готов сказать.

```
int n = read( fd, ... );
n = write( fd, ... );
close( fd );
```

4. Взаимодействие через псевдофайлы (путевые имена) системы `/proc` (файловая система `procfs`). Модуль может создавать специфические для него индикативные псевдофайлы в `/proc`, туда модуль может писать отладочную или диагностическую информацию, или читать оттуда информацию управляющую. Соответственно, пользователь может читать из этих имён диагностику, или записывать туда управляющие воздействия. Это то, что иногда называют передачей внеполосовых данных, часто этот механизм (система `/proc`) предлагается как замена, альтернатива ненадёжному механизму `ioctl()` для устройств. Эти псевдофайлы в `/proc` доступны для чтения-записи всеми стандартными командами Linux (в пределах регламента прав доступа, установленных для конкретного файлового имени):

```
# cat /proc/irq/27/smp_affinity
4
# echo 2 > /proc/irq/27/smp_affinity
```

Информация в `/proc` (и в `/sys`, о котором сказано ниже) представляется только в **символьном** изображении, даже если это отображение единичного числового значения.

5. Взаимодействие через файлы (имена) системы `/sys` (файловая система `sysfs`). Эта файловая система подобна (по назначению) `/proc`, но возникла заметно **позже**. Первоначально система `/sys` была создана для конфигурирования и управления периферийными устройствами (совместно с дуальной к ней подсистемой пространства пользователя `udev`). Но к настоящему времени её возможности намного шире этого. Считается, что её функциональность выше `/proc`, и она во многих качествах будет со временем заменять `/proc`. Некоторой отличительной особенностью информации в `/proc` и в `/sys` есть то (но это больше традиция и привычки, ничего больше), что в `/proc` текстовая информация чаще представлены многострочным содержимым, таблицами, а в `/sys` — это чаще символьное представление одиночных значений:

```
$ cat /sys/module/battery/parameters/cache_time
1000
```

6. Взаимодействие модуля со стеком сетевых протоколов (главным образом со стеком протоколов IP, но это не принципиально важно, стек протоколов IP просто намного более развит в Linux, чем другие протокольные семейства). Модуль может **создавать** новые сетевые интерфейсы, присваивая им произвольные имена, которые отображаются в общем списке сетевых интерфейсов хоста:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: em1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group
default qlen 1000
    link/ether a0:1d:48:f4:93:5c brd ff:ff:ff:ff:ff:ff
3: wlo1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DORMANT group
default qlen 1000
    link/ether 34:23:87:d6:85:0d brd ff:ff:ff:ff:ff:ff
```

Для такого интерфейса допустимо выполнение всех команд сетевой системы, таких как `ip`, `ifconfig`, `route`, `netstat` и всех других, по нему системой собирается статистика трафика и ошибок. Кроме того, модуль может, для вновь созданных или существующих интерфейсов, создавать новые протоколы, или модифицировать передаваемую по ним информацию (что актуально для шифрования, криптографирования потока, или сопровождения его какими-то метками данных).

Взаимодействие модуля с ядром

Для взаимодействия модуля с ядром, ядро (и подгружаемые к ядру модули) **экспортируют** набор имён, которые новый модуль использует в качестве **API ядра** (это и есть тот набор вызовов, о котором мы говорили чуть выше, специально в примере показан уже обсуждавшийся вызов `sprintf()`, о котором мы уже знаем, что он как близнец похож на `sprintf()` из системной библиотеки GCC, но это совсем другая реализация). **Все** известные (об экспортировании мы поговорим позже) имена ядра мы можем получить:

```
$ awk '/T/ && /print/ { print $0 }' /proc/kallsyms
...
```

```
c042666a T printk
...
c04e5b0a T sprintf
c04e5b2a T vsprintf
...
d087197e T scsi_print_status    [scsi_mod]
...
```

Вызовы API ядра осуществляются по прямому **абсолютному** адресу. Каждому экспортированному ядром или любым модулем имени соотносится адрес, он и используется для связывания при загрузке модуля, использующего это имя. Это основной механизм взаимодействия модуля с ядром.

Динамически формируемый (после загрузки) список имён ядра находится в файле `/proc/kallsyms`. Но в этом файле: а). ~85К строчек, и б). далеко не все перечисленные там имена доступны модулю для связывания. Для того, чтобы разрешить первую проблему, нам необходимо бегло пользоваться (для фильтрации по самым замысловатым критериям) такими инструментами анализа регулярных выражений, как `grep, awk (gawk), sed, perl` или им подобными. Ключ ко второй нам даёт информация по утилите `nm` (анализ символов объектного формата), хотя эта утилита никаким боком и не соотносится непосредственно с программированием для ядра:

```
$ nm --help
Usage: nm [option(s)] [file(s)]
 List symbols in [file(s)] (a.out by default).
...
$ man nm
...
    if uppercase, the symbol is global (external).
...
    "D" The symbol is in the initialized data section.
    "R" The symbol is in a read only data section.
    "T" The symbol is in the text (code) section.
...
```

Таким образом, например:

```
$ cat /proc/kallsyms | grep sys_call
c052476b t proc_sys_call_handler
c07ab3d8 R sys_call_table
```

- важнейшее имя ядра `sys_call_table` (таблица системных вызовов) известно в таблице имён, но не экспортируется ядром, и недоступно для связывания коду модулей (мы ещё детально вернёмся к этому вопросу).

Примечание: имя `sys_call_table` может присутствовать в `/proc/kallsyms`, а может и нет — я наблюдал 1-е в Fedora 12 (2.6.32) и 2-е в CentOS 5.2 (2.6.18). Это имя вообще экспортировалось ядром до версий ядра 2.5, и могло напрямую быть использовано в коде, но такое состояние дел было признано не безопасным к ядру 2.6.

Относительно API ядра нужно знать следующее:

1. Эти функции реализованы в ядре, при совпадении многих из них по форме с вызовами стандартной библиотеки С или системными вызовами по форме (например, всё тот же `sprintf()`) - это **совершенно другие** функции. Заголовочные файлы для функций пространства пользователя располагаются в `/usr/include`, а для API ядра — в совершенно другом месте, в каталоге `/lib/modules/`uname -r`/build/include`, это различие особенно важно.

2. Разработчики ядра не связаны требованиями совместимости снизу вверх, в отличие от очень жёстких ограничений на пользовательские API, налагаемые стандартом POSIX. Поэтому API ядра достаточно произвольно меняются даже от одной **подверсии** ядра к другой. Они плохо документированы (по крайней мере, в сравнении с документацией POSIX вызовов пользовательского пространства). Детально изучать их приходится только по исходным кодам Linux (и по кратким

текстовым файлам заметок в дереве исходных кодов ядра).

Коды ошибок

Коды ошибок API ядра в основной массе это те же коды ошибок, прекрасно известные по пространству пользователя, определены они в <asm-generic/errno-base.h> (показано только начало обширной таблицы):

```
#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Argument list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define ECHILD          10      /* No child processes */
#define EAGAIN          11      /* Try again */
#define ENOMEM          12      /* Out of memory */
#define EACCES          13      /* Permission denied */
#define EFAULT          14      /* Bad address */
#define ENOTBLK         15      /* Block device required */
#define EBUSY           16      /* Device or resource busy */
...
```

Основное различие состоит в том, что вызовы API ядра возвращают этот код со **знаком минус**, так как и нулевые и положительные значения возвратов зарезервированы для результатов нормального завершения. Так же (как отрицательные значения) должен возвращать коды ошибочного завершения программный код вашего модуля. Таковы соглашения в пространстве ядра.

Варианты загрузки модулей

При отработке нового модуля его загрузку на тестирование вы будете, скорее всего, производить утилитой `insmod`. Утилита `insmod` получает **имя файла модуля**, и пытается загрузить его без проверок каких-либо взаимосвязей, как это описано ниже. Если некоторые требуемые имена **не экспортированы** к моменту `insmod`, то загрузка отвергается.

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

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

```
$ ls -l /lib/modules/`uname -r`/*.dep
-rw-r--r-- 1 root root 206131 Map  6 13:14 /lib/modules/2.6.32.9-70.fc12.i686.PAE/modules.dep
```

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

```
# depmod -a
```

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

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

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

Интересующий нас файл содержит строки вида:

```
$ cat /lib/modules/`uname -r`/modules.dep
...
kernel/fs/ubifs/ubifs.ko: kernel/drivers/mtd/ubi/ubi.ko kernel/drivers/mtd/mtd.ko
...
```

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

Утилита `rmmod` выгружает ранее загруженный модуль, в качестве параметра утилита должна получать **имя модуля** (не **имя файла модуля**[6]). Если в системе есть модули, зависимые от выгружаемого (счётчик ссылок использования модуля больше нуля), то выгрузка модуля не произойдёт, и утилита `rmmod` завершится аварийно.

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

Параметры загрузки модуля

Модулю при его загрузке могут быть переданы значения параметров — здесь наблюдается полная аналогия (по смыслу, но не по формату) с передачей параметров пользовательскому процессу из командной строки через массив `argv[]`. Такую передачу модулю параметров при его загрузке можно видеть в ближайшем рассматриваемом драйвере символьного устройства (архив `cdev.tgz` примеров). Более того, в этом модуле, если не указано явно значение параметра, то для него устанавливается его умалчиваемое значение (динамически определяемый системой старший номер устройства), а если параметр указан — то принудительно устанавливается заданное значение, даже если оно и недопустимо с точки зрения системы. Этот фрагмент выглядит так:

```
static int major = 0;
module_param( major, int, S_IRUGO );
```

Здесь определяется переменная-параметр (с именем `major`), и далее это же имя указывается в макросе `module_param()`. Подобный макрос должен быть записан для каждого предусмотренного параметра, и должен последовательно определить: а). имя (параметра и переменной), б). тип значения этой переменной, в). права доступа к параметру, отображаемому как путевое имя в системе `/sys`.

Значения параметрам могут быть установлены во время загрузки модуля через `insmod` или `modprobe`; последняя команда также может прочитать значение параметров из своего файла конфигурации (`/etc/modprobe.conf`) для загрузки модулей.

Обработка входных параметров модуля обеспечивается макросами (описаны в `<linux/moduleparam.h>`), вот основные (там же есть ещё ряд мало употребляемых), два из них приводятся с полным определением через другие макросы (что добавляет понимания):

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

6 В последних версиях ядра в качестве аргумента `rmmod` может быть указано и полное имя файла модуля с расширением `.ko`.

```
module_param_string( name, string, len, perm )
module_param_array_named( name, array, type, nump, perm )
#define module_param_array( name, type, nump, perm ) \
      module_param_array_named( name, name, type, nump, perm )
```

Но даже из этого подмножества употребляются чаще всего только два: `module_param()` и `module_param_array()` (детально понять работу макросов можно реально выполняя обсуждаемый ниже пример).

Примечание: Последним параметром `perm` указаны права доступа (например, `S_IRUGO | S_IWUSR`), относящиеся к имени параметра, отображаемому в подсистеме `/sys`, если нас не интересует имя параметра отображаемое в `/sys`, то хорошим значением для параметра `perm` будет 0.

Для параметров модуля в макросе `module_param()` могут быть указаны следующие типы:

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

- `charp` - значение указателя на `char` - выделяется память для строки, заданной пользователем (не нужно предварительно распределять место для строки), и указатель устанавливается соответствующим образом.

- `int, long, short, uint, ulong, ushort` - базовые целые величины разной размерности; версии, начинающиеся с u, являются беззнаковыми величинами.

В качестве входного параметра может быть определён и массив выше перечисленных типов (макрос `module_param_array()`).

Пример, показывающий большинство приёмов использования параметров загрузки модуля (архив `parms.tgz`) показан ниже:

mod_params.c :

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

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

static int iparam = 0;        // целочисленный парамер
module_param( iparam, int, 0 );

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

static bool bparam = true;    // логический инверсный парамер
module_param( bparam, invbool, 0 );

static char* sparam;          // строчный параметр
module_param( sparam, charp, 0 );

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

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

static int __init mod_init( void ) {
```

```
    int j;
    char msg[ 40 ] = "";
    printk( "========================================\n" );
    printk( "iparam = %d\n", iparam );
    printk( "nparam = %d\n", k );
    printk( "bparam = %d\n", bparam );
    printk( "sparam = %s\n", sparam );
    printk( "cparam = %s {%d}\n", s, strlen( s ) );
    sprintf( msg, "aparam [ %d ] = ", arnum );
    for( j = 0; j < arnum; j++ ) sprintf( msg + strlen( msg ), " %d ", aparam[ j ] );
    printk( "%s\n", msg );
    printk( "========================================\n" );
    return -1;
}

module_init( mod_init );
```

В коде этого модуля присутствуют две вещи, которые могут показаться непривычными программисту на языке С, нарушающие стереотипы этого языка, и поначалу именно в этом порождающие ошибки программирования в собственных модулях:

- отсутствие резервирования памяти для символьного параметра `sparam`[7];
- и динамический размер параметра-массива `aparam`. (динамически изменяющийся после загрузки модуля);
- при этом этот динамический размер **не может** превысить статически зарезервированную максимальную размерность массива (такая попытка вызывает ошибку).

Но и то, и другое, хотелось бы надеяться, достаточно разъясняется демонстрируемым кодом примера.

Для сравнения - выполнение загрузки модуля с параметрами по умолчанию (без указания параметров), а затем с переопределением значений всех параметров:

```
# sudo insmod mod_params.ko
insmod: ERROR: could not insert module ./mod_params.ko: Operation not permitted
$ dmesg | tail -n8
[14562.245812] ========================================
[14562.245816] iparam = 0
[14562.245818] nparam = 0
[14562.245820] bparam = 1
[14562.245822] sparam = (null)
[14562.245824] cparam =  {0}
[14562.245828] aparam [ 5 ] =  0  0  0  0  0
[14562.245830] ========================================

# insmod mod_params.ko iparam=3 nparam=4 bparam=1 sparam=str1 cparam=str2 aparam=5,4,3
insmod: ERROR: could not insert module mod_params.ko: Operation not permitted
$ dmesg | tail -n8
[15049.389328] ========================================
[15049.389336] iparam = 3
[15049.389338] nparam = 4
[15049.389340] bparam = 0
[15049.389342] sparam = str1
[15049.389345] cparam = str2 {4}
[15049.389348] aparam [ 3 ] =  5  4  3
[15049.389350] ========================================
```

При этом массив `aparam` получил и новую размерность `arnum`, и его элементам присвоены новые значения.

7 Объявленный в коде указатель просто устанавливается на строку, размещённую где-то в параметрах запуска программы загрузки. При этом остаётся открытым вопрос: а если **после** отработки инсталляционной функции, **резидентный** код модуля обратится к такой строке, к чему это приведёт? Я могу предположить, что к критической ошибке, а вы можете проверить это экспериментально.

Вводимые параметры загрузки и их значения в команде `insmod` жесточайшим образом контролируются (хотя, естественно, всё проконтролировать абсолютно невозможно), потому как модуль, загруженный с ошибочными значениями параметров, который становится составной частью ядра — это угроза целостности системы. Если **хотя бы один** из параметров признан некорректным, загрузка модуля не производится. Вот как происходит контроль для некоторых случаев:

```
# insmod mod_params.ko aparam=5,4,3,2,1,0
insmod: ERROR: could not insert module mod_params.ko: Invalid parameters
# echo $?
1
$ dmesg | tail -n2
[15583.285554] aparam: can only take 5 arguments
[15583.285561] mod_params: `5' invalid for parameter `aparam'
```

Здесь имела место попытка заполнить в массиве `aparam` число элементов большее, чем его зарезервированная размерность (5).

Попытка загрузки модуля с указанием параметра с именем, не предусмотренным в коде модуля, в некоторых конфигурациях (Fedora 14) приведёт к ошибке нераспознанного параметра, и модуль не будет загружен:

```
$ sudo /sbin/insmod ./mod_params.ko zparam=3
insmod: error inserting './mod_params.ko': -1 Unknown symbol in module
$ dmesg | tail -n1
mod_params: Unknown parameter `zparam'
```

Но в других случаях (Fedora 20) такой параметр будет просто проигнорирован, а модуль будет нормально загружен:

```
# insmod mod_params.ko ZZparam=3
insmod: ERROR: could not insert module mod_params.ko: Operation not permitted
$ dmesg | tail -n8
[15966.050023] ========================================
[15966.050026] iparam = 0
[15966.050029] nparam = 0
[15966.050031] bparam = 1
[15966.050033] sparam = (null)
[15966.050035] cparam =  {0}
[15966.050039] aparam [ 5 ] =  0  0  0  0  0
[15966.050041] ========================================
```

К таким (волатильным) возможностям нужно относиться с большой осторожностью!

```
# insmod mod_params.ko iparam=qwerty
insmod: ERROR: could not insert module ./mod_params.ko: Invalid parameters
$ dmesg | tail -n1
[16625.270285] mod_params: `qwerty' invalid for parameter `iparam'
```

Так выглядит попытка присвоения не числового значения числовому типу.

```
# insmod mod_params.ko cparam=123456789
insmod: ERROR: could not insert module mod_params.ko: No space left on device
$ dmesg | tail -n2
[16960.871302] cparam: string doesn't fit in 4 chars.
[16960.871309] mod_params: `123456789' too large for parameter `cparam'
```

А здесь была превышена максимальная длина для строки-параметра, передаваемой копированием.

Сигнатура верификации модуля

Начиная с ядра примерно 3.13, при загрузке собранных модулей вы можете начать получать сообщения вида (которые раньше не появлялись):

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

Это означает, что модуль не подписан сигнатурой секретного ключа, если система собиралась с требованием сигнатур:

```
$ cat /boot/config-`uname -r` | grep CONFIG_MODULE_SIG
CONFIG_MODULE_SIG=y
# CONFIG_MODULE_SIG_FORCE is not set
CONFIG_MODULE_SIG_ALL=y
CONFIG_MODULE_SIG_UEFI=y
# CONFIG_MODULE_SIG_SHA1 is not set
# CONFIG_MODULE_SIG_SHA224 is not set
CONFIG_MODULE_SIG_SHA256=y
# CONFIG_MODULE_SIG_SHA384 is not set
# CONFIG_MODULE_SIG_SHA512 is not set
CONFIG_MODULE_SIG_HASH="sha256"
```

Такое сообщение не препятствует загрузке и использованию модуля, но предупреждает, что это некий проприетарный модуль, и аварийные сообщения системы, если они возникнут, могут быть искажены из-за этого модуля. Но если, например, установлен параметр `CONFIG_MODULE_SIG_FORCE`, то модули не подписанные сигнатурой будут вообще отвергаться, и не станут загружаться. Детальней о сигнатуре модуля можно почитать в файле документации исходников ядра module-signing.txt.

Конфигурационные параметры ядра[8]

Все, кто собирал ядро, знают, какое великое множество параметров там можно переопределить при конфигурировании, предшествующему компиляции. Эти параметры, с которыми собрано конкретное ядро, сохранены в файле `<linux/autoconf.h>`[9] и доступны в коде модуля. Символические имена параметров в этом файле имеют вид: `CONFIG_*`. Все конфигурационные параметры ядра, определяемые в диалоге сборки, бывают:

- те параметры ядра, которые были выбраны в диалоге сборки с ответом 'y' — соответствующие им конфигурационные параметры определены в файле `<linux/autoconf.h>` со значением 1.

- те параметры ядра, для которых в диалоге сборки были установлены символьные значения (чаще всего, это путевые имена в файловой системе Linux) — определены в файле `<linux/autoconf.h>` как символьные константы, имеющие именно эти значения.

- сложнее с теми параметрами ядра, которые были выбраны в диалоге сборки с ответом 'm' (собирать такую возможность как подгружаемый модуль) — тогда символическая константа с именем именем `CONFIG_XXX`, соответствующим конфигурационному параметру, не будет определена в файле `<linux/autoconf.h>`, но будет определена другая символическая константа с суффиксом `_MODULE`: `CONFIG_XXX_MODULE`, и значение её будет, естественно, 1.

Например:

```
$ cd /lib/modules/`uname -r`/build/include/linux
$ cat autoconf.h | grep CONFIG_SMP
#define CONFIG_SMP 1
$ cat autoconf.h | grep CONFIG_MICROCODE
```

8 Тема подсказана одним из читателей рукописи.

9 В зависимости от версии и платформы, месторасположение этого файла может отличаться, например: `<generated/autoconf.h>`

– те параметры ядра, которые были выбраны в диалоге сборки с ответот 'n' — не определены (не присутствуют) в файле <linux/autoconf.h>.

```
#define CONFIG_MICROCODE_INTEL 1
#define CONFIG_MICROCODE_MODULE 1
#define CONFIG_MICROCODE_OLD_INTERFACE 1
#define CONFIG_MICROCODE_AMD 1
$ cat autoconf.h | grep CONFIG_64BIT
$ cat autoconf.h | grep CONFIG_OUTPUT_FORMAT
#define CONFIG_OUTPUT_FORMAT "elf32-i386"
```

Таким образом, разрабатываемый вами модуль может динамически **считывать** параметры ядра, к которому он подгружается, и реконфигурироваться в зависимости от набора этих параметров. Естественно, что параметры конфигурации ядра являются для модуля «только для чтения», модуль не может на них никак влиять. Но модуль может анализировать такие параметры как **окружение**, в котором ему предстоит работать, и, в зависимости от этого, устанавливаться в системе тем или иным образом (или вообще не устанавливаться при отсутствии каких-то возможностей, критических с позиции разработчика модуля).

Пример модуля, демонстрирующего доступность конфигурационных параметров ядра в коде модуля показан в архиве config.tgz.

config.c :

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

static int __init hello_init( void ) {
// CONFIG_SMP=y
#if defined(CONFIG_SMP)
   printk( "CONFIG_SMP = %d\n", CONFIG_SMP );
#else
   printk( "CONFIG_SMP не определено\n" );
#endif
// CONFIG_64BIT is not set
#if defined(CONFIG_64BIT)
   printk( "CONFIG_64BIT = %d\n", CONFIG_64BIT );
#else
   printk( "CONFIG_64BIT не определено\n" );
#endif
//CONFIG_MICROCODE=m
#if defined(CONFIG_MICROCODE)
   printk( "CONFIG_MICROCODE = %d\n", CONFIG_MICROCODE );
#else
   printk( "CONFIG_MICROCODE не определено\n" );
#if defined(CONFIG_MICROCODE_MODULE)
   printk( "CONFIG_MICROCODE_MODULE = %d\n", CONFIG_MICROCODE_MODULE );
#else
   printk( "CONFIG_MICROCODE_MODULE не определено\n" );
#endif
#endif
//CONFIG_OUTPUT_FORMAT="elf32-i386"
#if defined(CONFIG_OUTPUT_FORMAT)
   printk( "CONFIG_OUTPUT_FORMAT = %s\n", CONFIG_OUTPUT_FORMAT );
#else
   printk( "CONFIG_OUTPUT_FORMAT не определено\n" );
#endif
   return -1;
}

module_init( hello_init );

MODULE_LICENSE( "GPL" );
```

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

Результат его использования совершенно ожидаемый (учитывая показанный ранее пример содержимого <linux/autoconf.h>):

```
$ sudo insmod config.ko
insmod: error inserting 'config.ko': -1 Operation not permitted
$ dmesg | tail -n30 | grep CONF
CONFIG_SMP = 1
CONFIG_64BIT не определено
CONFIG_MICROCODE не определено
CONFIG_MICROCODE_MODULE = 1
CONFIG_OUTPUT_FORMAT = elf32-i386
```

Эта возможность применима не только для извлечения значений параметров (как CONFIG_OUTPUT_FORMAT), но особенно полезна для предотвращения компиляции модуля в среде неподобающей ему (target platform not supported) конфигурации ядра (пример из mm/percpu-km.c):

```
#if defined(CONFIG_SMP) && defined(CONFIG_NEED_PER_CPU_PAGE_FIRST_CHUNK)
#error "contiguous percpu allocation is incompatible with paged first chunk"
#endif
```

Подсчёт ссылок использования

Одним из важных (и изрядно путанных по описаниям) понятий из сферы модулей есть подсчёт ссылок использования модуля (и не только модуля, но многих объектов и структур ядра). Счётчик ссылок является внутренним полем структуры описания модуля и, вообще то говоря, является слабо доступным пользователю непосредственно. При загрузке модуля начальное значение счётчика ссылок нулевое. При загрузке любого следующего модуля, который использует имена (**импортирует**), экспортируемые данным модулем, счётчик ссылок данного модуля инкрементируется. Модуль, счётчик ссылок использования которого не нулевой, **не может быть выгружен** командой rmmod. Такая тщательность отслеживания сделана из-за критичности модулей в системе: некорректное обращение к несуществующему модулю **гарантирует** крах всей системы.

Смотрим такую простейшую команду:

```
$ lsmod | grep i2c_core
i2c_core               21732  5 videodev,i915,drm_kms_helper,drm,i2c_algo_bit
```

Здесь модуль, зарегистрированный в системе под именем (не имя файла!) i2c_core (имя выбрано произвольно из числа загруженных модулей системы), имеет текущее значение счётчика ссылок 5, и далее следует перечисление имён 5-ти модулей на него ссылающихся. До тех пор, пока эти 5 модулей не будут удалены из системы, удалить модуль i2c_core будет невозможно.

В чём состоит отмеченная выше путаность всего, что относится к числу ссылок модуля? В том, что в области этого понятия происходят постоянные изменения от ядра к ядру, и происходят они с такой скоростью, что литература и обсуждения не поспевают за этими изменениями, а поэтому часто описывают какие-то несуществующие механизмы. До сих пор в описаниях часто можно встретить ссылки на макросы MOD_INC_USE_COUNT() и MOD_DEC_USE_COUNT(), которые увеличивают и уменьшают счётчик ссылок. Но эти макросы остались в ядрах 2.4. В ядре 2.6 их место заняли функциональные вызовы (определённые в <linux/module.h>):

- `int try_module_get( struct module *module )` - увеличить счётчик ссылок для модуля (возвращается признак успешности операции);
- `void module_put( struct module *module )` - уменьшить счётчик ссылок для модуля;
- `unsigned int module_refcount( struct module *mod )` - возвратить значение счётчика ссылок для модуля;

В качестве параметра всех этих вызовов, как правило, передаётся константный указатель `THIS_MODULE`, так что вызовы, в конечном итоге, выглядят подобно следующему:

```
try_module_get( THIS_MODULE );
```

Таким образом, видно, что имеется возможность управлять значением счётчика ссылок из собственного модуля. Делать это нужно крайне обдуманно, поскольку если мы увеличим счётчик и симметрично его позже не уменьшим, то мы вообще не сможем выгрузить модуль (до перезагрузки системы), это один из путей возникновения в системе «перманентных» модулей, другая возможность их возникновения: модуль не имеющий в коде функции завершения. В некоторых случаях может оказаться нужным динамически изменять счётчик ссылок, препятствуя на время возможности выгрузки модуля. Это актуально, например, в функциях, реализующих операции `open()` (увеличиваем счётчик обращений) и `close()` (уменьшаем, восстанавливаем счётчик обращений) для драйверов устройств — иначе станет возможна выгрузка модуля, обслуживающего открытое устройство, а следующие обращения (из процесса пользовательского пространства) к открытому дескриптору устройства будут направлены в не инициированную память!

И здесь возникает очередная путаница (которую можно наблюдать и по коду некоторых модулей): во многих источниках рекомендуется инкрементировать из собственного кода модуля счётчик использований при открытии устрйства, и декрементировать при его закрытии. Это было актуально, но с некоторой версии ядра (я не смог отследить с какой) это отслеживание делается автоматически при выполнении открытия/закрытия. Примеры этого, поскольку мы пока не готовы к рассмотрению многих деталей такого кода, будут детально рассмотрены позже при рассмотрении множественного открытия для устройств (архив `mopen.tgz`).

В деталях о сборке (пишем Makefile)

Далее рассмотрим некоторые особенности процедуры сборки (`make`) проектов, и нарисуем несколько типовых сценариев сборки (`Makefile`) для наиболее часто востребованных случаев, как например: сборка нескольких модулей в одном проекте, сборка модуля объединением нескольких файлов исходных кодов и подобные... Этих случаев хватает на все практически возникающие потребности.

Переменные периода компиляции

Параметры компиляции модуля можно существенно менять, изменяя **переменные периода компиляции**, определённые в скрипте, осуществляющем сборку (`Makefile`), например:

```
EXTRA_CFLAGS += -O3 -std=gnu89 –no-warnings
```

Таким же образом дополняем определения нужных нам препроцессорных переменных, специфических для сборки нашего модуля:

```
EXTRA_CFLAGS += -D EXPORT_SYMTAB -D DRV_DEBUG
```

Примечание: Откуда берутся переменные, не описанные явно по тексту файлу `Makefile`, как, например, `EXTRA_CFLAGS`? Или откуда берутся правила сборки по умолчанию (как в примере примере использования ассемблерного кода разделом ранее)? И как посмотреть эти правила? Всё это вытекает из правил работы утилиты `make`: в конце книги отдельным приложением приведена краткая справка по этим вопросам, там же приведена ссылка на детальное описание утилиты `make`.

Некоторые важные переменные компиляции, используемые в системном скрипте сборки модуля, могут быть переопределены непосредственно в команде сборки. Из числа наиболее важных:

```
$ make KROOT=/lib/modules/2.6.32.9-70.fc12.i686.PAE/build
```

Здесь показано явное определение пути к корневому каталогу исходных файлов сборки, таким образом можно собирать модуль для версии ядра, отличной от текущей, например, для загрузки во встраиваемую конфигурацию.

Ещё важный случай:

```
$ make ARCH=i386
```

В архитектуре x86_64 это будет указанием собирать 32-битные модули. Так же может определяться сборка для других процессорных платформ (ARM, MIPS, PPC, ...).

Как собрать одновременно несколько модулей?

В уже привычного нам вида `Makefile` может быть описано сборка сколь угодно много одновременно собираемых модулей (архив `export-dat.tgz`[10]):

Makefile :

```
...
TARGET1 = md1
TARGET2 = md2
obj-m   := $(TARGET1).o $(TARGET2).o
...
```

Как собрать модуль и использующие программы к нему?

Часто нужно собрать модуль и одновременно некоторое число пользовательских программ, используемых одновременно с модулем (тесты, утилиты, ...). Зачастую модуль и пользовательские программы используют общие файлы определений (заголовочные файлы). Вот фрагмент подобного `Makefile` - в одном рабочем каталоге собирается модуль и все использующие его программы (архив `ioctl.tgz`). :

Makefile :

```
...
TARGET = hello_dev
obj-m      := $(TARGET).o

all: default ioctl

default:
        $(MAKE) -C $(KDIR) M=$(PWD) modules

ioctl:  ioctl.h ioctl.c
        gcc ioctl.c -o ioctl
...
```

Интерес такой совместной сборки состоит в том, что и модуль и пользовательские процессы включают (директивой `#include`) одни и те же общие и согласованные определения (пример, в том же архиве `ioctl.tgz`):

```
#include "ioctl.h"
```

Такие файлы содержат общие определения:

ioctl.h :

```
typedef struct _RETURN_STRING {
   char buf[ 160 ];
} RETURN_STRING;
#define IOCTL_GET_STRING _IOR( IOC_MAGIC, 1, RETURN_STRING )
```

Некоторую дополнительную неприятность на этом пути составляет то, что при сборке приложений и модулей (использующих совместные определения) используются разные дефаултные каталоги поиска

10 К этому архиву мы ещё раз вернёмся в самом конце нашего рассмотрения, при анализе экспортирования имён, а пока только отметим в отношении него то, как собираются несколько модулей одновременно.

системных (<...>) файлов определений: /usr/include для процессов, и /lib/modules/`uname -r`/build/include для модулей. Приемлемым решением будет включение в общий включаемый файл фрагмента подобного вида:

```
#ifndef __KERNEL__         // ------------- user space applications
#include <linux/types.h>   // это /usr/include/linux/types.h !
#include <string.h>
...
#else                      // --------------- kernel modules
...
#include <linux/errno.h>
#include <linux/types.h>   // а это /lib/modules/`uname -r`/build/include/linux/types.h
#include <linux/string.h>
...
#endif
```

При всём подобии имён заголовочных файлов (иногда и полном совпадении написания: <linux/types.h>), это будут включения заголовков из совсем разных наборов API (API разделяемых библиотек *.so для пространства пользователя, и API ядра - для модулей). Первый (пользовательский) из этих источников будет обновляться, например, при переустановке в системе новой версии компилятора GCC и комплекта соответствующих ему библиотек (в первую очередь libc.so). Второй (ядерный) из этих источников будет обновляться, например, при обновлении сборки ядра (из репозитария дистрибутива), или при сборке и установке нового ядра из исходных кодов.

Пользовательские библиотеки

В дополнение к набору приложений, обсуждавшихся выше, удобно целый ряд совместно используемых этими приложениями функций собрать в виде единой библиотеки (так устраняется дублирование кода, упрощается внесение изменений, да и вообще улучшается структура проекта). Фрагмент Makefile из архива примеров time.tgz демонстрирует как это записать, не выписывая в явном виде все цели сборки (перечисленные списком в переменной OBJLIST) для каждого такого объектного файла, включаемого в библиотеку (реализующего отдельную функцию библиотеки). В данном случае мы собираем **статическую** библиотеку libdiag.a:

```
LIBTITLE = diag
LIBRARY = lib$(LIBTITLE).a

all:    prog lib

PROGLIST = clock pdelay rtcr rtprd
prog:   $(PROGLIST)

clock:  clock.c
        $(CC) $< -Bstatic -L./ -l$(LIBTITLE) -o $@
...
OBJLIST = calibr.o rdtsc.o proc_hz.o set_rt.o tick2us.o
lib:    $(OBJLIST)

LIBHEAD = lib$(LIBTITLE).h
%.o: %.c $(LIBHEAD)
        $(CC) -c $< -o $@
        ar -r $(LIBRARY) $@
        rm $@
```

Здесь собираются две цели prog и lib, объединённые в одну общую цель all. При желании, статическую библиотеку можно поменять на **динамическую** (разделяемую), что весьма часто востребовано в реальных крупных проектах. При этом в Makefile требуется внести всего незначительные изменения (все остальные файлы проекта остаются в неизменном виде):

```
LIBRARY = lib$(LIBTITLE).so
```

```
...
prog:   $(PROGLIST)
clock:  clock.c
        $(CC) $< -L./ -l$(LIBTITLE) -o $@
...
OBJLIST = calibr.o rdtsc.o proc_hz.o set_rt.o tick2us.o
lib:    $(OBJLIST)

LIBHEAD = lib$(LIBTITLE).h
%.o: %.c $(LIBHEAD)
        $(CC) -c -fpic -fPIC -shared $< -o $@
        $(CC) -shared -o $(LIBRARY) $@
        rm $@
```

Примечание: В случае построения **разделяемой** библиотеки необходимо, кроме того, обеспечить размещение вновь созданной библиотеки (в нашем примере это `libdiag.so`) на путях, где он будет найдена динамическим загрузчиком, размещение «текущий каталог» для этого случая неприемлем: относительные путевые имена не применяются для поиска динамических библиотек. Решается эта задача: манипулированием с переменными окружения `LD_LIBRARY_PATH` и `LD_RUN_PATH`, или с файлом `/etc/ld.so.cache` (файл `/etc/ld.so.conf` и команда `ldconfig`) ..., но это уже вопросы системного администрирования, далеко уводящие нас за рамки предмета рассмотрения.

Как собрать модуль из нескольких объектных файлов?

Соберём (архив `mobj.tgz`) модуль из основного файла `mod.c` и 3-х отдельно транслируемых файлов `mf1.c, mf2.c, mf3.c`, содержащих по одной отдельной функции, экспортируемой модулем (весьма общий случай), это наше первое пересечение с понятием экспорта имён ядра:

mod.c :

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

static int __init init_driver( void ) { return 0; }
static void __exit cleanup_driver( void ) {}
module_init( init_driver );
module_exit( cleanup_driver );
```

mf1.c :

```
#include <linux/module.h>
char *mod_func_A( void ) {
   static char *ststr = __FUNCTION__ ;
   return ststr;
};
EXPORT_SYMBOL( mod_func_A );
```

Файлы `mf2.c, mf3.c` полностью подобны `mf1.c` только имена экспортируемых функций в них заменены, соответственно, на `mod_func_B( void )` и `mod_func_C( void )`, вот здесь у нас впервые появляется макрос-описатель `EXPORT_SYMBOL`.

Заголовочный файл, включаемый в текст модулей:

mf.h :

```
extern char *mod_func_A( void );
extern char *mod_func_B( void );
extern char *mod_func_C( void );
```

Ну и, наконец, в том же каталоге собран второй (тестовый) модуль, который импортирует и вызывает эти три функции как внешние экспортируемые ядром символы:

mcall.c :

```
#include <linux/module.h>
#include "mf.h"
static int __init init_driver( void ) {
   printk( KERN_INFO "start module, export calls: %s + %s + %s\n",
           mod_func_A(), mod_func_B(), mod_func_C() );
   return 0;
}
static void __exit cleanup_driver( void ) {}
module_init( init_driver );
module_exit( cleanup_driver );
```

Самое интересное в этом проекте, это:

Makefile :

```
...
EXTRA_CFLAGS += -O3 -std=gnu89 --no-warnings
OBJS = mod.o mf1.o mf2.o mf3.o
TARGET = mobj
TARGET2 = mcall

obj-m      := $(TARGET).o $(TARGET2).o
$(TARGET)-objs := $(OBJS)

all:
        $(MAKE) -C $(KDIR) M=$(PWD) modules

$(TARGET).o: $(OBJS)
        $(LD) -r -o $@ $(OBJS)
...
```

- привычные из предыдущих примеров `Makefile`, всё те же определения переменных компиляции — опущены.

Теперь мы можем испытывать то, что мы получили:

```
$ nm mobj.ko | grep T
00000000 T cleanup_module
00000000 T init_module
00000000 T mod_func_A
00000010 T mod_func_B
00000020 T mod_func_C
$ sudo insmod ./mobj.ko
$ lsmod | grep mobj
mobj                    1032  0
$ cat /proc/kallsyms | grep mod_func
...
f7f9b000 T mod_func_A   [mobj]
f7f9b010 T mod_func_B   [mobj]
f7f9b020 T mod_func_B   [mobj]
...
$ modinfo mcall.ko
filename:       mcall.ko
license:        GPL
author:         Oleg Tsiliuric <olej@front.ru>
description:    multi jbjects module
srcversion:     5F4A941A9E843BDCFEBF95B
depends:        mobj
vermagic:       2.6.32.9-70.fc12.i686.PAE SMP mod_unload 686
$ sudo insmod ./mcall.ko
$ dmesg | tail -n1
```

```
start module, export calls: mod_func_A + mod_func_B + mod_func_C
```

И в завершение проверим число ссылок модуля, и попытаемся модули выгрузить:

```
$ lsmod | grep mobj
mobj                    1032  1 mcall
$ sudo rmmod mobj
ERROR: Module mobj is in use by mcall
$ sudo rmmod mcall
$ sudo rmmod mobj
```

Рекурсивная сборка

Это вопрос, не связанный непосредственно со сборкой модулей, но очень часто возникающий в проектах, оперирующих с модулями: выполнить сборку (одной и той же цели) во всех включаемых каталогах дерева проекта. Так, например, на каких-то этапах своего развития, архив примеров к этой книге имел вид:

```
$ ls -w80
blkdev  file         load_module  pci    sys_call_table  tree.txt  user_space
dev     first_hello  Makefile     proc   thread          udev
dma     int80        memory       signal  time           usb
exec    IRQ          network      sys     tools          user_io
```

Здесь, за исключением 2-х файлов (`Makefile, tree.txt`), всё остальное — это каталоги, которые, в свою очередь могут содержать каталоги отдельных проектов. Хотелось бы иметь возможность собирать (и очищать от мусора) всю эту иерархию каталогов-примеров одной командой. Для такой цели используем, как вариант, такой сценарий сборки:

Makefile :

```
SUBDIRS = $(shell ls -l | awk '/^d/ { print $$9 }')
all:
        @list='$(SUBDIRS)'; for subdir in $$list; do \
          echo "=============== making all in $$subdir ================="; \
          (cd $$subdir && make && cd ../) \
        done;
install:
        @list='$(SUBDIRS)'; for subdir in $$list; do \
          echo "============= making install in $$subdir ==============="; \
          (cd $$subdir; make install; cd ../) \
        done
uninstall:
        @list='$(SUBDIRS)'; for subdir in $$list; do \
          echo "============= making uninstall in $$subdir ==============="; \
          (cd $$subdir; make uninstall; cd ../) \
        done
clean:
        @list='$(SUBDIRS)'; for subdir in $$list; do \
          echo "=============== making clean in $$subdir ================"; \
          (cd $$subdir && make clean && cd ../) \
        done;
```

Интерес здесь представляет строка, динамически формирующая в переменной `SUBDIRS` список подкаталогов текущего каталога, для каждого из которых потом последовательно выполняется `make` для той же цели, что и исходный вызов. Это хорошо работает в дистрибутиве Fedora, но перестало работать в дистрибутиве Ubuntu. И связано это с разным форматом представления вывода команды `ls`, соответственно (в том же порядке — Fedora, Ubuntu):

```
$ ls -l
итого 100
drwxrwxr-x 7 olej olej 4096 Янв 26 02:36 dev
drwxrwxr-x 2 olej olej 4096 Авг 27 21:57 dma
```

```
...
s ls -l
total 96
drwxrwxr-x 7 user user 4096 2011-07-02 13:11 dev
drwxrwxr-x 2 user user 4096 2011-08-27 21:57 dma
...
```

В данном случае, разница обусловлена различным форматом даты, и различием в числе полей. Для такого случая можно предложить[11] другой вариант (здесь есть пространство для изобретательства) :

Makefile :

```
SUBDIRS = $(shell find . -maxdepth 1 -mindepth 1 -type d -printf "%f\n")

all install uninstall clean disclean:
        @list='$(SUBDIRS)'; for subdir in $$list; do \
           echo "=============== making $@ in $$subdir =================="; \
           (cd $$subdir && make $@) \
        done
```

Это мелкая техническая деталь, но описана она здесь для того, чтобы предупредить о возможности подобных артефактов, к ним нужно быть готовым, и они, обычно, легко разрешаются.

Инсталляция модуля

Инсталляция модуля, если говорить о **инсталляции** как о создании **цели** в `Makefile`, должна состоять в том, чтобы:

а). скопировать собранный модуль (`*.ko`) в его местоположение в иерархии модулей в исполняющейся файловой системе, часто это, например, каталог `/lib/modules/`uname -r`/misc`, туда, где операционная система ищет доступные ей модули ядра;

б). обновить информацию о взаимных зависимостях модулей (в связи с добавлением нового), что делает утилита `depmod`.

Но если создаётся цель в `Makefile` инсталляции модуля, то обязательно должна создаваться и обратная цель **деинсталляции**: лучше не иметь оформленной возможности инсталлировать модуль (оставить это на ручные операции), чем иметь инсталляцию не имея деинсталляции!

Задачи

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

```
TARGET = hello_printk
obj-m      := $(TARGET).o
default:
       $(MAKE) -C $(KDIR) M=$(PWD) modules
```

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

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

11 Предложен одним из читателей рукописи книги.

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

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

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

5. Напишите модуль, который должен зафиксировать **в системный журнал** значение счётчика системного таймера ядра `jiffies`, и сразу же завершается. Постарайтесь записать код модуля как можно более кратким.

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

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

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

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

4. Драйверы: символьные устройства

Интерфейс устройства в Linux

Смысл операций с интерфейсом `/dev` состоит в связывании именованного устройства в каталоге `/dev` с разрабатываемым модулем, а в самом коде модуля реализации разнообразных операций на этом устройстве (таких как `open()`, `read()`, `write()` и множества других). В таком качестве модуль ядра и называется драйвером устройства. Некоторую сложность в проектировании драйвера создаёт то, что для этого действа предлагаются несколько альтернативных, совершенно исключающих друг друга техник написания. Связано это с давней историей развития подсистемы `/dev` (одна из самых старых подсистем UNIX и Linux), и с тем, что на протяжении этой истории отрабатывались несколько отличающихся моделей реализации, а удачные решения закреплялись как альтернативы. В любом случае, при проектировании нового драйвера предстоит ответить для себя на **три группы** вопросов (по каждому из них возможны альтернативные ответы):

1. Каким способом драйвер будет **регистрироваться** в системе (под парой номеров `major` и `minor`), как станет известно системе, что у неё появился в распоряжении новый драйвер и новое устройство?
2. Каким образом драйвер создаёт (или использует созданное внешними средствами) **имя** соответствующего ему устройства в каталоге `/dev`, и как он (драйвер) **связывает** это имя с `major` и `minor` номерами этого устройства?
3. После того, как драйвер увязан с устройством, какие будут использованы особенности в реализации основных **операций обслуживания** устройства (`open(), read(), ...`)?

Но прежде, чем перейти к созданию интерфейса устройства, очень коротко вспомним философию устройств, общую не только для Linux, но и для всех UNIX/POSIX систем. Каждому устройству в системе соответствует имя этого устройства в каталоге `/dev`. Каждое именованное устройство в Linux однозначно характеризуется двумя (байтовыми: 0...255) номерами: старшим номером (`major`) — номером отвечающим за отдельный класс устройств, и младшим номером (`minor`) — номером конкретного устройства внутри своего класса. Например, для диска SATA:

```
$ ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 Июн 16 11:03 /dev/sda
brw-rw---- 1 root disk 8, 1 Июн 16 11:04 /dev/sda1
brw-rw---- 1 root disk 8, 2 Июн 16 11:03 /dev/sda2
brw-rw---- 1 root disk 8, 3 Июн 16 11:03 /dev/sda3
```

Здесь 8 — это старший номер для любого из дисков SATA в системе, а 2 — это младший номер для 2-го (`sda2`) раздела 1-го (`sd`**`a`**) диска SATA. Связать модуль с именованным устройством и означает установить ответственность модуля за операции с устройством, характеризующимся парой `major/minor`. В таком качестве модуль называют драйвером устройства. Связь номеров устройств с конкретными типами оборудования — жёстко регламентирована (особенно в отношении старших номеров), и определяется содержимым файла в исходных кодах ядра: `Documentation/devices.txt` (больше 100Kb текста, приведено в каталоге примеров `/dev`).

Номера `major` для **символьных** и **блочных** устройств составляют совершенно различные пространства номеров и могут использоваться независимо, пример чему — набор разнообразных системных устройств:

```
$ ls -l /dev | grep ' 1,'
...
crw-r-----  1 root kmem        1,   1 Июн 26 09:29 mem
crw-rw-rw-  1 root root        1,   3 Июн 26 09:29 null
...
crw-r-----  1 root kmem        1,   4 Июн 26 09:29 port
brw-rw----  1 root disk        1,   0 Июн 26 09:29 ram0
brw-rw----  1 root disk        1,   1 Июн 26 09:29 ram1
```

```
brw-rw----  1 root disk       1,  10 Июн 26 09:29 ram10
...
brw-rw----  1 root disk       1,  15 Июн 26 09:29 ram15
brw-rw----  1 root disk       1,   2 Июн 26 09:29 ram2
brw-rw----  1 root disk       1,   3 Июн 26 09:29 ram3
...
crw-rw-rw-  1 root root       1,   8 Июн 26 09:29 random
crw-rw-rw-  1 root root       1,   9 Июн 26 09:29 urandom
crw-rw-rw-  1 root root       1,   5 Июн 26 09:29 zero
```

Примечание: За времена существования систем UNIX сменилось несколько парадигм присвоения номеров устройствам и их классам. С этим и связано наличие заменяющих друг друга нескольких альтернативных API связывания устройств с модулем в Linux. Самая ранняя парадигма (унаследованная из ядер 2.4 мы её рассмотрим последней) утверждала, что старший `major` номер присваивается классу устройств, и за все 255 `minor` номеров отвечает модуль этого класса и только он (модуль) оперирует с этими номерами. В этом варианте не может быть двух классов устройств (модулей ядра), обслуживающих одинаковые значения `major`. Позже (ядра 2.6) модулю (и классу устройств) отнесли **фиксированный диапазон** ответственности этого модуля, таким образом для устройств с одним `major`, устройства с `minor`, скажем, 0...63 могли бы обслуживаться модулем `xxx1.ko` (и составлять отдельный класс), а устройства с `minor` 64...127 — другим модулем `xxx2.ko` (и составлять совершенно другой класс). Ещё позже, когда под статические номера устройств, определяемые в `devices.txt`, стало катастрофически не хватать номеров, была создана модель динамического распределения номеров, поддерживающая её файловая система `sysfs`, и обеспечивающий работу `sysfs` в пользовательском пространстве программный проект `udev`.

Символьные устройства

Практически вся полезная работа модуля в интерфейсе `/dev` (точно так же, как и в интерфейсах `/proc` и `/sys`, рассматриваемых позже), реализуется через таблицу (структуру) файловых операций `file_operations`, которая определена в файле `<linux/fs.h>` и содержит указатели на функции драйвера, которые отвечают за выполнение различных операций с устройством. Эта большая структура настолько важна, что она стоит того, чтобы быть приведенной полностью (в том виде, как это имеет место в ядре 2.6.37):

```
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 (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
     ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
     int (*readdir) (struct file *, void *, filldir_t);
     unsigned int (*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 *);
     int (*open) (struct inode *, struct file *);
     int (*flush) (struct file *, fl_owner_t id);
     int (*release) (struct inode *, struct file *);
     int (*fsync) (struct file *, int datasync);
     int (*aio_fsync) (struct kiocb *, int datasync);
     int (*fasync) (int, struct file *, int);
     int (*lock) (struct file *, int, struct file_lock *);
     ssize_t (*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 **);
};
```

Если мы переопределяем в своём коде модуля какую-то из функций таблицы, то эта функция становится обработчиком, вызываемым для обслуживания этой операции. Если мы не переопределяем операцию, то для большинства операций (`llseek`, `flush` и др.) используется **обработчик по умолчанию**. Такой обработчик может вообще не выполнять никаких действий, или выполнять некоторый минимальный набор простейших действий. Такая ситуация имеет место достаточно часто, например, в отношении операций `open` и `release` на устройстве, но тем не менее устройства замечательно открываются и закрываются. Но для некоторых операций (`mmap` и др.) обработчик по умолчанию будет всегда возвращать код ошибки (not implemented yet), и поэтому он имеет смысл только когда он переопределён.

Ещё одна структура, которая менее значима, чем `file_operations`, но также широко используется:

```
struct inode_operations {
        int (*create) (struct inode *, struct dentry *, int, struct nameidata *);
        struct dentry * (*lookup) (struct inode *, struct dentry *, struct nameidata *);
        int (*link) (struct dentry *, struct inode *, struct dentry *);
        int (*unlink) (struct inode *, struct dentry *);
        int (*symlink) (struct inode *, struct dentry *, const char *);
        int (*mkdir) (struct inode *, struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *, struct dentry *, int, dev_t);
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *);
...
}
```

Примечание: Отметим, что структура `inode_operations` соответствуют системным вызовам, которые оперируют с устройствами по их путевым именам, а структура `file_operations` — системным вызовам, которые оперируют с таким представлением файлов устройств, более понятным программистам, как файловый дескриптор. Но ещё важнее то, что имя ассоциируется с устройством всегда одно, а файловых дескрипторов может быть ассоциировано много. Это имеет следствием то, что указатель структуры `inode_operations`, передаваемый в операцию (например `int (*open) (struct inode*, struct file*)`) будет всегда один и тот же (до выгрузки модуля), а вот указатель структуры `file_operations`, передаваемый в ту же операцию, будет меняться при каждом новом открытии устройства. Вытекающие отсюда эффекты мы увидим в примерах в дальнейшем.

Варианты реализации

Возвращаемся к регистрации драйвера в системе. Некоторую путаницу в этом вопросе создаёт именно то, что, во-первых, это может быть проделано несколькими разными, альтернативными способами, появившимися в разные годы развития Linux, а, во-вторых, то, что в каждом из этих способов, если вы уже остановились на каком-то, нужно строго соблюсти последовательность нескольких предписанных шагов, характерных именно для этого способа. Именно на этапе связывания устройства и возникает, отмечаемое многими, изобилие операторов `goto`, когда при неудаче очередного шага установки приходится последовательно отменять результаты всех проделанных шагов. Для регистрации устройства и создания связи (интерфейса) модуля к `/dev`, в разное время и для разных целей, было создано несколько альтернативных (во многом замещающих друг друга) техник написания кода. Мы рассмотрим далее некоторые из них:

1. Новый способ (2.6), использующий структуру `struct cdev` (`<linux/cdev.h>`), позволяющий динамически выделять старший номер из числа свободных, и увязывать с ним ограниченный диапазон младших номеров.

2. Способ полностью динамического создания именованных устройств, так называемая техника misc drivers (miscellaneous — разнородных устройств).

3. Старый способ (2.4, использующий `register_chrdev()`), статически связывающий модуль со старшим номером, тем самым отдавая под контроль модуля **весь** диапазон допустимых младших номеров (0...255); название способа как старый не отменяет его актуальность и на сегодня.

Кроме нескольких различных техник **регистрации** устройства, независимо существует также несколько различающихся вариантов того, как имя устройства **создаётся** в `/dev` и связывается с `major` и `minor` номерами для этого устройства. Произведение этих двух подмножеств выбора и создаёт пространство альтернатив возможностей для разработчика драйвера.

Ручное создание имени

Наш первый вариант модуля символьного устройства, предоставляет пользователю только операцию чтения из устройства (операция записи реализуется абсолютно симметрично, и не реализована, чтобы не перегружать текст; аналогичная реализация будет показана на интерфейсе `/proc`). Кроме того, поскольку мы собираемся реализовать целую группу альтернативных драйверов интерфейса `/dev`, то сразу вынесем общую часть (главным образом, реализацию функции чтения) в отдельный включаемый файл (это даст нам большую экономию объёма изложения) :

dev.h :

```
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/module.h>
#include <asm/uaccess.h>

MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_VERSION( "6.3" );

static char *hello_str = "Hello, world!\n";         // buffer!

static ssize_t dev_read( struct file * file, char * buf,
                         size_t count, loff_t *ppos ) {
   int len = strlen( hello_str );
   printk( KERN_INFO "=== read : %d\n", count );
   if( count < len ) return -EINVAL;
   if( *ppos != 0 ) {
      printk( KERN_INFO "=== read return : 0\n" );  // EOF
      return 0;
   }
   if( copy_to_user( buf, hello_str, len ) ) return -EINVAL;
   *ppos = len;
   printk( KERN_INFO "=== read return : %d\n", len );
   return len;
}

static int __init dev_init( void );
module_init( dev_init );

static void __exit dev_exit( void );
module_exit( dev_exit );
```

Тогда первый вариант драйвера (архив `cdev.tgz`), использующий структуру `struct cdev`, будет иметь вид (рассмотренный общий файл `dev.h` включён как преамбула этого кода, так будет и в дальнейших примерах):

fixdev.c :

```
#include <linux/cdev.h>
#include "../dev.h"
```

```
static int major = 0;
module_param( major, int, S_IRUGO );

#define EOK 0
static int device_open = 0;

static int dev_open( struct inode *n, struct file *f ) {
   if( device_open ) return -EBUSY;
   device_open++;
   return EOK;
}

static int dev_release( struct inode *n, struct file *f ) {
   device_open--;
   return EOK;
}

static const struct file_operations dev_fops = {
   .owner = THIS_MODULE,
   .open = dev_open,
   .release = dev_release,
   .read  = dev_read,
};

#define DEVICE_FIRST  0
#define DEVICE_COUNT  3
#define MODNAME "my_cdev_dev"

static struct cdev hcdev;

static int __init dev_init( void ) {
   int ret;
   dev_t dev;
   if( major != 0 ) {
      dev = MKDEV( major, DEVICE_FIRST );
      ret = register_chrdev_region( dev, DEVICE_COUNT, MODNAME );
   }
   else {
      ret = alloc_chrdev_region( &dev, DEVICE_FIRST, DEVICE_COUNT, MODNAME );
      major = MAJOR( dev );  // не забыть зафиксировать!
   }
   if( ret < 0 ) {
      printk( KERN_ERR "=== Can not register char device region\n" );
      goto err;
   }
   cdev_init( &hcdev, &dev_fops );
   hcdev.owner = THIS_MODULE;
   ret = cdev_add( &hcdev, dev, DEVICE_COUNT );
   if( ret < 0 ) {
      unregister_chrdev_region( MKDEV( major, DEVICE_FIRST ), DEVICE_COUNT );
      printk( KERN_ERR "=== Can not add char device\n" );
      goto err;
   }
   printk( KERN_INFO "=========== module installed %d:%d ==============\n",
           MAJOR( dev ), MINOR( dev ) );
err:
   return ret;
}
```

```
static void __exit dev_exit( void ) {
   cdev_del( &hcdev );
   unregister_chrdev_region( MKDEV( major, DEVICE_FIRST ), DEVICE_COUNT );
   printk( KERN_INFO "=============== module removed =================\n" );
}
```

Здесь показан только один (для краткости) уход на метку ошибки выполнения (`err:`) на шаге инсталляции модуля, в коде реальных модулей вы увидите целые цепочки подобных конструкций для отработки возможных ошибок на каждом шаге инсталляции.

Этот драйвер умеет пока только тупо выводить по запросу `read()` фиксированную строку из буфера, но для изучения структуры драйвера этого пока достаточно. Здесь используется такой, уже обсуждавшийся ранее механизм, как указание параметра загрузки модуля: либо система сама выберет номер `major` для нашего устройства, если мы явно его не указываем в качестве параметра, либо система принудительно использует заданный параметром номер, даже если его значение неприемлемо и конфликтует с уже существующими номерами устройств в системе.

Дальше, путём экспериментирования, мы проверяем работоспособность написанного модуля, и эти эксперименты очень много проясняют относительно драйверов устройств Linux:

```
$ sudo insmod fixdev.ko major=250
insmod: error inserting 'fixdev.ko': -1 Device or resource busy
$ dmesg | grep ===
=== Can not register char device region
$ ls -l /dev | grep 250
crw-rw----  1 root root      250,   0 Янв 22 11:49 hidraw0
crw-rw----  1 root root      250,   1 Янв 22 11:49 hidraw1
```

В этот раз нам не повезло: наугад выбранный номер `major` для нашего устройства оказывается уже занятым другим устройством в системе. В конечном итоге, мы находим первый свободный `major` в системе (в вашей системе он может быть совершенно другой):

```
$ sudo insmod fixdev.ko major=255
$ ls -l /dev | grep 255
$ dmesg | grep ===
=========== module installed 255:0 ==============
$ lsmod | grep fix
fixdev                  1384  0
$ cat /proc/devices | grep my_
255 my_cdev_dev
```

Драйвер успешно установлен! Но этого мало для работы с ним, и здесь всплывает важная особенность реализации подсистемы устройств: драйвер оперирует с устройством как с парой **номеров** `major/minor`, а все команды GNU и функции POSIX API оперирует с устройством как с **именем** в каталоге `/dev`. Для работы с устройством мы должны установить взаимно однозначное соответствие между `major/minor` и имени устройства. Пока мы сделаем такое именованное устройство вручную, создав **произвольное** имя, и связав его с `major/minor`, обслуживаемыми модулем:

```
$ sudo mknod -m0666 /dev/abc c 255 0
$ cat /dev/abc
Hello, world!
$ sudo rm /dev/abc
```

Ещё убедительнее иллюстрирует то, что первично, а что вторично с позиции подсистемы устройств и драйвера — это создание имени устройства не в каталоге `/dev`, а ... в текущем рабочем каталоге, где ему совсем не место:

```
$ sudo mknod -m0666 ./z0 c 255 0
$ ls -l | grep ^c
crw-rw-rw- 1 root root 255, 0 Янв 22 16:43 z0
$ cat ./z0
Hello, world!
```

```
$ sudo rm ./z0
```

Экспериментируя с модулем, не забываем его периодически выгружать время от времени, перед очередным туром экспериментов, причём, процесс этот может дать тоже много поучительного:

```
$ cat /dev/abc
Hello, world!
$ sudo rmmod fixdev
$ lsmod | grep fix
$ cat /dev/abc
cat: /dev/abc: Нет такого устройства или адреса
$ ls -l /dev/abc
crw-rw-rw- 1 root root 255, 0 Янв 22 14:13 /dev/abc
```

Текст сообщение об ошибке чтения здесь не соответствует действительности: устройство существует, но нет модуля, обслуживающего данное устройство, разрушена связь между его именем и поддержкой соответствующих номеров устройства со стороны ядра.

Особое внимание обращаем на то, каким образом функция, обрабатывающая запросы `read()`, сообщает вызывающей программе об исчерпании потока доступных данных (признак `EOF`) — это важнейшая функция операции чтения. Смотрим на то, как утилита `cat` запрашивала данные, и что она получала в результате:

```
$ cat /dev/abc
Hello, world!
$ dmesg | tail -n20 | grep ===
=== read : 32768
=== read return : 14
=== read : 32768
=== read return : 0
```

И, наконец, убеждаемся, что созданный драйвер поддерживает именно заказанный ему диапазон `minor` номеров, не больше, но и не меньше (0-2 в показанном примере):

```
$ sudo insmod fixdev.ko major=255
$ sudo mknod -m0666 /dev/abc2 c 255 2
$ sudo mknod -m0666 /dev/abc3 c 255 3
$ ls -l /dev | grep 255
crw-rw-rw-  1 root root      255,   2 Янв 22 14:37 abc2
crw-rw-rw-  1 root root      255,   3 Янв 22 14:37 abc3
$ cat /dev/abc2
Hello, world!
$ cat /dev/abc3
cat: /dev/abc3: Нет такого устройства или адреса
```

А вот так происходит запуск без параметра в командной строке, когда номер устройства модуль запрашивает у ядра динамически:

```
$ sudo insmod fixdev.ko
$ dmesg | grep ===
========== module installed 249:0 ==============
```

Самого такого имени устройства (с `major` равным 249) у нас, естественно, пока не существует в `/dev` (мы не сможем пока воспользоваться этим модулем, даже если он загружен). Это та вторая группа вопросов, которая упоминалась раньше: как создаётся устройство с заданными номерами? Пока мы создадим такое символьное устройство вручную, связывая его со старшим номером, обслуживаемым модулем, и проверим работу модуля:

```
$ cat /proc/devices | grep my_
249 my_cdev_dev
$ sudo mknod -m0666 /dev/z0 c 249 0
$ ls -l /dev | grep 249
crw-rw-rw-  1 root root      249,   0 Янв 22 13:29 z0
$ cat /dev/z0
Hello, world!
$ sudo rm /dev/z0
```

```
$ cat /dev/z0
cat: /dev/z0: Нет такого файла или каталога
```

Старший номер для устройства был выбран системой динамически, из соображений, чтобы а). этот `major` был не занят в системе, б). для этого `major` был не занят и запрашиваемый **диапазон** `minor`. Но самого имени **для** такого устройства (с `major` равным 249) в системе не существовало, и мы были вынуждены создать его сами (команда `mknod`).

Примечание: Ручное создание **статического** имени в `/dev` с помощью команды `mknod` — это традиционный (и рудиментарный) подход к решению проблемы. Этот способ работает во всех системах UNIX и один из самых старых механизмов UNIX, существующий несколько десятков лет. На практике в современном Linux такой способ груб и не гибок. В системе существует целый ряд механизмов **динамического** создания имен устройств и связывания их с номерами. Но каждый из них базируется на автоматическом, скрытом выполнении того статического механизма, который рассмотрен выше. Для тонкого использования на практике современных динамических механизмов крайне полезно прежде поупражняться в ручном создании имён устройств, чтобы понимать всю подноготную динамического управления именами устройств.

Использование udev

Со времени создания файловой системы `/sys` (`sysfs`) каждое действие, связанное с устройствами (горячее подключение, отключение) или модулями ядра (загрузка, выгрузка) вызывают целый всплеск асинхронных **широковещательных** уведомлений через дэйтаграммный сокет, специально для этого случая спроектированного протокола обмена — `netlink`. Для большей ясности, **любой** пользовательский процесс может создать такой сокет вызовом вида:

```
fd = socket( AF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT );
```

(Отметим, что такой вид сокета, и обмен через него, практичекси нигде не упоминаются и не описаны в публикациях по сетевому программированию — это исключительно «придумка» Linux.)

И любой пользовательский процесс может обрабатывать информацию, передаваемую **из ядра** через сокет `netlink` и реагировать на эту информацию. Но программа пользовательского пространства `udevd` делает это по умолчанию, даже если никто другой это не обрабатывает. В этом и смысл **широковещания**.

После загрузки обсуждавшегося выше модуля `fixdev.ko`, мы можем наблюдать, например:

```
$ cat /proc/devices | grep my_cdev_dev
246 my_cdev_dev
$ ls -l /sys/module/fixdev
итого 0
-r--r--r-- 1 root root 4096 апр 10 12:29 coresize
drwxr-xr-x 2 root root    0 апр 10 12:29 holders
-r--r--r-- 1 root root 4096 апр 10 12:56 initsize
-r--r--r-- 1 root root 4096 апр 10 12:54 initstate
drwxr-xr-x 2 root root    0 апр 10 12:54 notes
drwxr-xr-x 2 root root    0 апр 10 12:54 parameters
-r--r--r-- 1 root root 4096 апр 10 12:29 refcnt
drwxr-xr-x 2 root root    0 апр 10 12:54 sections
-r--r--r-- 1 root root 4096 апр 10 12:54 srcversion
-r--r--r-- 1 root root 4096 апр 10 12:54 taint
--w------- 1 root root 4096 апр 10 12:28 uevent
-r--r--r-- 1 root root 4096 апр 10 12:54 version
$ cat /sys/module/fixdev/parameters/major
246
```

Характерно, что здесь мы можем извлечь значение параметра модуля `major`, даже в том случае, когда мы загружаем модуль без явного указания этого параметра при загрузке (что и было сделано специально в показанном примере). Таким образом через `/sys` мы можем извлекать все интересующие нас параметры

загрузки модулей.

Но возвратимся к асинхронным уведомлениям через дэйтаграммный сокет протокола `netlink`. Наблюдать содержимое уведомлений передаваемых от ядра к демону `udevd` мы можем, запустив на время наблюдаемого события (подключение-отключение устройства, загрузка-выгрузка модуля) **в отдельном терминале** программу:

```
$ udevadm monitor --property
monitor will print the received events for:
UDEV - the event which udev sends out after rule processing
KERNEL - the kernel uevent

KERNEL[21383.789383] add      /module/fixdev (module)
ACTION=add
DEVPATH=/module/fixdev
SEQNUM=2324
SUBSYSTEM=module

UDEV  [21383.790971] add      /module/fixdev (module)
ACTION=add
DEVPATH=/module/fixdev
SEQNUM=2324
SUBSYSTEM=module
USEC_INITIALIZED=383789737
...
```

Здесь показан именно момент загрузки тестового модуля `fixdev.ko`. (Это напоминает по смыслу аналогию, как то же делает и показывает общеизвестный сетевой снифер `tcpdump` для трафика IP протокола).

Получив подобного рода уведомление от ядра демон `udevd` (но и **любой** процесс пространства пользователя) имеет возможность проанализировать параметры в этих сообщениях (`ACTION`, `SUBSYSTEM`, ...) дерево `/sys` (на основе полученных из дэйтаграммного уведомления параметров), а далее **динамически** (по событию) создать соответствующее имя в `/dev`. Поскольку это весьма общий механизм, то разработан специальный синтаксис **правил действий** для **событий** в `/sys`. Такие правила записываются отдельными файлами `.rules` в каталоге `/etc/udev/rules.d`. Если выполняются **условия** (ключ соответствия, match key — они описываются с знаком ==) возникшего события (на основании параметров в сообщении `netlink`), то `udevd` (говорят: подсистема `udev`) предпринимает действия, описанные в **правиле** (ключ назначения, assignment key — они описываются с знаком =). Каждое правило записывается одной строкой (даже для весьма длинных строк не предусмотрено переносов). Например, запуская выбранную программу (возможно с параметрами):

```
SUBSYSTEM=="module", ACTION=="add", DEVPATH=="/module/fixdev", PROGRAM="/bin/my_fix_ctl try"
```

Или непосредственно создавая имя устройства в `/dev` :

```
SUBSYSTEM=="module", ACTION=="add", DEVPATH=="/module/fixdev", SYMLINK+="abc2"
```

Совершенно аналогичным образом добавляется правило на удаление устройства при выгрузке модуля (`ACTION=="remove"`). Это делается или отдельной строкой (в том же файле), или отдельным файлом в `/etc/udev/rules.d` каталоге.

Ещё одним альтернативным способом извлечения параметров **события** (а это может оказаться не так просто) для успешного написания эффективных **правил** `udev`, может быть (при загруженном модуле ядра):

```
$ udevadm info --path=/module/fixdev --export
P: /module/fixdev
E: DEVPATH=/module/fixdev
E: SUBSYSTEM=module
```

Подсистема `udev` всё ещё находится в активном развитии, хотя основные принципы её и устоялись. Тем не менее, **синтаксис** составления её правил всё ещё изменяется (публикации 2012 года, например, описывают другой синтаксис, относительно 2014-го, времени написания этих строк — ними руководствоваться нельзя). Поэтому относительно синтаксиса записи правил нужно `udev` справиться накануне их написания, см.:

```
$ man 7 udev
...
```

Но общая схема, при всех детализациях синтаксиса записи правил, остаётся неизменной.

Динамические имена

Вариацией на тему использования того же API будет вариант предыдущего модуля (в том же архиве `cdev.tgz`), но динамически создающий имя устройства в каталоге `/dev` с заданным старшим и младшим номером (это обеспечивается уже использованием механизмов системы `sysfs`, рассмотренных в ручном режиме ранее). Ниже показаны только принципиальные отличия (дополнения) относительно предыдущего варианта:

dyndev.c :

```
#include <linux/device.h>
#include <linux/version.h>      /* LINUX_VERSION_CODE */
#include <linux/init.h>
...
#define DEVICE_FIRST  0         // первый minor
#define DEVICE_COUNT  3         // число поддерживаемых minor
#define MODNAME "my_dyndev_mod"

static struct cdev hcdev;
static struct class *devclass;

static int __init dev_init( void ) {
   int ret, i;
   dev_t dev;
   if( major != 0 ) {
      dev = MKDEV( major, DEVICE_FIRST );
      ret = register_chrdev_region( dev, DEVICE_COUNT, MODNAME );
   }
   else {
      ret = alloc_chrdev_region( &dev, DEVICE_FIRST, DEVICE_COUNT, MODNAME );
      major = MAJOR( dev );  // не забыть зафиксировать!
   }
   if( ret < 0 ) {
      printk( KERN_ERR "=== Can not register char device region\n" );
      goto err;
   }
   cdev_init( &hcdev, &dev_fops );
   hcdev.owner = THIS_MODULE;
   ret = cdev_add( &hcdev, dev, DEVICE_COUNT );
   if( ret < 0 ) {
      unregister_chrdev_region( MKDEV( major, DEVICE_FIRST ), DEVICE_COUNT );
      printk( KERN_ERR "=== Can not add char device\n" );
      goto err;
   }
   devclass = class_create( THIS_MODULE, "dyn_class" ); /* struct class* */
   for( i = 0; i < DEVICE_COUNT; i++ ) {
#define DEVNAME "dyn"
      dev = MKDEV( major, DEVICE_FIRST + i );
#if LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,26)
/* struct device *device_create( struct class *cls, struct device *parent,
                                dev_t devt, const char *fmt, ...); */
      device_create( devclass, NULL, dev, "%s_%d", DEVNAME, i );
#else              // прототип device_create() изменился!
/* struct device *device_create( struct class *cls, struct device *parent,
```

```
                                dev_t devt, void *drvdata, const char *fmt, ...); */
      device_create( devclass, NULL, dev, NULL, "%s_%d", DEVNAME, i );
#endif
   }
   printk( KERN_INFO "======== module installed %d:[%d-%d] ===========\n",
           MAJOR( dev ), DEVICE_FIRST, MINOR( dev ) );
err:
   return ret;
}

static void __exit dev_exit( void ) {
   dev_t dev;
   int i;
   for( i = 0; i < DEVICE_COUNT; i++ ) {
      dev = MKDEV( major, DEVICE_FIRST + i );
      device_destroy( devclass, dev );
   }
   class_destroy( devclass );
   cdev_del( &hcdev );
   unregister_chrdev_region( MKDEV( major, DEVICE_FIRST ), DEVICE_COUNT );
   printk( KERN_INFO "=============== module removed ==================\n" );
}
```

Здесь создаётся класс устройств (с именем "dyn_class") в терминологии sysfs, а затем внутри него нужное **число** устройств, исходя из диапазона minor.

Примечание: Обращаем внимание на то, что прототип device_create() изменился с версии ядра 2.6.26, но поскольку он описан как функция с переменным числом аргументов (подобно sprintf()), то сделано это изменение настолько неудачно (из за совпадения типов параметров drvdata и fmt — неконтролируемые указатели), что код предыдущих версий будет **нормально компилироваться** (даже без предупреждений!), но не будет выполняться, или будет выполняться не тем образом, который ожидается.

Теперь не будет необходимости вручную создавать имя устройства в /dev и отслеживать соответствие его номеров — соответствующее имя возникает после загрузки модуля, и так же ликвидируется после выгрузки модуля:

```
$ ls -l /dev/dyn*
ls: невозможно получить доступ к /dev/dyn*: Нет такого файла или каталога
$ sudo insmod dyndev.ko
$ dmesg | tail -n30 | grep ==
======== module installed 249:[0-2] ===========
$ ls -l /dev/dyn*
crw-rw---- 1 root root 249, 0 Янв 22 18:09 /dev/dyn_0
crw-rw---- 1 root root 249, 1 Янв 22 18:09 /dev/dyn_1
crw-rw---- 1 root root 249, 2 Янв 22 18:09 /dev/dyn_2
$ cat /dev/dyn_2
Hello, world!
$ ls /sys/class/d*
...
/sys/class/dyn_class:
dyn_0  dyn_1  dyn_2
$ tree /sys/class/dyn_class/dyn_0
/sys/class/dyn_class/dyn_0
├── dev
├── power
│   └── wakeup
├── subsystem -> ../../../../class/dyn_class
└── uevent
$ cat /proc/modules | grep dyn
```

```
dyndev 1480 0 - Live 0xf88de000
$ cat /proc/devices | grep dyn
249 my_dyndev_mod
$ sudo rmmod dyndev
$ ls -l /dev/dyn*
ls: невозможно получить доступ к /dev/dyn*: Нет такого файла или каталога
```

Этот же модуль может также использоваться для создания диапазона устройств с принудительным указанием `major` (если использование этого номера возможно с точки зрения системы), но с динамическим созданием имён таких устройств:

```
$ sudo insmod dyndev.ko major=260
$ ls -l /dev | grep 260
crw-rw----  1 root root      260,   0 Янв 22 18:57 dyn_0
crw-rw----  1 root root      260,   1 Янв 22 18:57 dyn_1
crw-rw----  1 root root      260,   2 Янв 22 18:57 dyn_2
$ cat /dev/dyn_1
Hello, world!
$ sudo rmmod dyndev
$ ls -l /dev | grep 260
$
```

В версии ядра 2.6.32 всё будет происходить в точности как описано выше. Но в версии ядра 3.13 (я не могу пока сказать по какой точно версии здесь проходит раздел) прямое чтение не удастся из-за прав (флагов доступа), с которым создаются устройства в `/dev`:

```
$ cat /dev/dyn_2
cat: /dev/dyn_2: Отказано в доступе
$ ls -l /dev/dyn*
crw------- 1 root root 246, 0 апр 22 13:45 /dev/dyn_0
crw------- 1 root root 246, 1 апр 22 13:45 /dev/dyn_1
crw------- 1 root root 246, 2 апр 22 13:45 /dev/dyn_2
$ sudo cat /dev/dyn_2
Hello, world!
```

Легко убедиться, что эта сложность снимается изменением флагов, но это не удобный ручной способ:

```
$ sudo chmod 0444 /dev/dyn*
$ ls -l /dev/dyn*
cr--r--r-- 1 root root 246, 0 апр 22 13:45 /dev/dyn_0
cr--r--r-- 1 root root 246, 1 апр 22 13:45 /dev/dyn_1
cr--r--r-- 1 root root 246, 2 апр 22 13:45 /dev/dyn_2
$ cat /dev/dyn_2
Hello, world!
```

В обсуждениях неоднократно указывается, что изменение флагов устройства можно (нужно?) устанавливать (изменять) в правилах подсистемы `udev` для этих устройств. Но тогда мы можем вообще и создавать устройства правилами, не прибегая к услугам `device_create()`. Наверняка, флаги создания имени устройства по умолчанию могут переопределяться где-то в атрибутной записи ... , но пока я не готов сказать где именно.

Такое динамическое создание устройств сильно упрощает работу над драйвером. Но всегда ли хорош такой способ распределения номеров устройств? Всё зависит от решаемой задачи. Если номера реального **физического устройства** в системе «гуляют» от одного компьютера к другому, и даже при изменениях в конфигурациях системы, то это вряд ли понравится разработчикам этого устройства. С другой стороны, во множестве задач удобно создавать псевдоустройства, некоторые моделирующие сущности для каналов ввода вывода. Для таких случаев совершенно уместным будет полностью динамическое распределение параметров таких устройств. Одним из лучших иллюстрирующих примеров, поясняющих сказанное, есть, ставший уже стандартом де-факто, интерфейс zaptel/DAHDI к оборудованию передачи VoIP, используемый во многих проектах коммутаторов IP-телефонии: Asterisk, FreeSWITCH, YATE, ... В архитектуре этой драйверной подсистемы для синхронных цифровых линий связи E1/T1/J1 (и E3/T3/J3) создаётся много (возможно, до

нескольких сот) фиктивных устройств-имён в `/dev`, взаимно-однозначно соответствующих **виртуальным** каналам уплотнения реальных линий связи (тайм-слотам). Вся дальнейшая работа с созданными динамическими именами устройств обеспечивается традиционными `read()` или `write()`, в точности так, как это делается с реальным физическим оборудованием.

Разнородные (смешанные) устройства

Практика динамически перераспределяемых псевдоустройств приобрела настолько широкое распространение, что для упрощения реализации таких устройств был предложен специальный интерфейс (`<linux/miscdevice.h>`). Эта техника регистрации драйвера устройства часто называют в литературе как misc drivers (miscelleneous, интерфейс смешанных, разнородных устройств). Это **самая простая** в кодировании техника регистрации устройства. Каждое такое устройство создаётся с единым `major` значением 10, но может выбирать свой уникальный `minor` (либо задаётся принудительно, либо устанавливается системой). В этом варианте драйвер регистрируется и создаёт символическое имя устройства в `/dev` одним единственным вызовом `misc_register()` (архив `misc.tgz`):

misc_dev.c :

```
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include "../dev.h"

static int minor = 0;
module_param( minor, int, S_IRUGO );

static const struct file_operations misc_fops = {
   .owner  = THIS_MODULE,
   .read   = dev_read,
};

static struct miscdevice misc_dev = {
   MISC_DYNAMIC_MINOR,    // автоматически выбираемое
   "own_misc_dev",
   &misc_fops
};

static int __init dev_init( void ) {
   int ret;
   if( minor != 0 ) misc_dev.minor = minor;
   ret = misc_register( &misc_dev );
   if( ret ) printk( KERN_ERR "=== Unable to register misc device\n" );
   return ret;
}

static void __exit dev_exit( void ) {
   misc_deregister( &misc_dev );
}
```

Вот, собственно, и весь код всего драйвера. Вызов `misc_register()` регистрирует **единичное** устройство, с одним значением `minor`, определённым в `struct miscdevice`. Поэтому, если драйвер предполагает обслуживать группу однотипных устройств, различающихся по `minor`, то это не есть лучший выбор для использования. Хотя, конечно, драйвер может поочерёдно зарегистрировать **несколько** структур `struct miscdevice`. Вот как вся эта теория выглядит показанном примере:

```
$ sudo insmod misc_dev.ko
$ lsmod | head -n2
Module                  Size  Used by
misc_dev                1167  0
$ cat /proc/modules | grep misc
misc_dev 1167 0 - Live 0xf99e8000
```

```
$ cat /proc/devices  | grep misc
 10 misc
$ ls -l  /dev/own*
crw-rw---- 1 root root 10, 54 Янв 22 22:08 /dev/own_misc_dev
$ cat /dev/own_misc_dev
Hello, world!
```

Операционная система (и прикладные проекты, как например Oracle VirtualBox на листинге ниже) регистрирует с `major` значением 10 достаточно много разносортных устройств:

```
$ ls -l /dev | grep 10,
crw-------  1 root video      10, 175 Янв 22 11:49 agpgart
crw-rw----  1 root root       10,  57 Янв 22 11:50 autofs
crw-rw----  1 root root       10,  61 Янв 22 11:49 cpu_dma_latency
crw-rw-rw-  1 root root       10, 229 Янв 22 11:49 fuse
crw-rw----  1 root root       10, 228 Янв 22 11:49 hpet
crw-rw-rw-+ 1 root kvm        10, 232 Янв 22 11:50 kvm
crw-rw----  1 root root       10, 227 Янв 22 11:49 mcelog
crw-rw----  1 root root       10,  60 Янв 22 11:49 network_latency
crw-rw----  1 root root       10,  59 Янв 22 11:49 network_throughput
crw-r-----  1 root kmem       10, 144 Янв 22 11:49 nvram
crw-rw----  1 root root       10,  54 Янв 22 22:08 own_misc_dev
crw-rw-r--+ 1 root root       10,  58 Янв 22 11:49 rfkill
crw-rw----  1 root root       10, 231 Янв 22 11:49 snapshot
crw-------  1 root root       10,  56 Янв 22 11:50 vboxdrv
crw-rw----  1 root root       10,  55 Янв 22 11:50 vboxnetctl
crw-rw----  1 root root       10,  63 Янв 22 11:49 vga_arbiter
crw-rw----  1 root root       10, 130 Янв 22 11:49 watchdog
```

Все устройства, создаваемые единым модулем с общим `major` (если модуль регистрирует несколько однотипных устройств), регистрируются в `sysfs` в **едином** подклассе (под именем из модуля), в классе `misc`:

```
$ ls /sys/class/misc/own_misc_dev
dev  power  subsystem  uevent
$ tree /sys/devices/virtual/misc/own_misc_dev/
/sys/devices/virtual/misc/own_misc_dev/
├── dev
├── power
│   └── wakeup
├── subsystem -> ../../../../class/misc
└── uevent
```

А вот такое же использование этого модуля, но номер `minor` мы пытаемся задать принудительно:

```
$ sudo insmod misc_dev.ko minor=55
insmod: error inserting 'misc_dev.ko': -1 Device or resource busy
$ sudo insmod misc_dev.ko minor=200
$ ls -l /dev/own*
crw-rw---- 1 root root 10, 200 Янв 22 22:15 /dev/own_misc_dev
$ cat /dev/own*
Hello, world!
```

Управляющие операции устройства

То, что рассматривалось до сих пор, делалось всё на примере единственной операции `read()`. Операция `write()`, как понятно и интуитивно, симметричная, реализуется так же, до сих пор не включалась в обсуждение только для того, чтобы не перегружать текст, и будет показана позже. Но кроме этих операций, которые часто упоминают как операции ввода-вывода **в основном потоке** данных, в Linux достаточно широко используется операция `ioctl()`, применяемая для управления устройством, осуществляющая обмен с устройством вне основного потока данных.

Следующим примером рассмотрим (архив `ioctl.tgz`) реализацию таких управляющих операций. Но для реализации операций **регистрации** такого устройства воспользуемся, так называемым, старым методом регистрации символьного устройства (`register_chrdev()`). Эта техника не потеряла актуальности, и используются на сегодня — это и будет наш третий, последний альтернативный способ **регистрации** модуля драйвера устройства:

ioctl_dev.old.c :

```
#include "ioctl.h"
#include "../dev.h"

// Работа с символьным устройством в старом стиле...
static int dev_open( struct inode *n, struct file *f ) {
   // ... при этом MINOR номер устройства должна обслуживать функция open:
   // unsigned int minor = iminor( n );
   return 0;
}

static int dev_release( struct inode *n, struct file *f ) {
   return 0;
}

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

static const struct file_operations hello_fops = {
   .owner = THIS_MODULE,
   .open = dev_open,
   .release = dev_release,
   .read  = dev_read,
   .ioctl = dev_ioctl
};

#define HELLO_MAJOR 200
#define HELLO_MODNAME "my_ioctl_dev"

static int __init dev_init( void ) {
   int ret = register_chrdev( HELLO_MAJOR, HELLO_MODNAME, &hello_fops );
   if( ret < 0 ) {
      printk( KERN_ERR "=== Can not register char device\n" );
      goto err;
   }
err:
   return ret;
}

static void __exit dev_exit( void ) {
  unregister_chrdev( HELLO_MAJOR, HELLO_MODNAME );
}
```

Так выглядела первоначальная версия этого модуля для версии ядра ниже 2.6.35 ... что будет обсуждено вскоре.

Для согласованного использования типов и констант между модулем, и работающими с ним пользовательскими приложениями введен совместно используемый заголовочный файл `ioctl.h`. Это обычная практика, поскольку операции `ioctl()` никаким образом не стандартизованы, и индивидуальны для каждого проекта:

ioctl.h :

```
typedef struct _RETURN_STRING {
   char buf[ 160 ];
} RETURN_STRING;

#define IOC_MAGIC    'h'
#define IOCTL_GET_STRING _IOR( IOC_MAGIC, 1, RETURN_STRING )
#define DEVPATH "/dev/ioctl"
```

Для единообразного определения кодов для `ioctl`-команд используются (см. файл определений `/usr/include/asm-generic /ioctl.h`) макросы вида (показаны основные, и ещё некоторые):

```
#define _IOR(type,nr,size)      _IOC(_IOC_READ,(type),(nr),(_IOC_TYPECHECK(size)))
#define _IOW(type,nr,size)      _IOC(_IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size)))
#define _IOWR(type,nr,size)     _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size)))
```

Различия в этих макросах — в направлении передачи данных: чтение из устройства, запись в устройство, двухсторонний обмен данных. Отсюда видно, что **каждый** отдельный код операции `ioctl()` предусматривает свой индивидуальный **размер** порции переносимых данных. Попутно заметим, что для операций двухсторонних обменов данными `ioctl()` размер блока данных записываемых в устройство и размер затем считываемых из него данных (в ответ), должны быть **одинаковыми**, даже если объём полезной информации в них отличается в разы. Здесь же зафиксируем не сразу очевидную деталь: операция, которая воспринимается со стороны выполняющей `ioctl()` операцию программы как **запись** в устройство, выглядит как операция **чтения** со стороны модуля ядра, реализующего эту операцию. И симметрично аналогично относительно операции чтения по `ioctl()` из программы.

Для испытаний работы управления модулем необходимо создать тестовое приложение (файл `ioctl.c`), пользующееся вызовами `ioctl()`:

ioctl.c

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include "ioctl.h"

#define ERR(...) fprintf( stderr, "\7" __VA_ARGS__ ), exit( EXIT_FAILURE )

int main( int argc, char *argv[] ) {
   int dfd;                  // дескриптор устройства
   if( ( dfd = open( DEVPATH, O_RDWR ) ) < 0 ) ERR( "Open device error: %m\n" );
   RETURN_STRING buf;
   if( ioctl( dfd, IOCTL_GET_STRING, &buf ) ) ERR( "IOCTL_GET_STRING error: %m\n" );
   fprintf( stdout, (char*)&buf );
   close( dfd );
   return EXIT_SUCCESS;
};
```

Испытываем полученное устройство:

```
$ sudo insmod ioctl_dev.ko
$ cat /proc/devices | grep ioctl
200 my_ioctl_dev
$ sudo mknod -m0666 /dev/ioctl c 200 0
```

```
$ ls -l /dev/ioctl
crw-rw-rw- 1 root root 200, 0 Янв 22 23:27 /dev/ioctl
$ cat /dev/ioctl
Hello, world!
$ ./ioctl
Hello, world!
$ sudo rmmod ioctl_dev
$ ./ioctl
Open device error: No such device or address
$ cat /dev/ioctl
cat: /dev/ioctl: Нет такого устройства или адреса
```

Обратим внимание на одну особенность, которая была названа раньше: регистрируя устройство вызовом `register_chrdev()`, драйвер регистрирует только номер `major`, и получает под свой контроль весь диапазон (0-255) номеров `minor`:

```
$ sudo mknod -m0666 /dev/ioctl c 200 0
$ sudo mknod -m0666 /dev/ioctl200 c 200 200
$ ls -l /dev/ioctl*
crw-rw-rw- 1 root root 200,   0 Янв 22 23:52 /dev/ioctl
crw-rw-rw- 1 root root 200, 200 Янв 22 23:51 /dev/ioctl200
$ cat /dev/ioctl
Hello, world!
$ cat /dev/ioctl200
Hello, world!
```

Различать устройства по `minor` в этом случае должен код модуля, в обработчике операции `open()` по своему **первому** полученному параметру `struct inode*`.

Тот код, который здесь обсуждался, будет работоспособным в ядрах до версии 2.6.34. А в версии 2.6.35 (и везде далее) функция (поле) `ioctl` в таблице `struct file_operations`, была разделена на 2 обработчика: `unlocked_ioctl()` и `compat_ioctl()`. Прежний обработчик был описан с прототипом:

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

Для новых обработчиков прототип **изменён**:

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

Для новых версий ядра наш предыдущий пример должен быть переписан с использованием именно `unlocked_ioctl()`.

Примечание: Другой обработчик (`compat_ioctl()`) предназначен для того, чтобы предоставить возможность 32-битным процессам пространства пользователя иметь возможность осуществлять вызов `ioctl()` к 64-битному ядру. Здесь мы видим весьма искусственное решение, необходимсть которого вызвана тем, что `ioctl()` является весьма опасным вызовом, при котором абсолютно отсутствует контроль типов параметров.

Изменения в обсуждавшемся выше примере для новых ядер затронут только прототип функции обработчика `dev_ioctl()` и инициализацию таблицы `struct file_operations`. В новых версиях ядра (архив `ioctl`)

ioctl_dev.new.c :

```
...

static long dev_ioctl( struct file *f, unsigned int cmd, unsigned long arg ) {
   if( ( _IOC_TYPE( cmd ) != IOC_MAGIC ) ) return -EINVAL;
   ...
}

static const struct file_operations hello_fops = {
   ...
```

```
    .unlocked_ioctl = dev_ioctl
};
```

В таком виде весь остальной код примера останется в неизменном виде.

Примечание: Важно обратить внимание на то, что прототип функции обработчика изменился (даже по **числу** параметров). Если оставить старый прототип обработчика, то пример будет **нормально скомпилирован** (ограничившись предупреждениями, указывающими на строку заполнения таблицы файловых операций, с очень невнятными сообщениями...). Но выполняться такой пример не будет, возвращая ошибку несоответствия кода команды `ioctl()`, поскольку параметры вызова обработчика будут «сдвинуты». Это очень тяжело локализуемая ошибка!

В архиве примеров помещена объединённая версия (файл `ioctl_dev.c`), работоспособная для любых ядер, за счёт использования условной препроцессорной компиляции:

ioctl_dev.c :

```
...
#include <linux/version.h>      /* обязательно! : LINUX_VERSION_CODE */
...
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,35)
static long dev_ioctl( struct file *f, unsigned int cmd, unsigned long arg ) {
#else
static int dev_ioctl( struct inode *n, struct file *f, unsigned int cmd, unsigned long arg ) {
#endif
   if( ( _IOC_TYPE( cmd ) != IOC_MAGIC ) ) return -EINVAL;
   switch( cmd ) {
      case IOCTL_GET_STRING:
         if( copy_to_user( (void*)arg, hello_str, _IOC_SIZE( cmd ) ) ) return -EFAULT;
         break;
      default:
         return -ENOTTY;
   }
   return 0;
}

static const struct file_operations hello_fops = {
   .owner = THIS_MODULE,
   .open = dev_open,
   .release = dev_release,
   .read  = dev_read,
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,35)
   .unlocked_ioctl = dev_ioctl
#else
   .ioctl = dev_ioctl
#endif
};
...
```

Именно подобным образом (условной компиляцией по версии ядра) должен обходить несовместимости версий модуль для практического применения.

Множественное открытие устройства

В рассмотренных выше вариантах мы совершенно дистанцировались от вопроса: как должен работать драйвер устройства, если устройство попытаются использовать (открыть) одновременно несколько пользовательских процессов. Этот вопрос оставляется полностью на усмотрение разработчику драйвера. Здесь может быть несколько вариантов:

1. Драйвер вообще никак **не контролирует** возможности параллельного использования (то, что было

показано во всех рассматриваемых ранее примерах).

2. Драйвер допускает только **единственное** открытие устройства — попытки параллельного открытия будут завершаться ошибкой до тех пор, пока использующий устройство процесс (блокирующий) не завершит свою операцию и не закроет устройство. Это совершенно реалистический сценарий, например, для устройств передачи данных по физическим линиям передачи (Modbus, CAN и др.).

3. Драйвер допускает много **параллельных** сессий использования устройства. При этом драйвер должен реализовать индивидуальный экземпляр данных для каждой копии открытого устройства. Это очень частый случай для **псевдоустройств**, логических устройств, реализующих на программном уровне **модель** устройства. Примерами, являются, например, устройства `/dev/random`, `/dev/zero`, `/dev/null`, или, например, «точки подключения данных» в разнообразных SCADA системах при проектировании АСУТП.

Детальнее это проще рассмотреть на примере (архив `mopen.tgz`). Мы создаём модуль, реализующий все три названных варианта (а то, какой вариант он будет использовать, определяется параметром `mode` запуска модуля, соответственно выше перечисленным : 0, 1, или 2):

`mmopen.c :`

```
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/miscdevice.h>
#include "mopen.h"

MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_VERSION( "6.4" );

static int mode = 0; // открытие: 0 - без контроля, 1 - единичное, 2 - множественное
module_param( mode, int, S_IRUGO );
static int debug = 0;
module_param( debug, int, S_IRUGO );

#define LOG(...) if( debug !=0 ) printk( KERN_INFO "! "__VA_ARGS__ )

static int dev_open = 0;

struct mopen_data {         // область данных драйвера:
   char buf[ LEN_MSG + 1 ]; // буфер данных
   int odd;                 // признак начала чтения
};

static int mopen_open( struct inode *n, struct file *f ) {
   LOG( "open - node: %p, file: %p, refcount: %d", n, f, module_refcount( THIS_MODULE ) );
   if( dev_open ) {
      LOG( "device /dev/%s is busy", DEVNAM );
      return -EBUSY;
   }
   if( 1 == mode ) dev_open++;
   if( 2 == mode ) {
      struct mopen_data *data;
      f->private_data = kmalloc( sizeof( struct mopen_data ), GFP_KERNEL );
      if( NULL == f->private_data ) {
         LOG( "memory allocation error" );
         return -ENOMEM;
      }
      data = (struct mopen_data*)f->private_data;
```

```
      strcpy( data->buf, "dynamic: not initialized!" );  // динамический буфер
      data->odd = 0;
   }
   return 0;
}

static int mopen_release( struct inode *n, struct file *f ) {
   LOG( "close - node: %p, file: %p, refcount: %d", n, f, module_refcount( THIS_MODULE ) );
   if( 1 == mode ) dev_open--;
   if( 2 == mode ) kfree( f->private_data );
   return 0;
}

static struct mopen_data* get_buffer( struct file *f ) {
   static struct mopen_data static_buf = { "static: not initialized!", 0 }; // статический буфер
   return 2 == mode ? (struct mopen_data*)f->private_data : &static_buf;
}

// чтение из /dev/mopen :
static ssize_t mopen_read( struct file *f, char *buf, size_t count, loff_t *pos ) {
   struct mopen_data* data = get_buffer( f );
   LOG( "read - file: %p, read from %p bytes %d; refcount: %d",
        f, data, count, module_refcount( THIS_MODULE ) );
   if( 0 == data->odd ) {
      int res = copy_to_user( (void*)buf, data->buf, strlen( data->buf ) );
      data->odd = 1;
      put_user( '\n', buf + strlen( data->buf ) );
      res = strlen( data->buf ) + 1;
      LOG( "return bytes :  %d", res );
      return res;
   }
   data->odd = 0;
   LOG( "return : EOF" );
   return 0;
}

// запись в /dev/mopen :
static ssize_t mopen_write( struct file *f, const char *buf, size_t count, loff_t *pos ) {
   int res, len = count < LEN_MSG ? count : LEN_MSG;
   struct mopen_data* data = get_buffer( f );
   LOG( "write - file: %p, write to %p bytes %d; refcount: %d",
        f, data, count, module_refcount( THIS_MODULE ) );
   res = copy_from_user( data->buf, (void*)buf, len );
   if( '\n' == data->buf[ len -1 ] ) data->buf[ len -1 ] = '\0';
   else data->buf[ len ] = '\0';
   LOG( "put bytes : %d", len );
   return len;
}

static const struct file_operations mopen_fops = {
   .owner  = THIS_MODULE,
   .open =    mopen_open,
   .release = mopen_release,
   .read   =  mopen_read,
   .write  =  mopen_write,
};

static struct miscdevice mopen_dev = {
   MISC_DYNAMIC_MINOR, DEVNAM, &mopen_fops
```

```
};

static int __init mopen_init( void ) {
   int ret = misc_register( &mopen_dev );
   if( ret ) { LOG( "unable to register %s misc device", DEVNAM ); }
   else { LOG( "installed device /dev/%s in mode %d", DEVNAM, mode ); }
   return ret;
}

static void __exit mopen_exit( void ) {
   LOG( "released device /dev/%s", DEVNAM );
   misc_deregister( &mopen_dev );
}

module_init( mopen_init );
module_exit( mopen_exit );
```

Для тестирования полученного модуля мы будем использовать стандартные команды чтения и записи устройства: `cat` и `echo`, но этого нам будет недостаточно, и мы используем сделанное по этому случаю тестовое приложение, которое выполняет **одновременно** открытие двух дескрипторов нашего устройства, и делает на них поочерёдные операции записи-чтения (но в порядке выполнения операций чтения обратном записи):

pmopen.c :

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mopen.h"

char dev[ 80 ] = "/dev/";

int prepare( char *test ) {
   int df;
   if( ( df = open( dev, O_RDWR ) ) < 0 )
      printf( "open device error: %m\n" );
   int res, len = strlen( test );
   if( ( res = write( df, test, len ) ) != len )
      printf( "write device error: %m\n" );
   else
      printf( "prepared %d bytes: %s\n", res, test );
   return df;
}

void test( int df ) {
   char buf[ LEN_MSG + 1 ];
   int res;
   printf( "-----------------------------------\n" );
   do {
      if( ( res = read( df, buf, LEN_MSG ) ) > 0 ) {
         buf[ res ] = '\0';
         printf( "read %d bytes: %s\n", res, buf );
      }
      else if( res < 0 )
         printf( "read device error: %m\n" );
      else
         printf( "read end of stream\n" );
   } while ( res > 0 );
   printf( "-----------------------------------\n" );
}
```

```
int main( int argc, char *argv[] ) {
   strcat( dev, DEVNAM );
   int df1, df2;                    // разные дескрипторы одного устройства
   df1 = prepare( "1111111" );
   df2 = prepare( "22222" );
   test( df1 );
   test( df2 );
   close( df1 );
   close( df2);
   return EXIT_SUCCESS;
};
```

И модуль и приложение для слаженности своей работы используют небольшой общий заголовочный файл:

mopen.h :

```
#define DEVNAM "mopen"   // имя устройства
#define LEN_MSG 256      // длины буферов устройства
```

Пример, может, и несколько великоват, но он стоит того, чтобы поэкспериментировать с ним в работе для тонкого разграничения деталей возможных реализаций концепции устройства! Итак, первый вариант, когда драйвер никоим образом не контролирует открытия устройства (параметр `mode` здесь можно не указывать — это значение по умолчанию, я делаю это только для наглядности):

```
$ sudo insmod ./mmopen.ko debug=1 mode=0
$ cat /dev/mopen
static: not initialized!
```

Записываем на устройство произвольную символьную строку:

```
$ echo 777777777 > /dev/mopen
$ cat /dev/mopen
777777777
$ ./pmopen
prepared 7 bytes: 1111111
prepared 5 bytes: 22222
-----------------------------------
read 6 bytes: 22222

read end of stream
-----------------------------------
-----------------------------------
read 6 bytes: 22222

read end of stream
-----------------------------------
$ sudo rmmod mmopen
```

Здесь мы наблюдаем нормальную работу драйвера устройства при тестировании его утилитами POSIX (`echo/cat`) — это уже важный элемент контроля корректности, и с этих проверок всегда следует начинать. Но в контексте множественного доступа происходит полная ерунда: две операции записи пишут в один статический буфер устройства, а два последующих чтения, естественно, оба читают идентичные значения, записанные более поздней операцией записи. Очевидно, это совсем не то, что мы хотели бы получить от устройства!

Следующий вариант: устройство допускает только единичные операции доступа, и до тех пор, пока использующий процесс его не освободит, все последующие попытки использования устройства будут безуспешные:

```
$ sudo insmod ./mmopen.ko debug=1 mode=1
$ cat /dev/mopen
static: not initialized!
```

```
$ echo 777777777 > /dev/mopen
$ cat /dev/mopen
777777777
$ ./pmopen
prepared 7 bytes: 1111111
open device error: Device or resource busy
write device error: Bad file descriptor
-----------------------------------
read 8 bytes: 1111111

read end of stream
-----------------------------------
-----------------------------------
read device error: Bad file descriptor
-----------------------------------
$ sudo rmmod mmopen
```

Хорошо видно, как при второй попытке открытия устройства возникла ошибка «устройство занято». В более реалистичном случае, ошибка занятости устройства могла бы или блокировать запрос `open()` до освобождения устройства (в коде модуля), либо приводить к повторению операции с тайм-аутом, как это обычно делается при неблокирующих операциях ввода-вывода.

Следующий вариант: устройство допускающее параллельный доступ, и работающее в каждой копии со своим экземпляром данных. Повторяем для сравнимости всё те же манипуляции:

```
$ sudo insmod ./mmopen.ko debug=1 mode=2
$ cat /dev/mopen
dynamic: not initialized!
$ echo 777777777 > /dev/mopen
$ cat /dev/mopen
dynamic: not initialized!
```

Стоп! ... Очень странный результат. Понять то, что происходит, нам поможет отладочный режим загрузки модуля (для этого и добавлен параметр запуска `debug`, без этого параметра модуль ничего не пишет в системный журнал, чтобы не засорять его) и содержимое системного журнала (показанный вывод в точности соответствует показанной выше последовательности команд):

```
$ sudo insmod ./mmopen.ko mode=2 debug=1
$ echo 9876543210 > /dev/mopen
$ cat /dev/mopen
dynamic: not initialized!
$ dmesg | tail -n10
open - node: f2e855c0, file: f2feaa80, refcount: 1
write - file: f2feaa80, write to f2c5f000 bytes 11; refcount: 1
put bytes : 11
close - node: f2e855c0, file: f2feaa80, refcount: 1
open - node: f2e855c0, file: f2de2d80, refcount: 1
read - file: f2de2d80, read from f2ff9600 bytes 32768; refcount: 1
return bytes :  26
read - file: f2de2d80, read from f2ff9600 bytes 32768; refcount: 1
return : EOF
close - node: f2e855c0, file: f2de2d80, refcount: 1
```

Тестовые операции `echo` и `cat`, каждая, открывают **свой экземпляр** устройства, выполняют требуемую операцию и закрывают устройство. Следующая выполняемая команда работает с **совершенно другим экземпляром** устройства и, соответственно, с другой копией данных! Это косвенно подтверждает и число ссылок на модуль после завершения операций (но об этом мы поговорим детально чуть ниже):

```
$ lsmod | grep mmopen
mmopen                 2459  0
```

Хотя именно то, для чего мы готовили драйвер, множественное открытие дескрипторов усттройства — срабатывает отменно:

```
$ ./pmopen
prepared 7 bytes: 1111111
prepared 5 bytes: 22222
-----------------------------------
read 8 bytes: 1111111

read end of stream
-----------------------------------
-----------------------------------
read 6 bytes: 22222

read end of stream
-----------------------------------
$ sudo rmmod mmopen
$ dmesg | tail -n60
open - node: f2e85950, file: f2f35300, refcount: 1
write - file: f2f35300, write to f2ff9600 bytes 7; refcount: 1
put bytes : 7
open - node: f2e85950, file: f2f35900, refcount: 2
write - file: f2f35900, write to f2ff9200 bytes 5; refcount: 2
put bytes : 5
read - file: f2f35300, read from f2ff9600 bytes 256; refcount: 2
return bytes :  8
read - file: f2f35300, read from f2ff9600 bytes 256; refcount: 2
return : EOF
read - file: f2f35900, read from f2ff9200 bytes 256; refcount: 2
return bytes :  6
read - file: f2f35900, read from f2ff9200 bytes 256; refcount: 2
return : EOF
close - node: f2e85950, file: f2f35300, refcount: 2
close - node: f2e85950, file: f2f35900, refcount: 1
```

Как итог этого рассмотрения, вопрос: всегда ли последний вариант (`mode=2`) лучше других (`mode=0` или `mode=1`)? Этого категорично утверждать нельзя! Очень часто устройство физического доступа (аппаратная реализация) по своей природе требует только монопольного его использования, и тогда схема множественного параллельного доступа становится неуместной. Опять же, схема множественного доступа (в такой или иной реализации) должна предусматривать динамическое управление памятью, что принято считать более опасным в системах критической надёжности и живучести (но и само это мнение тоже может вызывать сомнения). В любом случае, способ открытия устройства может реализоваться по самым различным алгоритмам, должен соответствовать логике решаемой задачи, накладывает требования на реализацию всех прочих операций на устройстве, и, в итоге, заслуживает самой пристальной проработки при начале нового проекта.

Счётчик ссылок использования модуля

О счётчике ссылок использования модуля, и о том, что это один из важнейших элементов контроля безопасности загрузки модулей и целостности ядра системы — сказано неоднократно. Вернёмся ещё раз к вопросу счётчика ссылок использования модуля, и, на примере только что спроектированного модуля, внесём для себя окончательную ясность в вопрос. Для этого изготовим ещё одну элементарную тестовую программу (пользовательский процесс):

simple.c :

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mopen.h"
```

```
int main( int argc, char *argv[] ) {
   char dev[ 80 ] = "/dev/";
   strcat( dev, DEVNAM );
   int df;
   if( ( df = open( dev, O_RDWR ) ) < 0 )
      printf( "open device error: %m" ), exit( EXIT_FAILURE );
   char msg[ 160 ];
   fprintf( stdout, "> " );
   fflush( stdout );
   gets( msg ); // gets() - опасная функция!
   int res, len = strlen( msg );
   if( ( res = write( df, msg, len ) ) != len )
      printf( "write device error: %m" );
   char *p = msg;
   do {
      if( ( res = read( df, p, sizeof( msg ) ) ) > 0 ) {
         *( p + res ) = '\0';
         printf( "read %d bytes: %s", res, p );
         p += res;
      }
      else if( res < 0 )
         printf( "read device error: %m" );
   } while ( res > 0 );
   fprintf( stdout, "%s", msg );
   close( df );
   return EXIT_SUCCESS;
};
```

Смысл теста, на этот раз, в том, что мы можем в отдельных терминалах запустить сколь угодно много копий такого процесса, каждая из который будет ожидать ввода с терминала.

```
$ make
...
/tmp/ccfJzj86.o: In function `main':
simple.c:(.text+0x9c): warning: the `gets' function is dangerous and should not be used.
```

Такое предупреждение при сборке нас не должно смущать: мы и сами наслышаны об опасности функции `gets()` с точки зрения возможного переполнения буфера ввода, но для нашего теста это вполне допустимо, а мы будем соблюдать разумную осторожность при вводе:

```
$ sudo insmod ./mmopen.ko mode=2 debug=1
```

Запустим 3 копии тестового процесса:

```
$ ./simple
> 12345
read 6 bytes: 12345
12345
$ ./simple
> 987
read 4 bytes: 987
987
$ ./simple
> ^C
```

То, что показано, выполняется на 4-х независимых терминалах, и его достаточно сложно объяснять в линейном протоколе, но, будем считать, что мы оставили 3 тестовых процесса заблокированными на ожидании ввода строки (символ приглашения `'>'`). Выполним в этом месте:

```
$ lsmod | grep mmopen
mmopen                  2455  3
```

Примечание: Интересно: `lsmod` показывает число ссылок на модуль, но не знает (не показывает) имён ссылающихся модулей; из консольных команд (запуска модулей) имитировать (и увидеть) такой результат не получится.

```
$ dmesg | tail -n3
open - node: f1899ce0, file: f2e5ff00, refcount: 1
open - node: f1899ce0, file: f2f35880, refcount: 2
open - node: f1899ce0, file: f2de2500, refcount: 3
```

Хорошо видно, как счётчик ссылок использования пробежал диапазон от 0 до 3. После этого введём строки (разной длины) на 2-х копиях тестового процесс, а последний завершим по Ctrl+C (`SIGINT`), чтобы знать, как счётчик использования отреагирует на завершение (аварийное) клиента по сигналу. Вот что мы находим в системном журнале как протокол всех этих манипуляций:

```
$ dmesg | tail -n15
write - file: f2e5ff00, write to f2ff9200 bytes 5; refcount: 3
put bytes : 5
read - file: f2e5ff00, read from f2ff9200 bytes 160; refcount: 3
return bytes :  6
read - file: f2e5ff00, read from f2ff9200 bytes 160; refcount: 3
return : EOF
close - node: f1899ce0, file: f2e5ff00, refcount: 3
write - file: f2f35880, write to f1847800 bytes 3; refcount: 2
put bytes : 3
read - file: f2f35880, read from f1847800 bytes 160; refcount: 2
return bytes :  4
read - file: f2f35880, read from f1847800 bytes 160; refcount: 2
return : EOF
close - node: f1899ce0, file: f2f35880, refcount: 2
close - node: f1899ce0, file: f2de2500, refcount: 1
$ lsmod | grep mmopen
mmopen                  2455  0
```

Примечание: На всём протяжении выполнения функции, реализующей операцию `release()` устройства, счётчик использования ещё не декрементирован: так как сессия файлового открытия ещё не завершена!

Что ещё нужно подчеркнуть, что следует из протокола системного журнала, так это то, что после выполнения `open()` другие операции из той же таблицы файловых операций (`read()`, `write()`) никоим образом не влияют на значение счётчика ссылок.

Всё это говорит о том, что отслеживание ссылок использования при выполнении `open()` и `close()` на сегодня корректно выполняется ядром самостоятельно (что мне не совсем понятно каким путём, когда мы полностью подменяем реализующие операции для `open()` и `close()`, не оставляя места ни для каких умалчиваемых функций). И ещё о том, что неоднократно рекомендуемая необходимость корректировки ссылок из кода при выполнении обработчиков для `open()` и `close()` - на сегодня отпала.

Режимы выполнения операций ввода-вывода

Всё, что рассмотрено ранее относительно операций по `read()` и `write()`, неявно предполагало выполнение этих операций в блокирующем режиме: если данные ещё недоступны, не поступили (при чтении), или не могут быть пока записаны из-за отсутствия буферов (при записи) — операция **блокируется**, а запрашивавший операцию пользовательский процесс переводится в блокированное состояние. Это основной режим операций над файловым дескриптором (в пользовательском пространстве). Любой другой режим (например неблокирующего ввода-вывода) устанавливается явным выполнением управляющего вызова `fcntl()` над файловым дескриптором. Но в любом случае, любые другие возможные режимы и операции (такие, например, как `select()`) с устройством возможны только в том случае, когда для устройства реализована **поддержка** со стороны драйвера,

Примечание: Простейшей иллюстрацией для блокирующего и неблокирующего режима ввода может служить UNIX

операции ввода с терминала: в каноническом режиме терминала (блокирующем) операция `gets()` будет бесконечно ожидать ввода с клавиатуры и его завершения по <Enter>, а в неканоническом режиме операция `getchar()` будет «пробегать» операцию ввода (разве что отмечая для себя признак отсутствия ввода).

Наилучшую (наилучшую из известных автору) классификаций **режимов** и **способов** выполнения операций ввода-вывода дал У. Р. Стивенс [18], он выделяет 5 категорий, которые принципиально различаются:

- блокируемый ввод-вывод;
- неблокируемый ввод-вывод;
- мультиплексирование ввода-вывода (функции `select()` и `poll()`);
- ввод-вывод, управляемый сигналом (сигнал `SIGIO`);
- асинхронный ввод-вывод (функции POSIX.1 `aio_*()`).

Примеры использования их обстоятельнейшим образом описаны в книге того же автора [19], формулировки которого мы будем, без излишних объяснений, опираться в дальнейших примерах.

Неблокирующий ввод-вывод и мультиплексирование

Здесь имеются в виду реализация поддержки (в модуле, драйвере) операций мультиплексированного ожидания возможности выполнения операций ввода-вывода: `select()` и `poll()`. Примеры этого раздела будут много объёмнее и сложнее, чем все предыдущие, в примерах будут использованы механизмы ядра, которые мы ещё не затрагивали, и которые будут рассмотрены далее... Но сложность эта обусловлена тем, что здесь мы начинаем вторгаться в обширную и сложную область: неблокирующие и асинхронные операции ввода-вывода. При первом прочтении этот раздел можно пропустить — на него никак не опирается всё последующее изложение.

Сложность описания подобных механизмов и написания демонстрирующих их примеров состоит в том, чтобы придумать модель-задачу, которая: а). достаточно адекватно использует рассматриваемый механизм и б). была бы до примитивного простой, чтобы её код был не громоздким, легко анализировался и мог использоваться для дальнейшего развития. В данном разделе мы реализуем драйвер (архив `poll.tgz`) устройства (и тестовое окружение к нему), которое функционирует следующим образом:

- устройство допускает неблокирующие операции **записи** (в буфер) — в любом количестве, последовательности и в любое время; операция записи обновляет содержимое буфера устройства и устанавливает указатель чтения в начало нового содержимого;

- операция **чтения** может запрашивать любое число байт в последовательных операциях (от 1 до 32767), последовательные чтения приводят к ситуации `EOF` (буфер вычитан до конца), после чего следующие операции `read()` или `poll()` будут блокироваться до обновления данных операцией `write()`;

- может выполняться операция `read()` в неблокирующем режиме, при исчерпании данных буфера она будет возвращать признак «данные не готовы».

К модулю мы изготовим тесты записи (`pecho` — подобие `echo`) и чтения (`pcat` — подобие `cat`), но позволяющие варьировать режимы ввода-вывода... И, конечно, с этим модулем должны работать и объяснимо себя вести наши неизменные POSIX-тесты `echo` и `cat`. Для согласованного поведения всех составляющих эксперимента, общие их части вынесены в два файла `*.h` :

poll.h :

```
#define DEVNAME "poll"
#define LEN_MSG 160

#ifndef __KERNEL__         // only user space applications
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <poll.h>
#include <errno.h>
#include "user.h"

#else                       // for kernel space module
#include <linux/module.h>
#include <linux/miscdevice.h>
#include <linux/poll.h>
#include <linux/sched.h>
#endif
```

Второй файл (`user.h`) используют только тесты пространства пользователя, мы их посмотрим позже, а пока — сам модуль устройства:

poll.c :

```
#include "poll.h"

MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_VERSION( "5.2" );

static int pause = 100;       // задержка на операции poll, мсек.
module_param( pause, int, S_IRUGO );

static struct private {       // блок данных устройства
   atomic_t roff;             // смещение для чтения
   char buf[ LEN_MSG + 2 ];   // буфер данных
} devblock = {   // статическая инициализация того, что динамически делается в open()
   .roff = ATOMIC_INIT( 0 ),
   .buf = "not initialized yet!\n",
};
static struct private *dev = &devblock;

static DECLARE_WAIT_QUEUE_HEAD( qwait );

static ssize_t read( struct file *file, char *buf, size_t count, loff_t *ppos ) {
   int len = 0;
   int off = atomic_read( &dev->roff );
   if( off > strlen( dev->buf ) ) {         // нет доступных данных
      if( file->f_flags & O_NONBLOCK )
         return -EAGAIN;
      else interruptible_sleep_on( &qwait );
   }
   off = atomic_read( &dev->roff );         // повторное обновление
   if( off == strlen( dev->buf ) ) {
      atomic_set( &dev->roff, off + 1 );
      return 0;                             // EOF
   }
   len = strlen( dev->buf ) - off;          // данные есть (появились?)
   len = count < len ? count : len;
   if( copy_to_user( buf, dev->buf + off, len ) )
      return -EFAULT;
   atomic_set( &dev->roff, off + len );
   return len;
}
```

```
static ssize_t write( struct file *file, const char *buf, size_t count, loff_t *ppos ) {
   int res, len = count < LEN_MSG ? count : LEN_MSG;
   res = copy_from_user( dev->buf, (void*)buf, len );
   dev->buf[ len ] = '\0';                  // восстановить завершение строки
   if( '\n' != dev->buf[ len - 1 ] ) strcat( dev->buf, "\n" );
   atomic_set( &dev->roff, 0 );             // разрешить следующее чтение
   wake_up_interruptible( &qwait );
   return len;
}

unsigned int poll( struct file *file, struct poll_table_struct *poll ) {
   int flag = POLLOUT | POLLWRNORM;
   poll_wait( file, &qwait, poll );
   sleep_on_timeout( &qwait, pause );
   if( atomic_read( &dev->roff ) <= strlen( dev->buf ) )
      flag |= ( POLLIN | POLLRDNORM );
   return flag;
};

static const struct file_operations fops = {
   .owner = THIS_MODULE,
   .read  = read,
   .write = write,
   .poll  = poll,
};

static struct miscdevice pool_dev = {
   MISC_DYNAMIC_MINOR, DEVNAME, &fops
};

static int __init init( void ) {
   int ret = misc_register( &pool_dev );
   if( ret ) printk( KERN_ERR "unable to register device\n" );
   return ret;
}
module_init( init );

static void __exit exit( void ) {
   misc_deregister( &pool_dev );
}
module_exit( exit );
```

По большей части здесь использованы элементы уже рассмотренных ранее примеров, принципиально новые вещи относятся к реализации операции `poll()` и блокирования:

- Операции `poll()` вызывает (всегда) `poll_wait()` для одной (в нашем случае это `qwait`), или нескольких очередей ожидания (часто одна очередь для чтения и одна для записи);
- Далее производится анализ доступности условий для выполнения операций записи и чтения, и на основе этого анализа и возвращается флаг результата (биты тех операций, которые могут быть выполнены вослед без блокирования);
- В операции `read()` может быть указан неблокирующий режим операции: бит `O_NONBLOCK` в поле `f_flags` переданной параметром `struct file` ...
- Если же затребована блокирующая операция чтения, а данные для её выполнения недоступны, вызывающий процесс блокируется;
- Разблокирован читающий процесс будет при выполнении более поздней операции записи (в условиях теста — с другого терминала).

Теперь относительно процессов пространства пользователя. Вот обещанный общий включаемый файл:

user.h :

```
#define ERR(...) fprintf( stderr, "\7" __VA_ARGS__ ), exit( EXIT_FAILURE )

struct parm {
   int blk, vis, mlt;
};
struct parm parms( int argc, char *argv[], int par ) {
   int c;
   struct parm p = { 0, 0, 0 };
   while( ( c = getopt( argc, argv, "bvm" ) ) != EOF )
      switch( c ) {
         case 'b': p.blk = 1; break;
         case 'm': p.mlt = 1; break;
         case 'v': p.vis++; break;
         default: goto err;
   }
   // par > 0 - pecho; par < 0 - pcat
   if( ( par != 0 && ( argc - optind ) != abs( par ) )) goto err;
   if( par < 0 && atoi( argv[ optind ] ) <= 0 ) goto err;
   return p;
err:
   ERR( "usage: %s [-b][-m][-v] %s\n", argv[ 0 ], par < 0 ?
        "<read block size>" : "<write string>" );
}

int opendev( void ) {
   char name[] = "/dev/"DEVNAME;
   int dfd;                              // дескриптор устройства
   if( ( dfd = open( name, O_RDWR ) ) < 0 )
      ERR( "open device error: %m\n" );
   return dfd;
}

void nonblock( int dfd ) {               // операции в режиме O_NONBLOCK
   int cur_flg = fcntl( dfd, F_GETFL );
   if( -1 == fcntl( dfd, F_SETFL, cur_flg | O_NONBLOCK ) )
      ERR( "fcntl device error: %m\n" );
}

const char *interval( struct timeval b, struct timeval a ) {
   static char res[ 40 ];
   long msec = ( a.tv_sec - b.tv_sec ) * 1000 + ( a.tv_usec - b.tv_usec ) / 1000;
   if( ( a.tv_usec - b.tv_usec ) % 1000 >= 500 ) msec++;
   sprintf( res, "%02d:%03d", msec / 1000, msec % 1000 );
   return res;
};
```

Тест записи

pecho.c :

```
#include "poll.h"
int main( int argc, char *argv[] ) {
   struct parm p = parms( argc, argv, 1 );
   const char *sout = argv[ optind ];
   if( p.vis > 0 )
      fprintf( stdout, "nonblocked: %s, multiplexed: %s, string for output: %s\n",
         ( 0 == p.blk ? "yes" : "no" ),
```

```
         ( 0 == p.mlt ? "yes" : "no" ),
         argv[ optind ] );
   int dfd = opendev();                        // дескриптор устройства
   if( 0 == p.blk ) nonblock( dfd );
   struct pollfd client[ 1 ] = {
      { .fd = dfd,
        .events = POLLOUT | POLLWRNORM,
      }
   };
   struct timeval t1, t2;
   gettimeofday( &t1, NULL );
   int res;
   if( 0 == p.mlt ) res = poll( client, 1, -1 );
   res = write( dfd, sout, strlen( sout ) ); // запись
   gettimeofday( &t2, NULL );
   fprintf( stdout, "interval %s write %d bytes: ", interval( t1, t2 ), res );
   if( res < 0 ) ERR( "write error: %m\n" );
   else if( 0 == res ) {
      if( errno == EAGAIN )
         fprintf( stdout, "device NOT READY!\n" );
   }
   else fprintf( stdout, "%s\n", sout );
   close( dfd );
   return EXIT_SUCCESS;
};
```

Формат запуска этой программы (но если вы ошибётесь с опциями и параметрами, то оба из тестов выругаются и подскажут правильный синтаксис):

```
$ ./pecho
usage: ./pecho [-b][-m][-v] <write string>
```

где:

`-b` — установить блокирующий режим операции (по умолчанию неблокирующий);

`-m` — не использовать ожидание на `poll()` (по умолчанию используется);

`-v` — увеличить степень детализации вывода (для отладки);

Параметром задана строка, которая будет записана в устройство `/dev/poll`, если строка содержит пробелы или другие спецсимволы, то она, естественно, должна быть заключена в кавычки. Расширенные (варьируемые опциями) возможности тестовой программы записи, в отличие от следующего теста чтения, не используются и не нужны в полной мере с испытуемым вариантом драйвера. Это сделано чтобы излишне не усложнять изложение. В более реалистичном виде драйвер по записи должен был бы блокироваться при не пустом (не вычитанном) буфере устройства. И вот тогда все возможности теста окажутся востребованными.

Тест чтения (главное действующее лицо всего эксперимента, из-за чего всё делалось):

`pcat.c :`

```
#include "poll.h"
int main( int argc, char *argv[] ) {
   struct parm p = parms( argc, argv, -1 );
   int blk = LEN_MSG;
   if( optind < argc && atoi( argv[ optind ] ) > 0 )
      blk = atoi( argv[ optind ] );
   if( p.vis > 0 )
      fprintf( stdout, "nonblocked: %s, multiplexed: %s, read block size: %s bytes\n",
         ( 0 == p.blk ? "yes" : "no" ),
         ( 0 == p.mlt ? "yes" : "no" ),
         argv[ optind ] );
```

```
    int dfd = opendev();                // дескриптор устройства
    if( 0 == p.blk ) nonblock( dfd );
    struct pollfd client[ 1 ] = {
       { .fd = dfd,
         .events = POLLIN | POLLRDNORM,
       }
    };
    while( 1 ) {
       char buf[ LEN_MSG + 2 ];         // буфер данных
       struct timeval t1, t2;
       int res;
       gettimeofday( &t1, NULL );
       if( 0 == p.mlt ) res = poll( client, 1, -1 );
       res = read( dfd, buf, blk );     // чтение
       gettimeofday( &t2, NULL );
       fprintf( stdout, "interval %s read %d bytes: ", interval( t1, t2 ), res );
       fflush( stdout );
       if( res < 0 ) {
          if( errno == EAGAIN ) {
             fprintf( stdout, "device NOT READY\n" );
             if( p.mlt != 0 ) sleep( 3 );
          }
          else
             ERR( "read error: %m\n" );
       }
       else if( 0 == res ) {
          fprintf( stdout, "read EOF\n" );
          break;
       }
       else {
          buf[ res ] = '\0';
          fprintf( stdout, "%s\n", buf );
       }
    }
    close( dfd );
    return EXIT_SUCCESS;
};
```

Для теста чтения опции гораздо важнее и жёстче контролируются, чем для предыдущего:

```
$ ./pcat -v
usage: ./pcat [-b][-m][-v] <read block size>
```

Отличие здесь в параметре, который должен быть численным, и определяет размер блока (в байтах), который вычитывается за единичную операцию чтения (в цикле).

Примечание: У этого набора тестов множество степеней свободы (набором опций), позволяющих наблюдать самые различные операции: блокирующие и нет, с ожиданием на `poll()` и нет, и др. Ниже показывается только самый характерный набор результатов.

И окончательно наблюдаем как это всё работает...

```
$ sudo insmod poll.ko
$ ls -l /dev/po*
crw-rw---- 1 root root 10, 54 Июн 30 11:57 /dev/poll
crw-r----- 1 root kmem  1,  4 Июн 30 09:52 /dev/port
```

Запись производим сколько угодно раз последовательно:

```
$ echo qwerqr > /dev/poll
$ echo qwerqr > /dev/poll
$ echo qwerqr > /dev/poll
```

А вот чтение можем произвести только один раз:

```
$ cat /dev/poll
qwerqr
```

При повторной операции чтения:

```
$ cat /dev/poll
...
12346456
```

Здесь операция блокируется и ожидает (там, где нарисованы: `...`), до тех пор, пока с другого терминала на произведена операция:

```
$ echo 12346456 > /dev/poll
```

И, как легко можно видеть, заблокированная операция `cat` после разблокирования выводит уже новое, обновлённое значение буфера устройства (а не то, которое было в момент запуска `cat`).

Теперь посмотрим что говорят наши, более детализированные тесты... Вот итог повторного (блокирующегося) чтения, в режиме блокировки на `pool()` и циклическим чтением по 3 байта:

```
$ ./pcat -v 3
nonblocked: yes, multiplexed: yes, read block size: 3 bytes
interval 43:271 read 3 bytes: xxx
interval 00:100 read 3 bytes: xx
interval 00:100 read 3 bytes: yyy
interval 00:100 read 3 bytes: yyy
interval 00:100 read 3 bytes:  zz
interval 00:100 read 3 bytes: zzz
interval 00:100 read 3 bytes:  tt
interval 00:100 read 1 bytes:
interval 00:100 read 0 bytes: read EOF
```

Выполнение команды блокировалось (на этот раз на `pool()`) до выполнения (>43 секунд) в другом терминале:

```
$ ./pecho 'xxxxx yyyyyy zzzzz tt'
interval 00:099 write 21 bytes: xxxxx yyyyyy zzzzz tt
```

Так выглядело выполнение **мультиплексной** функции `poll()` (в реализацию операции `poll()` в драйвере искусственно введена задержка срабатывания 100ms — параметр `pause` установки модуля, чтобы было гарантировано видно, что срабатывает именно она). Для сравнения вот как выглядит то же исполнение, когда вместо `poll()` блокирование происходит на **блокирующем** `read()` (также, как у команды `cat`):

```
$ ./pcat -v -m -b 3
nonblocked: no, multiplexed: no, read block size: 3 bytes
interval 04:812 read 3 bytes: 12.
interval 00:000 read 3 bytes: 34.
interval 00:000 read 3 bytes: 56.
interval 00:000 read 3 bytes: 78.
interval 00:000 read 1 bytes:
interval 00:000 read 0 bytes: read EOF
$ ./pecho -v '12.34.56.78.'
nonblocked: yes, multiplexed: yes, string for output: 12.34.56.78.
interval 00:100 write 12 bytes: 12.34.56.78.
```

А вот как выглядит **неблокирующая** операция чтения не ожидающая на `pool()` (несколько первых строк с интервалом 3 сек. показывают неготовность до обновления данных):

```
$ ./pcat -v 3 -m
nonblocked: yes, multiplexed: no, read block size: 3 bytes
interval 00:000 read -1 bytes: device NOT READY
interval 00:000 read -1 bytes: device NOT READY
interval 00:000 read -1 bytes: device NOT READY
```

```
interval 00:000 read -1 bytes: device NOT READY
interval 00:000 read 3 bytes: 123
interval 00:000 read 3 bytes: 45
interval 00:000 read 3 bytes: 678
interval 00:000 read 3 bytes: 90
interval 00:000 read 0 bytes: read EOF
```

Опять же, делающая доступными данные операция с другого терминала:

```
$ ./pecho '12345 67890'
interval 00:099 write 11 bytes: 12345 67890
```

Здесь нулевые задержки на неблокирующих указывают не время между операциями, которые разрежены по времени, чтобы не засорять листинг и его можно было наблюдать, в **время выполнения самой операции** `read()`, что соответствует действительности.

Задачи

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

2. (*) Напишите драйвер символьного устройства, подобный предыдущему, которое работает по принципу очереди: записываемые данные помещаются в конец очереди, а считываемые берутся из головы, и удаляются из очереди. Для организации очереди используйте структуру struct `list_head`, использую повсеместно в ядре для организации динамических структур. Предусмотрите изменяющиеся, в том числе и весьма большие, объёмы данных, помещённых в очередь (файлы данных). Предполагаем, что драйвер предназначен для записи-чтения любых **бинарных** данных — проверьте работоспособность записью с контрольным считыванием самого файла полученного модуля .ko.

3. Проанализируйте, почему мы в качестве тестовых приложений для драйверов символьных устройств всегда предпочитаем стандартные команды Linux (POSIX) своим собственным тестовым приложениям (привести как можно больше аргументов или примеров).

5. Драйверы: блочные устройства

Прежде всего — некоторые терминологические детали... Блочные устройства в UNIX (и в Linux в частности) — это устройства хранения, устройства с произвольным доступом. Для дальнейшего использования над такими устройствами **надстраиваются** файловые системы.

Все блочные устройства одной природы поддерживаются единым модулем ядра, такие устройства в `/dev` будут представляться именами с одинаковыми префиксами: например, все дисковые устройства на интерфейсах IDE и EIDE будут представлены именами `hda, hdb, hdc, ...` (с единым **префиксом** имени). Так же точно, все дисковые устройства, представленные в одной **логической** модели (независимо от из аппаратной реализации), поддерживаются единым драйвером, например, все диски на интерфейсе SATA, флэш-диски на интерфейсе USB и внешние дисковые накопители на интерфейсе USB — будут представлены в одной модели SCSI интерфейса, и именоваться в `/dev` как `sda, sdb, ...` А близкие флэш накопители, выполненные в различных конструктивах SD-карт, будут поддерживаться другим модулем блочного устройства, и иметь префиксы имени `mmcblk`. Кроме того, над структурой физического блочного устройства (диска) может быть **надстроена** логическая структура, разбиение на разделы (partition), каждый из которых будет рассматриваться системой как отдельное блочное устройство. Каждый раздел получит свой индивидуальный **суффикс**, и уже именно на нём (а не на базовом блочном устройстве) будет создаваться файловая система. Например:

```
$ ls -l /dev/sd*
brw-rw---- 1 root disk 8,  0 апр 26 14:06 /dev/sda
brw-rw---- 1 root disk 8,  1 апр 26 14:06 /dev/sda1
brw-rw---- 1 root disk 8,  2 апр 26 14:06 /dev/sda2
brw-rw---- 1 root disk 8,  3 апр 26 14:06 /dev/sda3
brw-rw---- 1 root disk 8, 16 апр 26 14:51 /dev/sdb
brw-rw---- 1 root disk 8, 17 апр 26 14:51 /dev/sdb1
brw-rw---- 1 root disk 8, 32 апр 26 14:51 /dev/sdc
brw-rw---- 1 root disk 8, 33 апр 26 14:51 /dev/sdc1
brw-rw---- 1 root disk 8, 34 апр 26 14:51 /dev/sdc2
```

Здесь присутствуют 3 достаточно разнородных блочных устройства: `sda` — встроенный HDD накопитель ноутбука на SATA, `sdb` — съёмный флэш-диск на USB, а `sdc` — внешний HDD накопитель Transcend на USB, на каждом из **физических** устройств созданы **логические** разделы. А на следующем примере показано представление SD-карты:

```
$ ls -l /dev/mmc*
brw-rw---- 1 root disk 179, 0 апр 26 14:06 /dev/mmcblk0
brw-rw---- 1 root disk 179, 1 апр 26 14:06 /dev/mmcblk0p1
brw-rw---- 1 root disk 179, 2 апр 26 14:06 /dev/mmcblk0p2
```

Здесь интересно то, что:

- префикс имени устройства (определяемый модулем ядра) вовсе не обязательно 2-х символьный (как чаще всего бывает), для SD-карты это `mmcblk`;

- суффиксы **порядка** представления устройств не всегда литерно-алфавитные (`hda, hdb, ...`), для SD-карт это число;

- суффиксы разделов (partition) в разбиении устройства не всегда числовые (`hda1, hda2, ...`), для SD-карт это `p1, p2, ...`;

Из показанного можно сделать догадку, что все префиксы и суффиксы именования блочных устройств определяются кодом поддерживающего модуля, и очень скоро мы убедимся, что это именно так.

Блочное устройство (как и явствует из самого его названия) обеспечивает обмен блоками данных. **Блок** блок единицу данных фиксированного размера, которыми осуществляется обмен в системе. Размер блока данных определяется ядром (версией, аппаратной платформой), чаще всего размер блока совпадает с размером страницы аппаратной архитектуры, и для 32-битной архитектуры x86 составляет 4096 байт. Оборудование же хранит данные на физическом носителе разбитые в таких единицах как **сектор**. Исторически сложилось так, что несколько десятилетий аппаратное обеспечение создавалось для работы с размером сектора 512 байт. В последние годы существует тенденция в новых устройствах оперировать с большими секторами (4096 байт).

Вообще то, любое блочное устройство в UNIX, и в Linux в частности, представляется как последовательность байт (raw, сырое представление), например, `/dev/sda`. Именно такое представление и должен обеспечить модуль-драйвер блочного устройства. Именно поэтому мы можем легко, как с символьного устройства, считать загрузочный сектор диска в файл:

```
$ sudo dd if=/dev/sda of=MBR bs=512 count=1
1+0 записей считано
1+0 записей написано
 скопировано 512 байт (512 B), 0,0295399 с, 17,3 kB/с
$ hexdump -C MBR
00000000  eb 63 90 33 2e 30 fa fc  be 00 7c bf 00 06 8c c8  |.c.3.0....|.....|
00000010  8e d0 89 f4 8e c0 8e d8  51 b9 00 01 f3 a5 59 e9  |........Q.....Y.|
00000020  00 8a fb b4 02 cd 16 24  03 3c 03 75 05 c6 06 61  |.......$.<.u...a|
00000030  07 01 bb be 07 b9 04 00  80 3f 80 74 0e 83 03 02  |.........?.t....|
00000040  ff 00 00 20 01 00 00 00  00 02 fa 90 90 f6 c2 80  |... ............|
00000050  75 02 b2 80 ea 59 7c 00  00 31 00 80 01 00 00 00  |u....Y|..1......|
00000060  00 00 00 00 ff fa 90 90  f6 c2 80 74 05 f6 c2 70  |...........t...p|
00000070  74 02 b2 80 ea 79 7c 00  00 31 c0 8e d8 8e d0 bc  |t....y|..1......|
00000080  00 20 fb a0 64 7c 3c ff  74 02 88 c2 52 be 80 7d  |. ..d|<.t...R..}|
00000090  e8 17 01 be 05 7c b4 41  bb aa 55 cd 13 5a 52 72  |.....|.A..U..ZRr|
000000a0  3d 81 fb 55 aa 75 37 83  e1 01 74 32 31 c0 89 44  |=..U.u7...t21..D|
000000b0  04 40 88 44 ff 89 44 02  c7 04 10 00 66 8b 1e 5c  |.@.D..D.....f..\|
000000c0  7c 66 89 5c 08 66 8b 1e  60 7c 66 89 5c 0c c7 44  ||f.\.f..`|f.\..D|
000000d0  06 00 70 b4 42 cd 13 72  05 bb 00 70 eb 76 b4 08  |..p.B..r...p.v..|
000000e0  cd 13 73 0d f6 c2 80 0f  84 d8 00 be 8b 7d e9 82  |..s..........}..|
000000f0  00 66 0f b6 c6 88 64 ff  40 66 89 44 04 0f b6 d1  |.f....d.@f.D....|
00000100  c1 e2 02 88 e8 88 f4 40  89 44 08 0f b6 c2 c0 e8  |.......@.D......|
00000110  02 66 89 04 66 a1 60 7c  66 09 c0 75 4e 66 a1 5c  |.f..f.`|f..uNf.\|
00000120  7c 66 31 d2 66 f7 34 88  d1 31 d2 66 f7 74 04 3b  ||f1.f.4..1.f.t.;|
00000130  44 08 7d 37 fe c1 88 c5  30 c0 c1 e8 02 08 c1 88  |D.}7....0.......|
00000140  d0 5a 88 c6 bb 00 70 8e  c3 31 db b8 01 02 cd 13  |.Z....p..1......|
00000150  72 1e 8c c3 60 1e b9 00  01 8e db 31 f6 bf 00 80  |r...`......1....|
00000160  8e c6 fc f3 a5 1f 61 ff  26 5a 7c be 86 7d eb 03  |......a.&Z|..}..|
00000170  be 95 7d e8 34 00 be 9a  7d e8 2e 00 cd 18 eb fe  |..}.4...}.......|
00000180  47 52 55 42 20 00 47 65  6f 6d 00 48 61 72 64 20  |GRUB .Geom.Hard |
00000190  44 69 73 6b 00 52 65 61  64 00 20 45 72 72 6f 72  |Disk.Read. Error|
000001a0  0d 0a 00 bb 01 00 b4 0e  cd 10 ac 3c 00 75 f4 c3  |...........<.u..|
000001b0  00 00 00 00 00 00 00 00  61 d9 61 d9 00 00 00 20  |........a.a.... |
000001c0  21 00 83 fe ff ff 00 08  00 00 00 00 5f 05 00 fe  |!..........._...|
000001d0  ff ff 82 fe ff ff 00 08  5f 05 00 c0 5d 00 80 fe  |........_...]...|
000001e0  ff ff 83 fe ff ff 00 c8  bc 05 00 00 40 01 00 00  |............@...|
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
00000200
```

Или можем полностью переносить (структуру и содержимое) одного физического носителя на другой простым копированием блочного устройства как байтового потока:

```
$ sudo cp /dev/sda /dev/sdf
...
```

Позже на последовательность байт, созданную драйвером блочного устройства, может быть **наложена** структура разделов (partition) в формате MBR (Master Boot Record — программой `fdisk`) или в новом, идущем на смену, формате GPT (GUID Pertition Table — программами `parted, gparted, gdisk`). Далее, на сам диск, или любой его раздел может быть **наложена** структура любой из **многих** файловых систем, поддерживаемых Linux, что делается программами:

```
$ ls -w80 /sbin/mkfs*
/sbin/mkfs         /sbin/mkfs.ext3     /sbin/mkfs.gfs2     /sbin/mkfs.ntfs
/sbin/mkfs.btrfs   /sbin/mkfs.ext4     /sbin/mkfs.hfsplus  /sbin/mkfs.reiserfs
/sbin/mkfs.cramfs  /sbin/mkfs.ext4dev  /sbin/mkfs.minix    /sbin/mkfs.vfat
/sbin/mkfs.ext2    /sbin/mkfs.fat      /sbin/mkfs.msdos    /sbin/mkfs.xfs
```

Здесь обратим внимание на то, что все многочисленные перечисленные программы (`*disk, mkfs*` и др.) являются процессами **пользовательского** адресного пространства, и никакого отношения к ядру (и модулям яда) уже не имеют. Задача модуля блочного устройства состоит только в создании сырого устройства `/dev/xxx`, позволяющего выполнять на нём блочные операции. В этом состоит ещё одно отличие блочных устройств: после загрузки модуля поддержки устройства, нужно ещё выполнить достаточно много манипуляций пользовательскими программами, чтобы выполнить подготовку файловой структуры устройства, и сделать его пригодным для использования.

Так как работа блочного устройства в значительной мере завязана на его последующее структурирование уже в пользовательском пространстве, а отдельные характеристики для возможностей такого структурирования вообще ограничиваются кодом модуля, то рассмотрение техники написания модулей блочных устройств будет перемежаться со связанными вопросами структурирования и использования таких блочных устройств.

В принципе, модуль блочного устройства мог бы во многом наследовать технику символьных устройств, детально рассматриваемых на примерах ранее, и для некоторых операций это так и происходит. Но сами операции, осуществляющие поддержку обмена данными (ввода-вывода) строятся по-другому. Начнём с рассмотрения именно аналогий между символьными и блочными устройствами, и только затем перейдём к специфике.

Особенности драйвера блочного устройства

Из различий использования и происходят отличия модулей блочных устройств от рассматриваемых выше символьных. Основных факторов, порождающими эти отличия, есть следующие:

1. Для блочных устройств первостепенным вопросом является вопрос производительности (скорости). Для символьных устройств производительность может быть не основным фактором, многие символьные устройства могут работать ниже своей максимальной скорости и производительность системы в целом от этого не страдает. Но для блочных вопрос производительности — это вопрос вообще конкурентных преимуществ операционной системы на рынке. Поэтому большая часть разработки модуля блочного уровня будет сосредоточена на обеспечении производительности.

2. Запросы на ввод вывод символьные устройства получают из процессов пространства пользователя, явно выполняющих операции `read()` или `write()`. Блочные устройства могут получать запросы ввода-вывода как из процессов пространства пользователя, так и из кода ядра (модулей ядра), например, при монтировании дисковых устройств, или виртуализации страниц оперативной памяти на диск. Исходя из этого, запросы ввода-вывода должны прежде попадать в блочную подсистему ввода-вывода ядра, и только затем передаваться нею непосредственно драйверу, осуществляющему обмен с устройством.

3. Блочное устройство, как будет рассмотрено вскоре, не выполняет непосредственно запросы `read()` и `write()`, как это делает символьное устройство. Напротив, запросы на обработку драйверу поступают через очередь запросов на обслуживание, которую поддерживает ядро. И ядро формирует вызовом метода `request()` характер, последовательность и объём операций, выполняемых драйвером. Это (обработка очереди ядра) основной, наиболее частый механизм работы блочного драйвера, но могут быть и более редкие в использовании варианты, которые тоже будут рассмотрены далее.

4. Блочное устройство, представленное модулем как имя в `/dev`, ещё далеко не пригодно непосредственно для дальнейшего использования. Над ним ещё необходимо выполнить целый ряд подготовительных манипуляций пространства пользователя, прежде чем устройство можно использовать (создание структуры разделов, **форматирование** файловых систем на разделах).

Обзор примеров реализации

Техника блочных устройств гораздо более громоздкая, чем устройств символьных. Архив кода примеров для блочных устройств представляет несколько совершенно различных реализаций, а поэтому требует кратких комментариев:

```
$ ls -l | grep ^d
drwxrwxr-x 2 Olej Olej  4096 апр 24 13:00 block_mod.ELDD_14
drwxr-xr-x 2 Olej Olej  4096 апр 24 13:00 block_mod.LDD3
drwxr-xr-x 2 Olej Olej  4096 апр 24 13:00 block_mod.LDD_35
drwxr-xr-x 2 Olej Olej  4096 апр 24 13:00 dubfl
```

Первые 3 реализации в перечислении — это реализации RAM-дисков (структура блочного устройства в памяти). Такая организация наиболее гибкая для отработки и экспериментов, а потом может быть перенесена на реальный физический носитель. Эти 3 реализации отталкивались в начале развития от примеров, более или менее полно писанных в 3-х книгах (все детально названы в библиографии), откуда за ними и остались такие имена тестовых каталогов:

- ELDD_14 — Sreekrishnan Venkateswaran, «Essential Linux Device Drivers» (гл.14), имя этого модуля `block_mod_e.ko`;

- LDD3 — Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, «Linux Device Drivers», имя этого модуля `block_mod_s.ko`;

- LDD_35 — Jerry Cooperstein, «Writing Linux Device Drivers» в 2-х томах (гл.35), имя этого модуля `block_mod_c.ko`

Именно под такими именами эти модули и будут показываться в примерах их использования без дальнейших разъяснений. Все 3 реализации, в исходном своём виде (такой вид вложен в подкаталоги к виде `.tgz` архивов), даже не компилируются в ядре уже даже начиная с 2.6.32 (из-за большой волатильности API новых ядер), поэтому они радикально трансформированы, а потом дальше развивались в нужном мне направлении.

Последний вариант (`dubfl`) использует в качестве физического носителя предварительно создаваемый файл (командой `dd` или любым другим образом) в файловой системе Linux (конечно, достаточно большго размера). Информация на таком «устройстве» хранится перманентно (между выключениями питания компьютера), поэтому такой вариант гораздо удобней для определённых групп экспериментов.

Регистрация устройства

Регистрация блочного устройства во многом напоминает регистрацию символьного устройства, с коррекцией на присущую им специфику.

Подготовка к регистрации

Начинается регистрация (и, соответственно, завершается регистрация по окончанию работы) устройства с вызовов API:

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

Точно так же, как это было для символьных устройств, в качестве `major` в вызов `register_blkdev()` может быть: нулевое значение, что позволит системе присвоить `major` первое свободное значение.

В любом случае, вызов `register_blkdev()` возвратит текущее значение `major`, или сигнализирующее

об ошибке отрицательное значение (как это всегда принято в ядре), обычно это бывает когда в вызове для `major` задаётся принудительное значение, уже занятое ранее другим устройством в системе.

В качестве `name` в этот вызов передаётся родовое имя устройств, например, для SCSI устройств это было бы `"sd"`, а для блочных устройств, создаваемых в примерах далее (`xda, xdb, ...`) — это будет `"xd"`. Регистрация имени устройства создаёт соответствующую запись в файле `/proc/devices`, но не создаёт ещё самого устройства в `/dev`:

```
$ cat /proc/devices | grep xd
252 xd
```

В ядре 2.6 и старше, как пишут, в принципе, регистрацию с помощью `register_blkdev()` можно и не проводить, но обычно это делается, и это больше дань традиции.

Здесь же (сразу перед `register_blkdev()`, или после), **для каждого** устройства (привода) драйвера, проделывают инициализацию **очереди обслуживания**, связанной с устройством (об этом мы мы будем говорить вскоре, когда перейдём к рассмотрению операций чтения и записи). Делается это кодом подобным следующему:

```
spinlock_t xda_lock;
struct request_queue* xda_request_queue;
...
spin_lock_init( &xda_lock );
if( !( xda_request_queue = blk_init_queue( &xda_request_func, &xda_lock ) ) ) {
   // ошибка и завершение
}
```

Показанный фрагмент:

а). создаёт очередь обслуживания запросов `xda_request_queue` в ядре,

б). увязывает её с примитивом синхронизации `xda_lock`, который будет использоваться для монополизации выполняемых операций с этой очередью;

в). определяет для очереди `xda_request_queue` функцию обработки запросов, которая будет вызываться **ядром** при каждом требовании на обработку запроса чтения или записи, функция должна иметь прототип вида:

```
static void xda_request_func( struct request_queue *q ) { ... }
```

Забегая вперёд, отметим, что обработка запросов обслуживания с помощью очереди ядра является не единственным способом обеспечения операций чтения и записи. Но такой способ используют 95% драйверов. Остающиеся 5% реализуют прямое выполнение запросов по мере их поступления, и такой способ также будет рассмотрен далее.

Но все эти подготовительные операции пока не приблизили нас к созданию отображения блочного устройства в каталог `/dev`. Для реального создания отображения **каждого диска** в `/dev` нужно а). создать для этого устройства структуру `struct gendisk`, б). заполнить эту структуру, в). зарегистрировать структуру в ядре. Структура `struct gendisk`, (описана в `<linux/genhd.h>`) является описанием каждого диска в ядре.

Примечание: Экземпляром точно такой же структуры в ядре будет описываться **каждый раздел** (partition) диска, создаваемый с помощью `fdisk` или `parted`. Но разработчику не нужно беспокоится об этих экземплярах — их позже нормально создаст ядро при работе всё тех же утилит, производящих разбивку диска.

Структуру `struct gendisk`, нельзя создать просто так, как мы создаём другие структуры (размещая память `kmalloc()` или подобными вызовами), она представляет собой динамически создаваемую структуру, которая сильно завязана в ядре, и требует специальных манипуляций со стороны ядра для инициализации. Поэтому создаём её (а в конце работы уничтожаем) вызовами:

```
struct gendisk* alloc_disk( int minors );
void del_gendisk( struct gendisk* );
```

Диски с разметкой MBR и GPT

Параметр `minors` в вызове `alloc_disk()` должен быть числом младших номеров, **резервируемых** для

отображения в /dev этого диска и **всех его разделов** в будущем.

Примечание: Не следует рассчитывать, что позже можно будет просто изменить соответствующее поле minors в структуре struct gendisk, которое и заполняется вызовом alloc_disk(), и ожидать, что всё будет нормально работать: там инициализация существенно сложнее, похоже, что под размер выделяется minors соответствующее число слотов (дочерних структур struct gendisk).

Параметр minors в вызове alloc_disk() заслуживает отдельного рассмотрения... Если вы зададите для minors значение 4, то вы получите возможность с помощью fdisk создавать до 4-х **первичных** (primary) разделов в MBR этого диска (/dev/xda1 ... /dev/xda4). Но вы не сможете в этом случае создать **ни единого расширенного** (extended) раздела, потому, что первый же созданный логический (logical) раздел внутри расширенного получит номер 5:

```
# fdisk -l /dev/xda
...
Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/xda1               1        4000        2000   83  Linux
/dev/xda2            4001        8191        2095+   5  Расширенный
/dev/xda5            4002        8191        2095   83  Linux
# ls -l /dev/xda*
brw-rw---- 1 root disk 252, 0 нояб. 12 10:44 /dev/xda
brw-rw---- 1 root disk 252, 1 нояб. 12 10:44 /dev/xda1
brw-rw---- 1 root disk 252, 2 нояб. 12 10:44 /dev/xda2
brw-rw---- 1 root disk 252, 5 нояб. 12 10:46 /dev/xda5
```

Обычно в качестве minors указывают значение 16. Некоторые драйверы указывают это значение как 64.

Примечание: В обсуждениях пользователи часто изумляются почему никакими сторонними средствами разбивки диска не удаётся получить больше 16 разделов на диске (хоть сам fdisk и не умеет этого делать, но по определению цепочка вложенных extended разделов может быть безграничной, и сторонние средства разбивки позволяют создавать такие вложенные структуры расширенных разделов). Как мы здесь видим, невозможно не **создать** большее число разделов на физическом диске, а невозможно **отобразить** эти созданные разделы драйвером, поддерживающим этот диск.

Всё сказанное выше относится к способу разбивки диска в стандарте разметки MBR (Master Boot Record), который привычен и использовался, как минимум, на протяжении последних 35 лет. Но в настоящее время происходит массированный переход к разметке диска в стандарте GPT (GUID Partition Table), который давно уже назрел. Основные отличительные стороны GPT в контексте нашего рассмотрения модулей блочных устройств (GPT в целом — слишком обширная тема) следующие:

- используются только первичные разделы;
- число таких разделов может быть до 128;
- полностью изменены идентификаторы типов разделов и они теперь 32-бит (например, для Linux файловых систем — 8300 вместо прежних 83);

Утилита fdisk используется для работы с форматом MBR, она не умеет понимать GPT формат диска, и будет сообщать :

```
$ sudo fdisk -l /dev/sdc
WARNING: GPT (GUID Partition Table) detected on '/dev/sdc'! The util fdisk doesn't support GPT.
Use GNU Parted.
...
```

Для работы с дисками GPT используем утилиты parted (gparted), или ставшую уже популярной утилиту gdisk. Вот как видит parted один из GPT дисков:

```
$ sudo parted /dev/sdb
GNU Parted 3.0
Используется /dev/sdb
(parted) print
Модель: Ut163 USB2FlashStorage (scsi)
Диск /dev/sdb: 1011MB
```

```
Размер сектора (логич./физич.): 512B/512B
Таблица разделов: gpt
Disk Flags:.

Номер Начало  Конец   Размер  Файловая система Имя                  Флаги
1     1049kB  53,5MB  52,4MB  ext2             EFI System           загрузочный, legacy_boot
10    53,5MB   578MB   524MB                    Microsoft basic data
20    578MB   1011MB   433MB                    Linux filesystem
```

Утилитами для работы с дисками GPT можно создать до 128 разделов (или любое число дисков с номерами разделов из диапазона 1...128). Вот как утилитой `gdisk` созданы 18 разделов на диске и нею отображаются:

```
$ sudo gdisk -l /dev/sdf
GPT fdisk (gdisk) version 0.8.4

Partition table scan:
  MBR: protective
  BSD: not present
  APM: not present
  GPT: present

Found valid GPT with protective MBR; using GPT.
Disk /dev/sdf: 7827456 sectors, 3.7 GiB
Logical sector size: 512 bytes
Disk identifier (GUID): C9533CB0-A119-429D-84D8-2B5C1DEA7E30
Partition table holds up to 128 entries
First usable sector is 34, last usable sector is 7827422
Partitions will be aligned on 2048-sector boundaries
Total free space is 5984189 sectors (2.9 GiB)

Number  Start (sector)    End (sector)    Size   Code  Name
 21           2048          104447    50.0 MiB   8300  Linux filesystem
 22         104448          206847    50.0 MiB   8300  Linux filesystem
 23         206848          309247    50.0 MiB   8300  Linux filesystem
 24         309248          411647    50.0 MiB   8300  Linux filesystem
 25         411648          514047    50.0 MiB   8300  Linux filesystem
 26         514048          616447    50.0 MiB   8300  Linux filesystem
 27         616448          718847    50.0 MiB   8300  Linux filesystem
 28         718848          821247    50.0 MiB   8300  Linux filesystem
 29         821248          923647    50.0 MiB   8300  Linux filesystem
101        1128448         1230847    50.0 MiB   8300  Linux filesystem
102        1230848         1333247    50.0 MiB   8300  Linux filesystem
103        1333248         1435647    50.0 MiB   8300  Linux filesystem
104        1435648         1538047    50.0 MiB   8300  Linux filesystem
105        1538048         1640447    50.0 MiB   8300  Linux filesystem
106        1640448         1742847    50.0 MiB   8300  Linux filesystem
107        1742848         1845247    50.0 MiB   8300  Linux filesystem
108        1845248         1947647    50.0 MiB   8300  Linux filesystem
109        1947648         2050047    50.0 MiB   8300  Linux filesystem
```

И вот как такая структура GPT диска отображается в Linux хорошо написанным (Fedora 17, ядро 3.5) модулем поддержки:

```
$ ls /dev/sdf*
/dev/sdf     /dev/sdf103  /dev/sdf106  /dev/sdf109  /dev/sdf23  /dev/sdf26  /dev/sdf29
/dev/sdf101  /dev/sdf104  /dev/sdf107  /dev/sdf21   /dev/sdf24  /dev/sdf27
/dev/sdf102  /dev/sdf105  /dev/sdf108  /dev/sdf22   /dev/sdf25  /dev/sdf28
```

Из показанного очевидно, что грядут большие перемены. И это должно быть учтено при создании модулей ядра блочных устройств. В частности, параметр `minors` в вызове `alloc_disk()`, наверное, должен указываться как 128.

Заполнение структуры

Мы остановились на **создании** структуры `struct gendisk`, после чего самое время перейти к **заполнению** её полей. Это весьма большая структура (описана в `<linux/genhd.h>`), отметим только те поля, которые нужно заполнить под свой драйвер (есть ещё несколько полей, которые требуют «технического» заполнения, их можно видеть в примере далее):

```
struct gendisk {
```

```
        int major;                      /* major number of driver */
        int first_minor;
        int minors;                     /* maximum number of minors, =1 for
                                         * disks that can't be partitioned. */
       char disk_name[DISK_NAME_LEN];   /* name of major driver */
...
       const struct block_device_operations *fops;
...
}
```

Здесь для нас важны:

- `major` — уже встречавшийся нам старший номер для устройств такого **класса** (поддерживаемых этим модулем ядра), мы его ранее присвоили устройству принудительно, или получили в результате вызова `register_blkdev( 0, ... );`

- `minors` — максимальное число **разделов**, обслуживаемое на диске, это значение обсуждалось выше — это поле заполнено вызовом `alloc_disk()`, оно, главным образом, для чтения;

- `first_minor` — номер `minor`, представляющий сам диск в `/dev` (например `/dev/xda`), последующие разделы диска (в пределах их максимального числа `minors`) получат соответствующие младшие номера, например, для `/dev/xda5` будет использован номер `first_minor + 5`:

```
# ls -l /dev/xda*
brw-rw---- 1 root disk 252, 0 нояб. 12 10:44 /dev/xda
brw-rw---- 1 root disk 252, 1 нояб. 12 10:44 /dev/xda1
brw-rw---- 1 root disk 252, 2 нояб. 12 10:44 /dev/xda2
brw-rw---- 1 root disk 252, 5 нояб. 12 10:46 /dev/xda5
```

- `disk_name` — имя диска, с которым он будет отображаться в `/dev`, родовое имя устройств `"xd"` (шаблон имени, **префикс** — драйвер может обслуживать более одного привода устройства) мы уже указывали в вызове типа:

```
register_blkdev( major, MY_DEVICE_NAME );
```

Теперь здесь мы можем персонифицировать имя для конкретного привода (если создаётся несколько устройств `"xd"`, как в примере), нечто по типу:

```
snprintf( disk_name, DISK_NAME_LEN - 1, "xd%c",  'a' + i );
```

Это те имена (`xda, xdb, xdc`, ...), которые появятся в:

```
$ cat /proc/partitions
major minor  #blocks  name
   8        0   58615704 sda
   8        1   45056000 sda1
   8        2    3072000 sda2
   8        3   10485760 sda3
  11        0    1048575 sr0
 179        0    7774208 mmcblk0
 179        1     522081 mmcblk0p1
 179        2    1322111 mmcblk0p2
 252        0       4096 xda
 252        1       2000 xda1
 252        2          1 xda2
 252        5       2095 xda5
 252       16       4096 xdb
 252       32       4096 xdc
 252       48       4096 xdd
```

- `fops` — адрес таблицы операций устройства, во многом аналогичной такой же для символьного

устройства, которую мы детально обсудим вскоре;

Кроме непосредственно заполнения полей структуры `struct gendisk` примерно в этом месте делается занесение (в ту же структуру) полной ёмкости устройства, выраженной в **секторах** по 512 байт:

```
inline void set_capacity( struct gendisk*, sector_t size );
```

Завершение регистрации

Теперь структура `struct gendisk` выделена и заполнена, но это ещё не делает блочное устройство доступным для использования системе. Чтобы завершить все наши подготовительные операции, мы должны вызвать:

```
void add_disk( struct gendisk* gd );
```

Здесь `gd` и есть структура, которую мы так тщательно готовили накануне.

Вызов `add_disk()` достаточно **опасный**: как только происходит вызов, диск становится активным и его методы могут быть вызваны, независимо от нашего участия, асинхронно, в любое время. На самом же деле, первые такие вызовы происходят немедленно, даже ещё до того, как произойдёт возврат из самого вызова `add_disk()` (это можно наблюдать в системном журнале командой `dmesg`). Этим ранним вызовам мы обязаны попыткам ядра вычитать начальные сектора диска в поисках таблицы разделов (MBR или GPT, см. ранее). Если к моменту вызова `add_disk()` драйвер ещё не полностью или некорректно инициализирован, то, с большой вероятностью, вызов приведёт к ошибке драйвера и, через некоторое, обычно короткое, время — к полному краху системы.

На этом регистрация устройства завершена, устройство создано в системе, и мы можем перейти к детальному рассмотрению теперь уже работы такого устройства.

Таблица операций устройства

Выше мы заполнили в структуре описания диска `struct gendisk` поле адреса `fops` — таблицу операций блочного устройства. Таблица функций `struct block_device_operations` (ищите её в `<linux/blkdev.h>`) по смыслу выполняет для блочного устройства ту же роль, что и таблица файловых операций символьного устройства `struct file_operations`, которая детально изучалась ранее (вид таблицы показан из ядра 3.5):

```
struct block_device_operations {
        int (*open) (struct block_device *, fmode_t);
        int (*release) (struct gendisk *, fmode_t);
        int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*direct_access) (struct block_device *, sector_t,
                              void **, unsigned long *);
        unsigned int (*check_events) (struct gendisk *disk,
                                      unsigned int clearing);
        /* ->media_changed() is DEPRECATED, use ->check_events() instead */
        int (*media_changed) (struct gendisk *);
        void (*unlock_native_capacity) (struct gendisk *);
        int (*revalidate_disk) (struct gendisk *);
        int (*getgeo)(struct block_device *, struct hd_geometry *);
        /* this callback is with swap_lock and sometimes page table lock held */
        void (*swap_slot_free_notify) (struct block_device *, unsigned long);
        struct module *owner;
};
```

Примечание: Ещё до версии ядра 3.10 (3.9 и ниже, все 2.6.X) прототип метода `release` должен был возвращать значение

int, но после этого (3.10 и старше) он имеет возвращаемое значение void (см. <linux/blkdev.h>) Поэтому мы должны здесь использовать условную компиляцию, что уже неоднократно показывалось.

Здесь всё относительно интуитивно понятно, и во многом напоминает то, что мы видим для символьных устройств. Операции open() и release() — это функции, вызываемые при открытии и закрытии устройства, весьма часто они вызываются **неявно**, и их не приходится описывать. Неявный их вызов означает, что они вызываются **ядром** в нескольких случаях, например: при загрузке драйвера после вызова add_disk() ядро пытается прочитать в своих интересах таблицу разделов (partititon) диска (MBR или GPT). Вызовы open() и release() **всегда** следуют за **монтированием** и **размонтированием**, соответственно, блочного устройства командами пользователя. Это можно наблюдать из кода и результатов испытаний модуля dubfl.ko (каталог примера dubfl):

```
# insmod dubfl.ko file=XXX
$ ls -l /dev/dbf
brw-rw---- 1 root disk 252, 1 май 20 18:57 /dev/dbf
# mount -t vfat /dev/dbf /mnt/xdb
# umount /dev/dbf
$ dmesg | tail -n2
[20961.008908] + open device /dev/dbf
[21129.476313] + close device /dev/dbf
```

В качестве образцов того, как используются некоторые простейшие функции из таблицы файловых операций, можно посмотреть то, как реализуются операции getgeo() (возвратить геометрию блочного устройства) и ioctl() из таблицы fops в архиве примеров:

common.c :

```
...
static struct block_device_operations mybdrv_fops = {
   .owner = THIS_MODULE,
   .ioctl = my_ioctl,
   .getgeo = my_getgeo
};
...
```

ioctl.c :

```
/* #include <linux/hdreg.h>
struct hd_geometry {
   unsigned char heads;
   unsigned char sectors;
   unsigned short cylinders;
   unsigned long start;
}; */

static int my_getgeo( struct block_device *bdev, struct hd_geometry *geo ) {
   unsigned long sectors = ( diskmb * 1024 ) * 2;
   DBG( KERN_INFO "getgeo\n" );
   geo->heads = 4;
   geo->sectors = 16;
   geo->cylinders = sectors / geo->heads / geo->sectors;
   geo->start = geo->sectors;
   return 0;
};

static int my_ioctl( struct block_device *bdev, fmode_t mode,
                     unsigned int cmd, unsigned long arg ) {
   DBG( "ioctl cmd=%d\n", cmd );
   switch( cmd ) {
      case HDIO_GETGEO: {
         struct hd_geometry geo;
         LOG( "ioctk HDIO_GETGEO\n" );
```

```
            my_getgeo( bdev, &geo );
            if( copy_to_user( (void __user *)arg, &geo, sizeof( geo ) ) )
               return -EFAULT;
            return 0;
         }
         default:
            DBG( "ioctl unknown command\n" );
            return -ENOTTY;
      }
   }
```

Примечание: Используемые в этом архиве макросы `ERR(), LOG(), DBG()` определены **в примерах** только для компактности кода, не должны смущать, и скрывают под собой всего лишь `printk()`:

```
#define ERR(...) printk( KERN_ERR "! "__VA_ARGS__ )
#define LOG(...) printk( KERN_INFO "+ "__VA_ARGS__ )
#define DBG(...) if( debug > 0 ) printk( KERN_DEBUG "# "__VA_ARGS__ )
```

Современное ядро никак не связано с **геометрией** блочного устройства, устройство рассматривается как линейный массив **секторов** (в наших примерах это массив секторов последовательно размещённых в памяти). Для работы операционной системы с блочным устройством не принципиально разбиение его пространства в терминах цилиндров, головок и числа секторов на дорожку. Но достаточно многие утилиты GNU для работы с дисковыми устройствами (`fdisk`, `hdparm`, ...) предполагают получение информации о геометрии диска. Чтобы не создавать неожиданностей при работе с этими утилитами, целесообразно реализовать показанные выше операции (без них может оказаться, что вы не сможете создать разделы на поддерживаемом вашим модулем устройстве). Посмотрим, что получилось в нашей реализации:

```
# insmod block_mod_s.ko
# hdparm -g /dev/xdd
/dev/xdd:
 geometry      = 128/4/16, sectors = 8192, start = 16
```

Примечание: Убедиться в том, что для работы Linux геометрия диска не имеет значения, можно на реальном флэш-диске записав с помощью той же утилиты `hdparm` произвольные значения T/S (число секторов в одной дорожке), H (число головок) и C (число цилиндров), но так, чтобы их произведение сохранило корректный общий размер диска, выраженный в секторах. При этом диск может выглядеть как имеющий совершенно разные геометрии, но при этом сохраняющий нормальную работоспособность. Геометрия диска, очевидно, имеет существенное значение для механических устройств, особенно для пересчёта линейного номера сектора в соответствующий номер цилиндра и перемещение головки. Это может использоваться в оптимизирующем механизме доступа к очереди запросов в ядре, но разработчик модуля не имеет доступа для влияния на этот механизм.

В таблице операций `struct block_device_operations`, конечно, находит некоторое отражение тот факт, что это операции именно блочного устройства. Например, если вы разрабатываете устройство со сменными носителями, то для проверки несменяемости носителя при каждом новом открытии устройства в `open()` нужно вызвать **функцию API ядра** `check_disk_change( struct block_device* )`:

```
int open( struct block_device* dev, fmode_t mode ) {
...
   check_disk_change( dev );
...
}
```

Но для проверки **фактической** смены носителя функция `check_disk_change()` осуществит обратным порядком вызов метода из вашей же таблицы операций:

```
int (*media_changed)( struct gendisk* );
```

Если носитель сменился, `media_changed()` возвращает ненулевое значение. Если будет возвращён признак смены носителя (ненулевое значение), ядро тут же вызовет ещё один метод из таблицы операций:

```
int (*revalidate_disk)( struct gendisk* );
```

Этот, реализованный вами, метод должен сделать всё необходимое, чтобы подготовить драйвера для операций **с новым** носителем (если устройство это RAM диск, как в примерах, то вызов должен, например, обнулить область памяти устройства). После осуществления вызова `revalidate_disk()`, ядро тут же предпримет

попытку перечитать **таблицу разделов** устройства. Так обрабатываются сменные носители в блочных устройствах.

Примечание: Нужно различать устройство со сменным носителем, и сменное устройство. Например, устройства со сменным носителем — это: DVD/RW, Iomega ZIP и подобные устройства. А сменные устройства — это: hotplug жёсткие диски, USB флеш диски, или USB внешние дисковые накопители. Реинициализация драйвера в том и в другом случае происходит совершенно разными способами.

И последние замечания относительно таблицы операций:

1. Обновления и изменения (временами достаточно радикальные) в `struct block_device_operations` происходят довольно часто при изменении версии ядра: например, показанный выше вызов `swap_slot_free_notify()` в таблице (относящейся к версии 3.5) появился только начиная с ядра 2.6.35 (см. http://lxr.free-electrons.com/source/include/linux/blkdev.h?v=2.6.34);
2. При изменениях уже существующие и описанные в публикациях методы объявляются устаревшими (deprecated, они могут быть изъяты в последующем), так объявлен (с версии 2.6.38) обсуждавшийся ранее метод `media_changed()` и вместо него предложен метод `check_events()`, о чём и напоминает комментарий:

```
...
unsigned int (*check_events) (struct gendisk *disk,
                              unsigned int clearing);
/* ->media_changed() is DEPRECATED, use ->check_events() instead */
int (*media_changed) (struct gendisk *);
...
```

Обмен данными

Всё, что было до сих пор — должно быть просто и понятно. Но, при внимательном рассмотрении, изумляет одна вещь: а где же сами операции чтения и записи данных устройства? А вот сами операции чтения и записи в таблице операций блочного устройства и **отсутствуют**! Они выполняются по-другому:

- получив запрос на чтение или запись ядро помещает запрос обслуживания в очередь, связанную с драйвером (которую мы инициализировали ранее);
- запросы в очереди ядро **пересортирует** так, чтобы оптимизировать выполнение последовательности запросов (например, минимизировать перемещения головок жёсткого диска, объединить два или более соседних запроса в один более протяжённый, ...);
- очередной (текущий крайний) запрос из очереди передаётся на обработку **функции драйвера**, которую мы определили для обработки запросов;
- направление (ввод или вывод) и параметры запроса (начальные сектора, число секторов и др.) определены в самом **теле** запроса.

Единичный запрос к блочному драйверу (`struct request`), в общем случае, представляется набором (**вектором**) из **нескольких** операций (`struct bio`, буфер памяти) одного типа (чтение или запись). Сектора для этих операций могут быть разбросаны по диску (не последовательны) в зависимости от характера **файловой системы**. Адреса получателей в памяти этих операций могут тоже быть разбросаны (**вектор ввода-вывода**, IOV). Вот обслуживание таких запросов и является задачей драйвера блочного устройства. Вот, в целом, и основная схема работы драйвера блочного устройства, хотя и очень огрублённо.

Но к этому времени мы рассмотрели все детали, необходимые для того, чтобы рассмотреть и понимать код работающего примера модуля блочного устройства (в архиве `blkdev.tgz` несколько альтернативных примеров, которые рассматриваются для сравнения). Первый обсуждаемый пример `block_mod_s.c` реализует собой блочное устройство в оперативной памяти (самый наглядный и простой для понимания случай).

Этот модуль реализован в файле `block_mod_s.c` с некоторым небольшими включениями, общими для

всех приведенных вариантов блочных устройств, которые почти все были обсуждены выше, и вот последнее из таких включений:

common.h :

```
#include <linux/module.h>
#include <linux/vmalloc.h>
#include <linux/blkdev.h>
#include <linux/genhd.h>
#include <linux/errno.h>
#include <linux/hdreg.h>
#include <linux/version.h>

#define KERNEL_SECTOR_SIZE    512

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,35)
#define blk_fs_request(rq)      ((rq)->cmd_type == REQ_TYPE_FS)
#endif

static int diskmb = 4;
module_param_named( size, diskmb, int, 0 ); // размер диска в Mb, по умолчанию - 4Mb
static int debug = 0;
module_param( debug, int, 0 );              // уровень отладочных сообщений

#define ERR(...) printk( KERN_ERR "! "__VA_ARGS__ )
#define LOG(...) printk( KERN_INFO "+ "__VA_ARGS__ )
#define DBG(...) if( debug > 0 ) printk( KERN_DEBUG "# "__VA_ARGS__ )
```

Здесь, в частности, описан полезный в экспериментах параметр загрузки модуля `diskmb` — это объём создаваемого блочного устройства в мегабайтах.

Код модуля `block_mod_s.c` предполагает реализацию **трёх** альтернативных стратегий реализации обмена данными. Выбор осуществляется при загрузке модуля параметром `mode`. Значение параметра `mode` определяет какая функция обработки будет использоваться. Поэтому (для облегчения понимания), сначала мы рассмотрим полный код модуля, но исключим из него реализации самих обработчиков обмена, а позже по порядку рассмотрим как реализуются обработчики в каждой стратегии:

block_mod_s.c :

```
#include "../common.h"

static int major = 0;
module_param( major, int, 0 );
static int hardsect_size = KERNEL_SECTOR_SIZE;
module_param( hardsect_size, int, 0 );
static int ndevices = 4;
module_param( ndevices, int, 0 );
// The different "request modes" we can use:
enum {  RM_SIMPLE  = 0,       // The extra-simple request function
        RM_FULL    = 1,       // The full-blown version
        RM_NOQUEUE = 2,       // Use make_request
     };
static int mode = RM_SIMPLE;
module_param( mode, int, 0 );

static int nsectors;

struct disk_dev {               // The internal representation of our device.
   int size;                    // Device size in sectors
   u8 *data;                    // The data array
   spinlock_t lock;             // For mutual exclusion */
   struct request_queue *queue; // The device request queue */
```

```
   struct gendisk *gd;          // The gendisk structure */
};

static struct disk_dev *Devices = NULL;
...

static void simple_request( struct request_queue *q ) {   // Простой запрос с обработкой очереди
...
}

static void full_request( struct request_queue *q ) {  // Запрос с обработкой вектора BIO
...
}

static void make_request( struct request_queue *q,
                          struct bio *bio ) { // Прямое выполнение запроса без очереди
...
}

#include "../ioctl.c"
#include "../common.c"

#define MY_DEVICE_NAME "xd"
#define DEV_MINORS    16

static void setup_device( struct disk_dev *dev, int which ) { // Set up our internal device.
   memset( dev, 0, sizeof( struct disk_dev ) );
   dev->size = diskmb * 1024 * 1024;
   dev->data = vmalloc( dev->size );
   if( dev->data == NULL ) {
      ERR( "vmalloc failure.\n" );
      return;
   }
   spin_lock_init( &dev->lock );
   switch( mode ) { // The I/O queue mode
      case RM_NOQUEUE:
         dev->queue = blk_alloc_queue( GFP_KERNEL );
         if( dev->queue == NULL ) goto out_vfree;
         blk_queue_make_request( dev->queue, make_request );
         break;
      case RM_FULL:
         dev->queue = blk_init_queue( full_request, &dev->lock );
         if( dev->queue == NULL ) goto out_vfree;
         break;
      default:
         LOG( "bad request mode %d, using simple\n", mode );
         /* fall into.. */
      case RM_SIMPLE:
         dev->queue = blk_init_queue( simple_request, &dev->lock );
         if( dev->queue == NULL ) goto out_vfree;
         break;
   }
   blk_queue_logical_block_size( dev->queue, hardsect_size );  // Set the hardware sector size
   dev->queue->queuedata = dev;
   dev->gd = alloc_disk( DEV_MINORS );                         // Число разделов при разбиении
   if( ! dev->gd ) {
      ERR( "alloc_disk failure\n" );
      goto out_vfree;
   }
```

```
   dev->gd->major = major;
   dev->gd->minors = DEV_MINORS;
   dev->gd->first_minor = which * DEV_MINORS;
   dev->gd->fops = &mybdrv_fops;
   dev->gd->queue = dev->queue;
   dev->gd->private_data = dev;
   snprintf( dev->gd->disk_name, 32, MY_DEVICE_NAME"%c", which + 'a' );
   set_capacity( dev->gd, nsectors * ( hardsect_size / KERNEL_SECTOR_SIZE ) );
   add_disk( dev->gd );
   return;
out_vfree:
   if( dev->data ) vfree( dev->data );
}

static int __init blk_init( void ) {
   int i;
   nsectors = diskmb * 1024 * 1024 / hardsect_size;
   major = register_blkdev( major, MY_DEVICE_NAME );
   if( major <= 0 ) {
      ERR( "unable to get major number\n" );
      return -EBUSY;
   }
   // Allocate the device array
   Devices = kmalloc( ndevices * sizeof( struct disk_dev ), GFP_KERNEL );
   if( Devices == NULL ) goto out_unregister;
   for( i = 0; i < ndevices; i++ ) // Initialize each device
      setup_device( Devices + i, i );
   return 0;
out_unregister:
   unregister_blkdev( major, MY_DEVICE_NAME );
   return -ENOMEM;
}

static void blk_exit( void ) {
   int i;
   for( i = 0; i < ndevices; i++ ) {
      struct disk_dev *dev = Devices + i;
      if( dev->gd ) {
         del_gendisk( dev->gd );
         put_disk(dev->gd);
      }
      if( dev->queue ) {
         if( mode == RM_NOQUEUE )
            blk_put_queue( dev->queue );
         else
            blk_cleanup_queue( dev->queue );
      }
      if( dev->data ) vfree( dev->data );
   }
   unregister_blkdev( major, MY_DEVICE_NAME );
   kfree( Devices );
}

MODULE_AUTHOR( "Jonathan Corbet" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );

MODULE_LICENSE( "GPL v2" );
MODULE_VERSION( "1.5" );
```

Здесь собрано воедино всё то, что мы разрозненно обсуждали ранее. Нам осталось рассмотреть стратегии обработки, и как это реализуется.

Примечание: Для простоты, начать рассмотрение можно с примеров модулей `block_mod_c.c` и `block_mod_e.c` которые также реализуют равноценные блочные RAM-диски, но делают это только классическим, наиболее часто используемым способом, обслуживанием очереди управляемой ядром.

Классика: очередь и обслуживание ядром

Первый способ (`mode=0`), как уже упоминалось ранее, это почти классика написания модулей блочных устройств (90-95% драйверов, особенно механических дисковых устройств, используют эту стратегию). Обработка осуществляется двумя функциями:

block_mod_s.c :

```
...
static int transfer( struct disk_dev *dev, unsigned long sector,
                     unsigned long nsect, char *buffer, int write ) { // Собственно обмен
   unsigned long offset = sector * KERNEL_SECTOR_SIZE;
   unsigned long nbytes = nsect * KERNEL_SECTOR_SIZE;
   if( (offset + nbytes) > dev->size ) {
      ERR( "beyond-end write (%ld %ld)\n", offset, nbytes );
      return -EIO;
   }
   if( write )
      memcpy( dev->data + offset, buffer, nbytes );
   else
      memcpy( buffer, dev->data + offset, nbytes );
   return 0;
}

static void simple_request( struct request_queue *q ) {   // Простой запрос с обработкой очереди
   struct request *req;
   unsigned nr_sectors, sector;
   DBG( "entering simple request routine\n" );
   req = blk_fetch_request( q );
   while( req ) {
      int ret = 0;
      struct disk_dev *dev = req->rq_disk->private_data;
      if( !blk_fs_request( req ) ) {          // не валидный запрос
         ERR( "skip non-fs request\n" );
         __blk_end_request_all( req, -EIO );
         req = blk_fetch_request( q );
         continue;
      }
      nr_sectors = blk_rq_cur_sectors( req ); // валидный запрос - обработка
      sector = blk_rq_pos( req );
      ret = transfer( dev, sector, nr_sectors, req->buffer, rq_data_dir( req ) );
      if( !__blk_end_request_cur( req, ret ) )
         req = blk_fetch_request( q );
   }
}
...
```

Ещё не обработанные (или не завершённые ранее) запросы выбираются из очереди ядра (`request_queue`) вызовом:

```
struct request* req blk_fetch_request( struct request_queue* );
```

Запрос может требовать выполнения специальных (управляющих) операций, не являющихся чтением или

записью, и переданными «не тому» устройству — такие запросы отфильтровываются вызовом
```
bool blk_fs_request( struct request* );
```

Если же запрос признан валидным, то из него извлекаются начальный сектор для обмена (`blk_rq_pos( struct request* )`), число секторов подлежащих обмену (`blk_rq_cur_sectors( struct request* )`), буфер (`(struct request*)->buffer` — в пространстве ядра!) и направление требуемого обмена (`rq_data_dir( struct request* )` — запись или чтение). Далее управление передаётся функции драйвера `transfer()`. Эта наша функция, проверив предварительно, что данные запрошены не за пределами протяжённости пространства устройства (это была бы грубая ошибка), осуществляет копирование данных в или из буфера ядра. Всё дальнейшее «разбрасывание» полученных данных получателю запроса **делает ядро**.

Независимо от того, успешно или по ошибке завершился очередной запрос, ядру нужно сообщить о его завершении и переходе к следующему запросу из очереди. Уведомление ядра выполняют вызовы:

```
__blk_end_request_all( struct request*, int errno );
__blk_end_request_cur( struct request*, int errno );
```

Здесь в `errno` сообщается результат операции: 0 в случае успеха, отрицательный код (как принято везде в ядре) в случае ошибки. После этих вызовов текущий запрос считается обработанным и удаляется из очереди.

Примечание: В API блочных операций практически всем вызовам «с подчёркиванием» существуют парные вызовы «без подчёркивания», например, `__blk_end_request_all()` соответствует `blk_end_request_all()`. Разница в том, что первые (с `__`) сами захватывают блокировку монопольного использования очереди (помните, мы инициализировали её в самом начале, вместе с очередью?). А для вторых это должен обеспечить программист функции, они должны выполняться с уже захваченной блокировкой, но это предоставляет дополнительную гибкость разработчику. Но если вы механически замените в коде форму «с подчёркиванием» на «без подчёркивания» без соответствующих изменений в коде — то это прямой путь тут же «подвесить» операционную систему в целом.

Вот, собственно, и вся «классическая» стратегия. Она описана специально очень поверхностно, потому, что детальный разбор того, что происходит с очередью и запросом — это очень увлекательное исследование, но оно не нужно для того, чтобы писать драйвер: показанный код (с минимальными вариациями) является **шаблоном** для написания обработчиков.

Очередь и обработка запроса в драйвере

Следующий вариант (`mode=1`) оказывается сложнее по смыслу происходящего, но столь же прост в реализации.

Каждая структура `struct request` (единичный **блочный** запрос в очереди) представляет собой один запрос ввода/вывода, хотя этот запрос может быть образован как результат слияния нескольких самостоятельных запросов, проделанного планирующими механизмами ядра (оптимизацией запросов). Секторы, являющиеся адресатом любого конкретного запроса, могут быть распределены по всей оперативной памяти получателя (представлять вектор ввода/вывода), хотя они всегда соответствуют на блочном устройстве набору последовательных **секторов**. Запрос (`struct request`) представлен в виде **вектора** сегментов, каждый из которых соответствует одному буферу в памяти (`struct bio`). Ядра может объединить в один запрос несколько элементов вектора, связанных со смежными секторами на диске. Структура `struct bio` является низкоуровневым описанием элемента запроса блочного ввода/вывода (`struct request`) — такого уровня детализации нам уже достаточно, чтобы продолжить писать драйвер.

В этой стратегии функциями самого драйвера будет производится последовательная обработка всей последовательности `struct bio`, переданных в составе единичного `struct request`:

block_mod_s.c :

```
...
static int xfer_bio( struct disk_dev *dev, struct bio *bio ) {   // Передача одиночного BIO.
   int i, ret;
   struct bio_vec *bvec;
   sector_t sector = bio->bi_sector;
   DBG( "entering xfer_bio routine\n" );
```

```
   bio_for_each_segment( bvec, bio, i ) { // Работаем с каждым сегментом независимо.
      char *buffer;
      sector_t nr_sectors;
#if LINUX_VERSION_CODE < KERNEL_VERSION(3,11,0)
      buffer = __bio_kmap_atomic( bio, i, KM_USER0 );
#else
      buffer = __bio_kmap_atomic( bio, i );
#endif
      nr_sectors = bio_sectors( bio );
      ret = transfer( dev, sector, nr_sectors, buffer, bio_data_dir( bio ) == WRITE );
      if( ret != 0 ) return ret;
      sector += nr_sectors;
#if LINUX_VERSION_CODE < KERNEL_VERSION(3,11,0)
      __bio_kunmap_atomic( bio, KM_USER0 );
#else
      __bio_kunmap_atomic( bio );
#endif
   }
   return 0;
}

static int xfer_request( struct disk_dev *dev, struct request *req ) { // Передача всего запроса.
   struct bio *bio;
   int nsect = 0;
   DBG( "entering xfer_request routine\n" );
   __rq_for_each_bio( bio, req ) {
      xfer_bio( dev, bio );
      nsect += bio->bi_size / KERNEL_SECTOR_SIZE;
   }
   return nsect;
}

static void full_request( struct request_queue *q ) { // запрос с обработкой вектора BIO
   struct request *req;
   int sectors_xferred;
   DBG( "entering full request routine\n" );
   req = blk_fetch_request( q );
   while( req ) {
      struct disk_dev *dev = req->rq_disk->private_data;
      if( !blk_fs_request( req ) ) {                  // невалидный запрос
         ERR( "skip non-fs request\n" );
         __blk_end_request_all( req, -EIO );
         req = blk_fetch_request( q );
         continue;
      }
      sectors_xferred = xfer_request( dev, req );     // валидный запрос - обработка
      if( !__blk_end_request_cur( req, 0 ) )
         req = blk_fetch_request( q );
   }
}
```

Код обработчика `full_request()` чрезвычайно похож на предыдущий случай `simple_request()` (а как могло быть иначе? — их даже можно было бы объединить в единый код), только вместо передачи `transfer()` вызывается `xfer_request()`, которая, последовательно пробегая для каждой `struct bio` в запросе, вызывает для каждого элемента вектора `xfer_bio()`. Эта функция извлекает параметры, и осуществляет обмен данными для единичного сегмента, используя уже знакомую нам функцию `transfer()`.

Отказ от очереди

Очереди запросов в ядре реализуют интерфейс подключения модулей, которые представляют собой различные планировщики ввода/вывода (транспортёр, elevator). Работой планировщика ввода/вывода является переупорядочение очереди так, чтобы предоставлять запросы ввода/вывода драйверу в такой последовательности, которая максимизировала бы производительность. Планировщик ввода/вывода также отвечает за объединение прилегающих запросов: когда в планировщик ввода/вывода получает новый запрос, он ищет в очереди запросы, относящиеся к прилегающим секторам, если таковые нашлись, он объединяет такие запросы (если результирующий запрос в не будет нарушать некоторые условия, например, не будет слишком большим). В некоторых случаях, когда производительность не зависит от расположения секторов и последовательности обращений (RAM-диски, флеш память, ...) можно отказаться от планировщика, предоставив для очереди свой планировщик, который будет немедленно получать запрос по его поступлений.

Это третья (`mode=2`) из рассматриваемых нами стратегий (отказ от очереди). В этом случае мы совершенно по другому создаём и инициализируем очередь в ядре для устройства:

```
...
   case RM_NOQUEUE:
         dev->queue = blk_alloc_queue( GFP_KERNEL );
         if( dev->queue == NULL ) goto out_vfree;
         blk_queue_make_request( dev->queue, make_request );
         break;
...
```

Мы устанавливаем для такой очереди свой обработчик всех единичных сегментов (`struct bio`), требующих обслуживания (без оптимизации, слияний и т.п.):

```
#if LINUX_VERSION_CODE <= KERNEL_VERSION(3,1,0)
static int make_request( struct request_queue *q, struct bio *bio ) {
#else
static void make_request( struct request_queue *q, struct bio *bio ) {
#endif
   struct disk_dev *dev = q->queuedata; // прямое выполнение запроса без очереди
   int status = xfer_bio( dev, bio );
   bio_endio( bio, status );
#if LINUX_VERSION_CODE <= KERNEL_VERSION(3,1,0)
   return 0;
#endif
}
```

Функцию `xfer_bio()` мы уже видели ранее, в предыдущем варианте.

Вот на этом и исчерпываются наши варианты организации обработки потока данных в модуле блочного устройства.

Пример перманентных данных

Ещё одним примеров, представленным в архиве, является модуль блочного устройства `dubfl.c` — здесь носитель данных изменён из оперативной памяти на реальный дисковый файл (предварительно созданный). Таким образом наш RAM-диск превращается в перманентный, сохраняющим своё содержимое между последовательными включениями и выключениями питания компьютера. Это позволяет производительнее экспериментировать с кодом модуля. Кроме того, код модуль демонстрирует некоторые любопытные детали работы с файловыми системами из модуля ядра, ... но это выходит за рамки нашего рассмотрения.

Важной отличительной особенностью этой реализации есть то, что она позволяет продемонстрировать и изучить технику **сменных носителей**: данные из внешнего файла загружаются при монтировании (открытии) устройства, и сохраняются в файл с произведенными в сеансе изменениями, когда устройство закрывается.

Некоторые важные API

Неприятностью является то, что API блочных устройств достаточно быстро изменяется, то, что было работоспособно ещё в ядре, скажем, 2.6.29 — уже даже не компилируется в ядре 3.0. Некоторую (но недостаточно полную) справку по API блочных устройств в ядре можно найти по ссылке: http://www.hep.by/gnu/kernel/kernel-api/blkdev.html

Результаты тестирования

Теперь мы можем закончить рассмотрение созданных модулей блочных устройств объёмным тестированием того, что получено:

```
# insmod block_mod_s.ko
# ls -l /dev/x*
brw-rw---- 1 root disk 252,  0 нояб. 13 02:01 /dev/xda
brw-rw---- 1 root disk 252, 16 нояб. 13 02:01 /dev/xdb
brw-rw---- 1 root disk 252, 32 нояб. 13 02:01 /dev/xdc
brw-rw---- 1 root disk 252, 48 нояб. 13 02:01 /dev/xdd
# hdparm -g /dev/xdd
/dev/xdd:
 geometry      = 128/4/16, sectors = 8192, start = 16
# mkfs.vfat /dev/xdd
mkfs.vfat 3.0.12 (29 Oct 2011)
```

Создадим некоторую структуру разделов с помощью `fdisk` (детали процесса не показаны из экономии — всё делается тривиальным способом), после чего:

```
# fdisk -l /dev/xda
Диск /dev/xda: 4 МБ, 4194304 байт
4 heads, 16 sectors/track, 128 cylinders, всего 8192 секторов
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x2fb10b5b

Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/xda1               1        2000        1000   83  Linux
/dev/xda2            2001        5000        1500    5  Расширенный
/dev/xda3            5001        8191        1595+  83  Linux
/dev/xda5            2002        3500         749+  83  Linux
/dev/xda6            3502        5000         749+  83  Linux
# ls -l /dev/x*
brw-rw---- 1 root disk 252,  0 нояб. 13 02:11 /dev/xda
brw-rw---- 1 root disk 252,  1 нояб. 13 02:11 /dev/xda1
brw-rw---- 1 root disk 252,  2 нояб. 13 02:11 /dev/xda2
brw-rw---- 1 root disk 252,  3 нояб. 13 02:11 /dev/xda3
brw-rw---- 1 root disk 252,  5 нояб. 13 02:11 /dev/xda5
brw-rw---- 1 root disk 252,  6 нояб. 13 02:11 /dev/xda6
brw-rw---- 1 root disk 252, 16 нояб. 13 02:01 /dev/xdb
brw-rw---- 1 root disk 252, 32 нояб. 13 02:01 /dev/xdc
brw-rw---- 1 root disk 252, 48 нояб. 13 02:06 /dev/xdd
```

Отформатируем пару разделов на `/dev/xda` (диск `/dev/xdd` я уже отформатировал ранее в FAT-32 без разметки на разделы, как дискету):

```
# mkfs.ext2 /dev/xda1
mke2fs 1.42.3 (14-May-2012)
...
```

```
# fsck /dev/xda1
fsck из util-linux 2.21.2
e2fsck 1.42.3 (14-May-2012)
/dev/xda1: clean, 11/128 files, 38/1000 blocks
# mkfs.ext2 /dev/xda5
mke2fs 1.42.3 (14-May-2012)
...
# fsck /dev/xda5
fsck из util-linux 2.21.2
e2fsck 1.42.3 (14-May-2012)
# fsck /dev/xdd
fsck из util-linux 2.21.2
dosfsck 3.0.12, 29 Oct 2011, FAT32, LFN
/dev/xdd: 0 files, 0/2036 clusters
```

Теперь мы можем смонтировать диски (из числа отформатированных) в заранее созданные каталоги:

```
# ls /mnt
dska  dskb  dskc  dskd  dske  efi  iso
# mount -tvfat /dev/xdd /mnt/dskd
# mount -text2 /dev/xda1 /mnt/dska
# mount -text2 /dev/xda5 /mnt/dskb
# mount | grep /xd
/dev/xdd on /mnt/dskd type vfat (rw,relatime,fmask=0022,dmask=0022,codepage=cp437,
iocharset=ascii,shortname=mixed,errors=remount-ro)
/dev/xda1 on /mnt/dska type ext2 (rw,relatime)
/dev/xda5 on /mnt/dskb type ext2 (rw,relatime)
# df | grep /xd
/dev/xdd              4072            0     4072            0% /mnt/dskd
/dev/xda1              979           17      912            2% /mnt/dska
/dev/xda5              731           16      678            3% /mnt/dskb
```

И проверим файловые операции копирования на этих файловых системах (я уже не один раз повторял, что нет лучше POSIX тестов, чем стандартные GNU утилиты):

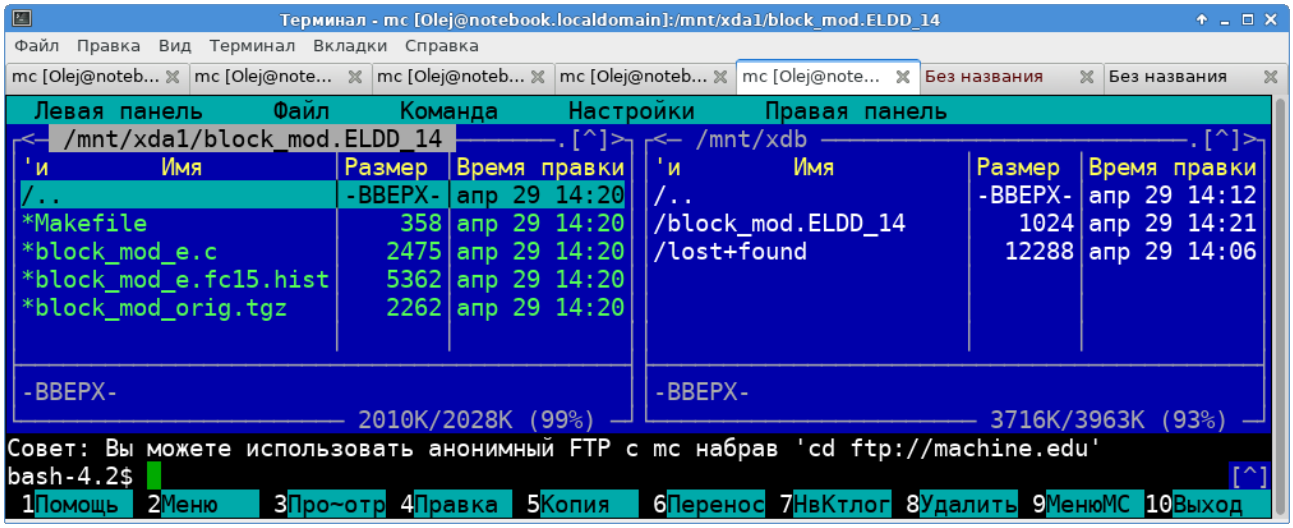
```
# echo ++++++++++++++++++++++ > /mnt/dska/f1
# cp /mnt/dska/f1 /mnt/dskb/f2
# cp /mnt/dskb/f2 /mnt/dskd/f3
# cat /mnt/dskd/f3
++++++++++++++++++++++
# tree /mnt/dsk*
/mnt/dska
|-- f1
`-- lost+found
/mnt/dskb
|-- f2
`-- lost+found
/mnt/dskc
/mnt/dskd
`-- f3
/mnt/dske
2 directories, 3 files
```

В итоге мы должны получить дисковые устройства, с точки зрения **любых** утилит Linux ничем не отличающиеся от существующих дисковых устройств, например, выполнение операций группового копирования:

```
$ sudo cp block_mod.ELDD_14 /mnt/xda1 -R
$ sudo cp block_mod.ELDD_14 /mnt/xda2 -R
$ sudo cp block_mod.ELDD_14 /mnt/xdb -R
$ df | grep xd
/dev/xda1             2028           18     2010            1% /mnt/xda1
/dev/xda2             2028           18     2010            1% /mnt/xda2
```

```
/dev/xdb              3963          43      3716            2% /mnt/xdb
$ tree /mnt
/mnt
├── sysimage
│   └── home
├── xda1
│   └── block_mod.ELDD_14
│       ├── block_mod_e.c
│       ├── block_mod_e.fc15.hist
│       ├── block_mod_orig.tgz
│       └── Makefile
├── xda2
│   └── block_mod.ELDD_14
│       ├── block_mod_e.c
│       ├── block_mod_e.fc15.hist
│       ├── block_mod_orig.tgz
│       └── Makefile
└── xdb
    ├── block_mod.ELDD_14
    │   ├── block_mod_e.c
    │   ├── block_mod_e.fc15.hist
    │   ├── block_mod_orig.tgz
    │   └── Makefile
    └── lost+found [error opening dir]

9 directories, 12 files
```



И ещё полезный пример, подтверждающий то, что реализация операций `open()` и `release()` для блочного устройства вовсе не являются критически необходимой (за исключением случаев, когда они обеспечивают специальные цели, такие, например, как подсчёт пользователей, работающих с устройством):

```
# lsmod | head -n4
Module                  Size  Used by
vfat                   17208  1
fat                    54611  1 vfat
block_mod_s            13210  4
# rmmod block_mod_s
Error: Module block_mod_s is in use
```

Число счётчика ссылок на модуль не нулевое! ... естественно, до тех пор, пока мы не размонтируем все используемые диски этого модуля:

```
# umount /mnt/dska
# umount /mnt/dskb
# umount /mnt/dskd
# lsmod | grep ^block
block_mod_s            13210  0
# rmmod block_mod_s
# ls /dev/xd*
ls: невозможно получить доступ к /dev/xd*: Нет такого файла или каталога
```

А вот что показывает **сравнительное** тестирование скорости созданных дисковых устройств, выполненное типовой GNU утилитой, для разных выбранных стратегий обработки запросов ввода/вывода:

```
# insmod block_mod_s.ko mode=0
# hdparm -t /dev/xda
/dev/xda:
 Timing buffered disk reads:   4 MB in  0.01 seconds = 318.32 MB/sec

# insmod block_mod_s.ko mode=1
# hdparm -t /dev/xda
/dev/xda:
 Timing buffered disk reads:   4 MB in  0.03 seconds = 116.02 MB/sec

# insmod block_mod_s.ko mode=2
# hdparm -t /dev/xda
/dev/xda:
 Timing buffered disk reads:   4 MB in  0.01 seconds = 371.92 MB/sec
```

Видны весьма значительные различия в производительности ... , но это уже предмет для совсем другого рассмотрения.

И ещё несколько примеров с перманентным сохранением данных:

```
$ time dd if=/dev/zero of=./XXX bs=512 count=10000
10000+0 записей считано
10000+0 записей написано
 скопировано 5120000 байт (5,1 MB), 0,0414243 с, 124 MB/с
real    0m0.086s
user    0m0.007s
sys     0m0.049s
$ ls -l XXX
-rw-rw-r--. 1 olej olej 5120000 апр.  26 16:18 XXX
```

Здесь мы создали «сырой» файл размером 5Mb, который и будет образом носителя для нашего будущего блочного устройства.

Загружаем модуль блочного устройства, который загружает и выгружает свои данные в этот файл между сеансами своего использования:

```
$ sudo insmod dubfl.ko file=XXX
$ ls -l /dev/db*
brw-rw----. 1 root disk 252, 1 апр.  26 16:20 /dev/dbf
$ sudo hdparm -t /dev/dbf
/dev/dbf:
 Timing buffered disk reads:   4 MB in  0.01 seconds = 393.04 MB/sec
$ sudo mkfs.vfat /dev/dbf
mkfs.vfat 3.0.12 (29 Oct 2011)
$ ls /mnt
common
$ sudo mount -t vfat /dev/dbf /mnt/common/
$ df | grep dbf
/dev/dbf              4974            0     4974            0% /mnt/common
$ sudo cp -R block_mod.ELDD_14 /mnt/common
$ tree /mnt/common/
```

```
/mnt/common/
└── block_mod.ELDD_14
    ├── block_mod_e.c
    ├── block_mod_e.fc15.hist
    ├── block_mod_e.ko
    ├── block_mod_orig.tgz
    └── Makefile
 1 directory, 5 files
$ lsmod | head -n5
Module                  Size  Used by
vfat                   17209  1
fat                    54645  1 vfat
dubfl                  13074  1
tcp_lp                 12584  0
$ sudo umount /mnt/common
$ lsmod | head -n5
Module                  Size  Used by
vfat                   17209  0
fat                    54645  1 vfat
dubfl                  13074  0
tcp_lp                 12584  0
$ sudo rmmod dubfl
```

Здесь мы выгрузили модуль из памяти, но мы можем, для достоверности, вообще перезагрузить компьютер, а затем возобновить работу со своими данными:

```
$ sudo reboot
...
$ sudo insmod dubfl.ko file=XXX
$ sudo mount -t vfat /dev/dbf /mnt/common/
$ ls -R /mnt/common
/mnt/common:
block_mod.ELDD_14
/mnt/common/block_mod.ELDD_14:
block_mod_e.c  block_mod_e.fc15.hist  block_mod_e.ko  block_mod_orig.tgz  Makefile
$ df | grep dbf
/dev/dbf              4974          176     4798            4% /mnt/common
```

Задачи

1. Напишите свой собственный драйвер RAM-диска (по любой из описываемых схем). Проведите для него полный цикл испытаний и тестирования: разбиения разделов, создание файловых систем, создание файловых иерархий, контроль производительности...

6. Интерфейс /proc

Интерфейс к файловым именам /proc (procfs) и более поздний (по времени создания) интерфейс к именам /sys (sysfs) рассматривается как канал передачи диагностической (из) и управляющей (в) информации для модуля. Такой способ взаимодействия с модулем может полностью заменить средства вызова ioctl() для устройств, который устаревший и считается опасным (из-за отсутствия контроля типизации в ioctl()).

Любое изменение в состояниях подсистемы ввода-вывода: загрузка нового модуля ядра, горячее подключение-отключение реального устройства и многие другие события будут порождать многочисленные **автоматические** изменения в иерархии файловых структур /proc и /sys, и изменения содержимого их псевдофайлов. Но эти изменения происходят **автоматически**, независимо от выполнения кода пользователя (средствами sysfs, сокета netlink и подсистемы udev, которые уже упоминались). Нас же в этой части рассмотрения будут интересовать возможности **ручного** создания путевых имён в /proc и /sys (или древовидной иерархии имён). Эти имена должны создаваться **сверх** действий sysfs, из кода загружаемого пользовательского модуля. Такие именованные псевдофайлы и должны служить шлюзами из пользовательского пространства к значениям переменных внутри кода модуля.

В настоящее время сложилась тенденция многие управляющие функции переносить их /proc в /sys, отображения путевых имён модулем в эти две подсистемы по своему назначению и возможностям являются очень подобными. Содержимое имён-псевдофайлов в обоих системах является только **текстовым** отображением некоторых внутренних данных ядра. Но нужно иметь в виду и ряд отличий между ними:

- Файловая система /proc является общей, «родовой» принадлежностью всех UNIX систем (Free/Open/Net BSD, Solaris, QNX, MINIX 3, ...), её наличие и общие принципы использования оговариваются стандартом POSIX 2; а файловая система /sys является сугубо Linux «изобретением» и используется только этой системой.
- Так сложилось по традиции, что немногочисленные диагностические файлы в /proc содержат зачастую большие таблицы текстовой информации, в то время, как в /sys создаётся много больше по числу имён, но каждое из них даёт только информацию об единичном значении (в символьном представлении!), часто соответствующем одной элементарной переменной языка C: int, long, ...

Сравним:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 14
model name      : Genuine Intel(R) CPU           T2300  @ 1.66GHz
stepping        : 8
cpu MHz         : 1000.000
...
$ wc -l /proc/cpuinfo
58 cpuinfo
```

- это 58 строк текста.

А вот образец информации (выбранной достаточно наугад) системы /sys:

```
$ tree /sys/module/cpufreq
/sys/module/cpufreq
└── parameters
    ├── debug
    └── debug_ratelimit
1 directory, 2 files
$ cat /sys/module/cpufreq/parameters/debug
0
$ cat /sys/module/cpufreq/parameters/debug_ratelimit
```

```
1
```

Различия в форматном представлении информации, часто используемой в той или иной файловой системе, породили заблуждение (мне приходилось не раз это слышать), что интерфейс в `/proc` создаётся только для чтения, а интерфейс `/sys` для чтения и записи. Это совершенно неверно, оба интерфейса допускают и чтение и запись.

Значения в `/proc` и `/sys`

Ещё одна из особенностей, которая вообще присуща философии UNIX, но особенно явно проявляется при работе с именами в `/proc` и, **особенно**, в `/sys`: все считываемые и записываемые значения представляются в **символьном**, не **бинарном**, формате. И здесь, при преобразовании символьных строк в числовые значения в коде ядра, возникает определённое замешательство: в API ядра нет многочисленных вызовов, подобных POSIX API `atol()` и других подобных. Но здесь на помощь приходят функции, определенные в `<linux/kernel.h>`:

```
extern unsigned long simple_strtoul( const char *cp, char **endp, unsigned int base );
extern long simple_strtol( const char *cp, char **endp, unsigned int base );
extern unsigned long long simple_strtoull( const char *cp, char **endp, unsigned int base );
extern long long simple_strtoll( const char *cp, char **endp, unsigned int base );
```

Здесь везде:

`cp` — преобразовываемая символьная строка;

`endp` — возвращаемый указатель на позицию, где завершилось преобразование (символ, который не мог быть преобразован) — может быть и `NULL`, если не требуется;

`base` — система счисления (нулевое значение в этой позиции эквивалентно 10);

Возвращают все эти вызовы преобразованные целочисленные значения. Примеры их возможного использования:

```
long j = simple_strtol( "-1000", NULL, 16 );
long j = simple_strtol( "-1000 значение", pchr, 0 );
```

Для более сложных (но и более склонных к ошибкам) преобразований там же определены функции, подобные эквивалентам в POSIX:

```
extern int sscanf( const char *cp, const char *format, ... ) ;
extern int vsscanf( const char *cp, const char *format, va_list ap );
```

Функции возвращают количество успешно преобразованных и назначенных полей. Примеры:

```
    char str[ 80 ];
    int d, y, n;
    ...
    n = sscanf( "January 01 2000", "%s%d%d", str, &d, &y );
    ...
    sscanf( "24\tLewis Carroll", "%d\t%s", &d, str );
```

Параметры, как легко видеть из примеров, должны поступать в вызов `sscanf()` **по ссылке** (а контроль соответствия типов, из-за переменного числа параметров, отсутствует, что и служит источником ошибок, и, поэтому требует повышенной тщательности).

Обратные преобразования (численные значения в строку) производятся, как мы уже неоднократно видели, вызовом `sprintf()`.

Аналогичные преобразования форматов представлений, естественно, возникают не только при работе с файловыми системами `/proc` и `/sys`, та же необходимость возникает, временами, при работе с устройствами `/dev` и с сетевой подсистемой... Но при работе с именами в `/proc` и, как уже было отмечено, особенно, в `/sys` эта потребность возникает постоянно. Поэтому сделать акцент на форматные преобразования показалось уместным именно здесь.

Использование /proc

Описываемая далее программная техника организации интерфейса в /proc использует API и структуры данных, используемые в ядре до версии 3.9 (включительно). Все описания для этого сосредоточены в заголовочном файле <linux/proc_fs.h>. После версии ядра 3.10 API и структуры активно изменяются, описание ключевой структуры вообще вынесено из <include> и находится в Linux/fs/proc/internal.h в дереве исходных кодов ядра.

Примечание: Теперь главное различие при написании модулей ядра после версии 3.10 выражается в том, что вынесенное теперь в Linux/fs/proc/internal.h описание struct proc_dir_entry недоступно вашему коду из хедер-файлов, вы не можете ссылаться непосредственно к полям этой основной структуры, а можно только использовать функции доступа, предоставляемые в <linux/proc_fs.h>. Эффекты этого изменения мы увидим в коде далее.

По аналогии с тем, как все операции в /dev обеспечивались через таблицу файловых операций (struct file_operations), все операции над именами в /proc увязываются (определения в <linux/proc_fs.h>) через специфическую для этих целей, довольно обширную структуру (показаны только поля, интересные для последующего рассмотрения):

```
struct proc_dir_entry {
...
        unsigned short namelen;
        const char *name;
        mode_t mode;
...
        uid_t uid;
        gid_t gid;
        loff_t size;
...
        /*
         * NULL ->proc_fops means "PDE is going away RSN" or
         * "PDE is just created". In either case, e.g. ->read_proc won't be
         * called because it's too late or too early, respectively.
         *
         * If you're allocating ->proc_fops dynamically, save a pointer
         * somewhere.
         */
        const struct file_operations *proc_fops;
...

        read_proc_t *read_proc;
        write_proc_t *write_proc;
...
};
```

Но что отличает эту структуру, так это то, что она из числа очень старых образований Linux, с ней происходили многочисленные изменения (и накладки), на неё наложились требования совместимости снизу-вверх с предыдущими реализациями.

Наличие в структуре **одновременно** полей read_proc и write_proc, а одновременно с ними уже известной нам таблицы файловых операций proc_fops может быть подсказкой, и так оно оказывается на самом деле — для описания операций ввода-вывода над /proc существуют два **альтернативных** (на выбор) способов их определения:

1. Используя **специфический** для имён в /proc программный интерфейс интерфейс в виде функций:

```
typedef int (read_proc_t)( char *page, char **start, off_t off,
                           int count, int *eof, void *data );
typedef int (write_proc_t)( struct file *file, const char __user *buffer,
                            unsigned long count, void *data );
```

Этот интерфейс (`read_proc()` и `write_proc()`) исчезают только в ядре 3.10, и начиная с этого времени всё, о чём мы будем говорить как о **старом** (или **специфическом**) интерфейсе /proc становится **недоступным**.

2. Используя **общий** механизм доступа к именами файловой системы, а именно таблицу файловых операций `proc_fops` в составе структуры ;

Теперь, когда мы кратно пробежались на качественном уровне по свойствам интерфейсов, можно перейти к примерам кода модулей, реализующих обсуждаемые механизмы. Интерфейс `/proc` рассматривается на примерах из архива `proc.tgz`.

Специфический механизм procfs[12]

Первое, что явно бросается в глаза, это радикальной различие прототипов функций `read_proc_t` и `write_proc_t` (начиная с того, что, как будет показано далее, первая из них вообще не оперирует с адресами пользовательского пространствами, а вторая делает именно это). Это связано и с тем, что функция записи типа `write_proc_t` появилась в API ядра значительно позже, чем функция чтения `read_proc_t`, и по своей семантике значительно проще последней. Пока мы сосредоточимся на функции чтения, а позже вернёмся в два слова и к записи.

Мы будем собирать целую линейку однотипных модулей, поэтому общую часть определений снесём в отдельный файл:

common.h :

```
#define NAME_DIR  "mod_dir"
#define NAME_NODE "mod_node"
#define LEN_MSG 160

#define LOG(...) printk( KERN_INFO "! "__VA_ARGS__ )
#define ERR(...) printk( KERN_ERR "! "__VA_ARGS__ )
```

mod_proc.h :

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/stat.h>
#include <asm/uaccess.h>
#include "common.h"

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

static int  __init proc_init( void );
static void __exit proc_exit( void );

module_init( proc_init );      // предварительные определения
module_exit( proc_exit );
```

Сценарий (файл `Makefile`) сборки многочисленных модулей этого проекта — традиционный (архив `proc.tgz`), поэтому не будем на нём останавливаться.

Основную работу по созданию и уничтожению имени в `/proc` (независимо от используемого дальше интерфейса ввода-вывода) выполняет пара вызовов (`<linux/proc_fs.h>`) для ядра ранее 3.10:

```
struct proc_dir_entry *create_proc_entry( const char *name, mode_t mode,
                                          struct proc_dir_entry *parent );
void remove_proc_entry( const char *name, struct proc_dir_entry *parent );
```

12 Если вас не интересует состояние дел и ваши возможности в ядре раньше 3.10, вы можете совершенно безболезненно пропустить весь этот раздел.

Операции создания сменились и расширились после 3.10 (и это является причиной тяжёлых ошибок компиляции), теперь функции создания каталога и терминального имени разделились и выглядят теперь так:

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

Кроме того, там вы найдёте ещё несколько функций вариантов, позволяющих установить флаги имени и другие опции. Заметим пока, что в создание терминального имени добавился обязательный последний параметр, указывающий на таблицу файловых операций, к которой мы будем ещё неоднократно обращаться.

Эти операции создания и уничтожения, как будет показано далее, оперируют в равной мере как с новыми создаваемыми **каталогами** в `/proc` (создание иерархии имён), так и с конечными **терминальными** именами, являющимися источниками данных, то есть, имя структуры `proc_dir_entry` не должно вводить в заблуждение. Обеспечивается такая универсальность вторым параметром (`mode`) вызова `create_proc_entry()`. В результате такого вызова и создаётся структура **описания имени**, которую мы уже видели ранее:

```
struct proc_dir_entry {
...
        read_proc_t *read_proc;
        write_proc_t *write_proc;
...
};
```

Первый пример показывает создание интерфейса к модулю в `/proc` доступного только для чтения из пользовательских программ (наиболее частый на практике случай):

mod_procr.c :

```
#include "mod_proc.h"
#include "proc_node_read.c"

static int __init proc_init( void ) {
   int ret;
   struct proc_dir_entry *own_proc_node;
   own_proc_node = create_proc_entry( NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO, NULL );
   if( NULL == own_proc_node ) {
      ret = -ENOENT;
      ERR( "can't create /proc/%s", NAME_NODE );
      goto err_node;
   }
   own_proc_node->uid = own_proc_node->gid = 0;
   own_proc_node->read_proc = proc_node_read;
   LOG( "/proc/%s installed", NAME_NODE );
   return 0;
err_node:
   return ret;
}

static void __exit proc_exit( void ) {
   remove_proc_entry( NAME_NODE, NULL );
   LOG( "/proc/%s removed", NAME_NODE );
}
```

Здесь и далее, флаги прав доступа к файлу вида `S_I*` (известные и из пользовательского API) — ищите и заимствуйте в `<linux/stat.h>`. Напомним только, что название результирующей структуры `struct proc_dir_entry`, создаваемой при регистрации имени функцией `create_proc_entry()` (и такого же типа 3-й параметр вызова в этой функции), не должно вводить в заблуждение: это не обязательно каталог, будет ли это каталог, или терминальное файловое имя — определяется значением флагов (параметр `mode`). Относительно 3-го параметра вызова `parent` отметим, что это, как прямо следует из имени, **ранее созданный** родительский каталог в `/proc`, внутри которого создаётся имя, если он равен `NULL`, то имя создаётся непосредственно в

/proc. Эта техника позволяет создавать произвольной сложности иерархии имён в /proc. Всё это хорошо видно в кодах примеров.

Сама реализация функции чтения (типа read_proc_t) будет использована в различных модулях, и вынесена в файл proc_node_read.c:

proc_node_read.c :

```
#include "common.h"

// в точности списан прототип read_proc_t из <linux/proc_fs.h> :
static ssize_t proc_node_read( char *buffer, char **start, off_t off,
                               int count, int *eof, void *data ) {
   static char buf_msg[ LEN_MSG + 1 ] =
          ".........1.........2.........3.........4.........5.........6\n";
   LOG( "read: %d (buffer=%p, off=%ld)", count, buffer, off );
   strcpy( buffer, buf_msg );
   LOG( "return bytes: %d%s", (int)strlen( buf_msg ), *eof != 0 ? " ... EOF" : "" );
   return strlen( buf_msg );
};
```

Для этой реализация оказывается **достаточно** фактически единственную операцию strcpy(), независимо от размера запрашиваемых данных, всё это требует некоторых более детальных пояснений:

- интерфейс read_proc_t развивается давно, и приобрёл за это время довольно причудливую семантику, отражаемую множеством используемых параметров;
- в функции read_proc_t **не осуществляется** обмен между пространствами ядра и пользователя, она только передаёт данные на некоторый **промежуточный слой** ядра, который обеспечивает позиционирование данных, их дробление в соответствии с запрошенным размером, и обмен данными с пространством пользователя;
- функция ориентирована на одно-**страничный** обмен памятью, почему её прототип записывается так (первый параметр):

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

- но при соблюдении условий что: а). объём передаваемых данных укладывается **в одну страницу** (3Kb данных при 4Kb гранулярности страниц памяти операционной системы), что и требуется в подавляющем большинстве случаев, и б). считываемые из /proc данные **не изменяются** при последовательных чтениях до достижения конца файла — вполне достаточно такой **простейшей** реализации, как показана выше (практически ничто из параметров кроме адреса выходных данных не использовано);
- более детально (но тоже достаточно невнятно и путано) о семантике read_proc_t, назначении её параметров, и использовании в редких случаях работы с большими объёмами данных, можно найти в [2];
- для любознательных, мы вернёмся ещё коротко к вариантам реализации read_proc_t далее...
- в литературе иногда утверждается, что для /proc нет API для записи, аналогично API для чтения, но с некоторых пор (позже, чем для чтения) в <linux/proc_fs.h> появилось и такое описание типа (аналогичного типу read_proc_t):

```
typedef int (write_proc_t)( struct file *file, const char __user *buffer,
                            unsigned long count, void *data );
```

- как легко видеть, это определение и по прототипу и логике (обмен с пространством пользователя) заметно отличается от read_proc_t;

Наконец, последним шагом нашей подготовительной работы, мы создадим приложение (тестовое, пользовательского пространства), подобное утилите cat, но которое позволит параметром запуска (число) указать размер объёма данных, запрашиваемых к считыванию за один раз, в процессе последовательного циклического чтения (утилита cat, например, запрашивает 32767 байт за одно чтение read()):

mcat.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include "common.h"

static int get_proc( void ) {
   char dev[ 80 ];
   int df;
   sprintf( dev, "/proc/%s", NAME_NODE );
   if( ( df = open( dev, O_RDONLY ) ) < 0 ) {
      sprintf( dev, "/proc/%s/%s", NAME_DIR, NAME_NODE );
      if( ( df = open( dev, O_RDONLY ) ) < 0 )
         printf( "open device error: %m\n" ), exit( EXIT_FAILURE );
   }
   return df;
}

int main( int argc, char *argv[] ) {
   int df = get_proc(),
       len = ( argc > 1 && atoi( argv[ 1 ] ) > 0 ) ?
             atoi( argv[ 1 ] ) : LEN_MSG;
   char msg[ LEN_MSG + 1 ] = "";
   char *p = msg;
   int res;
   do {
      if( ( res = read( df, p, len ) ) >= 0 ) {
         *( p += res )  = '\0';
         printf( "read + %02d bytes, input buffer: %s", res, msg );
         if( *( p - 1 ) != '\n' ) printf( "\n" );
      }
      else printf( "read device error: %m\n" );
   } while ( res > 0 );
   close( df );
   return EXIT_SUCCESS;
};
```

Теперь всё готово к испытаниям первого из полученных модулей:

```
$ sudo insmod mod_procr.ko
$ ls -l /proc/mod_node
-rw-rw-rw-. 1 root root 0 февр. 13 21:17 /proc/mod_node
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
$ sudo rmmod mod_procr.ko
$ dmesg | tail -n14
[ 1860.021217] ! /proc/mod_node installed
[ 1885.118627] ! read: 3072 (buffer=ec03c000, off=0)
[ 1885.118631] ! return bytes: 61
[ 1885.118636] ! read: 3072 (buffer=ec03c000, off=61)
[ 1885.118639] ! return bytes: 61
[ 1885.118664] ! read: 3072 (buffer=ec03c000, off=61)
[ 1885.118666] ! return bytes: 61
[ 1887.806555] ! read: 3072 (buffer=ec03c000, off=0)
[ 1887.806559] ! return bytes: 61
[ 1887.806564] ! read: 3072 (buffer=ec03c000, off=61)
[ 1887.806567] ! return bytes: 61
[ 1887.806591] ! read: 3072 (buffer=ec03c000, off=61)
[ 1887.806594] ! return bytes: 61
```

```
[ 1925.270224] ! /proc/mod_node removed
```

Хорошо видно откуда возникла цифра 3Kb - 3072. Детально природу того, что происходит «под крышкой», просмотрим созданным тестовым приложением, изменяя размер запрашиваемых данных:

```
$ ./mcat 10
read + 10 bytes, input buffer: .........1
read + 10 bytes, input buffer: .........1.........2
read + 10 bytes, input buffer: .........1.........2.........3
read + 10 bytes, input buffer: .........1.........2.........3.........4
read + 10 bytes, input buffer: .........1.........2.........3.........4.........5
read + 10 bytes, input buffer: .........1.........2.........3.........4.........5.........6
read + 01 bytes, input buffer: .........1.........2.........3.........4.........5.........6
read + 00 bytes, input buffer: .........1.........2.........3.........4.........5.........6
$ dmesg | tail -n20
[ 2136.594396] ! /proc/mod_node installed
[ 2177.736723] ! read: 10 (buffer=cc384000, off=0)
[ 2177.736727] ! return bytes: 61
[ 2177.736786] ! read: 10 (buffer=cc384000, off=10)
[ 2177.736789] ! return bytes: 61
[ 2177.736809] ! read: 10 (buffer=cc384000, off=20)
[ 2177.736811] ! return bytes: 61
[ 2177.736829] ! read: 10 (buffer=cc384000, off=30)
[ 2177.736831] ! return bytes: 61
[ 2177.736848] ! read: 10 (buffer=cc384000, off=40)
[ 2177.736851] ! return bytes: 61
[ 2177.736891] ! read: 10 (buffer=cc384000, off=50)
[ 2177.736894] ! return bytes: 61
[ 2177.736913] ! read: 10 (buffer=cc384000, off=60)
[ 2177.736916] ! return bytes: 61
[ 2177.736918] ! read: 9 (buffer=cc384000, off=61)
[ 2177.736921] ! return bytes: 61
[ 2177.736942] ! read: 10 (buffer=cc384000, off=61)
[ 2177.736944] ! return bytes: 61
[ 2192.404366] ! /proc/mod_node removed
$ ./mcat 10
read + 10 bytes, input buffer: .........1
read + 10 bytes, input buffer: .........1.........2
read + 10 bytes, input buffer: .........1.........2.........3
read + 10 bytes, input buffer: .........1.........2.........3.........4
read + 10 bytes, input buffer: .........1.........2.........3.........4.........5
read + 10 bytes, input buffer: .........1.........2.........3.........4.........5.........6
read + 01 bytes, input buffer: .........1.........2.........3.........4.........5.........6
read + 00 bytes, input buffer: .........1.........2.........3.........4.........5.........6
$ dmesg | tail -n20
[ 2136.594396] ! /proc/mod_node installed
[ 2177.736723] ! read: 10 (buffer=cc384000, off=0)
[ 2177.736727] ! return bytes: 61
[ 2177.736786] ! read: 10 (buffer=cc384000, off=10)
[ 2177.736789] ! return bytes: 61
[ 2177.736809] ! read: 10 (buffer=cc384000, off=20)
[ 2177.736811] ! return bytes: 61
[ 2177.736829] ! read: 10 (buffer=cc384000, off=30)
[ 2177.736831] ! return bytes: 61
[ 2177.736848] ! read: 10 (buffer=cc384000, off=40)
[ 2177.736851] ! return bytes: 61
[ 2177.736891] ! read: 10 (buffer=cc384000, off=50)
[ 2177.736894] ! return bytes: 61
[ 2177.736913] ! read: 10 (buffer=cc384000, off=60)
```

```
[ 2177.736916] ! return bytes: 61
[ 2177.736918] ! read: 9 (buffer=cc384000, off=61)
[ 2177.736921] ! return bytes: 61
[ 2177.736942] ! read: 10 (buffer=cc384000, off=61)
[ 2177.736944] ! return bytes: 61
```

Обращаем **принципиальное** внимание на тот факт, что эти интерфейсные функции не выполняют обмена с задачей пространства пользователя, они только копируют данне между буферами самого ядра, а уже ядро обменивает данные с пространством пользователя (откуда и цифра 3072).

Следующий пример делает регистрацию имени в `/proc` (в точности то же самое, в предыдущем примере), но более простым и более описанным в литературе способом, вызывая `create_proc_read_entry()` (но этот способ просто скрывает суть происходящего):

mod_procr2.c :

```
#include "mod_proc.h"
#include "proc_node_read.c"

static int __init proc_init( void ) {
   if( create_proc_read_entry( NAME_NODE, 0, NULL, proc_node_read, NULL ) == 0 ) {
      ERR( "can't create /proc/%s", NAME_NODE );
      return -ENOENT;
   }
   LOG( "/proc/%s installed", NAME_NODE );
   return 0;
}

static void __exit proc_exit( void ) {
   remove_proc_entry( NAME_NODE, NULL );
   LOG( "/proc/%s removed", NAME_NODE );
}
```

Вот и всё, что нужно вашему модулю для того, чтобы начать отображать себя в `/proc`!

Примечание (важно!): `create_proc_read_entry()` пример того, что API ядра, доступный программисту, **намного** шире, чем список **экспортируемых имён** в `/boot/System.map-*`, и включает ряд вызовов, которые вы не найдёте в `/proc/kallsyms` (как и в случае `create_proc_read_entry()`). Это происходит за счёт достаточно широкого использования `inline` определений (синтаксическое расширение GCC):

```
$ cat /proc/kallsyms | grep create_proc_
c0522237 T create_proc_entry
c0793101 T create_proc_profile
$ cat /proc/kallsyms | grep create_proc_read_entry
$
```

Смотрим файл определений `<linux/proc_fs.h>` :

```
static inline struct proc_dir_entry *create_proc_read_entry(
   const char *name, mode_t mode, struct proc_dir_entry *base,
   read_proc_t *read_proc, void * data ) {
...
}
```

Возвращаемся к испытаниям второго полученного нами минимального модуля:

```
$ sudo insmod mod_procr2.ko
$ echo $?
0
$ ls -l /proc/mod_node
```

```
-r--r--r--. 1 root root 0 февр. 13 21:45 /proc/mod_node
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
$ sudo rmmod mod_procr2.ko
```

И можем повторить всё множество испытаний, показанных ранее для `mod_procr.ko`.

Варианты реализации чтения[13]

Уже было отмечено, что функция чтения `read_proc_t` претерпела в процессе своей эволюции множество изменений, и на сегодня предполагает большое разнообразие её использования. В подтверждение такого многообразия сошлёмся на комментарий в коде **реализации** `read_proc_t` в ядре:

```
/*
 * Prototype:
 * int f(char *buffer, char **start, off_t offset,
 * int count, int *peof, void *dat)
 * Assume that the buffer is "count" bytes in size.
 * If you know you have supplied all the data you
 * have, set *peof.
 *
 * You have three ways to return data:
 * 0) Leave *start = NULL. (This is the default.)
 * Put the data of the requested offset at that
 * offset within the buffer. Return the number (n)
 * of bytes there are from the beginning of the
 * buffer up to the last byte of data. If the
 * number of supplied bytes (= n - offset) is
 * greater than zero and you didn't signal eof
 * and the reader is prepared to take more data
 * you will be called again with the requested
 * offset advanced by the number of bytes
 * absorbed. This interface is useful for files
 * no larger than the buffer.
 * 1) Set *start = an unsigned long value less than
 * the buffer address but greater than zero.
 * Put the data of the requested offset at the
 * beginning of the buffer. Return the number of
 * bytes of data placed there. If this number is
 * greater than zero and you didn't signal eof
 * and the reader is prepared to take more data
 * you will be called again with the requested
 * offset advanced by *start. This interface is
 * useful when you have a large file consisting
 * of a series of blocks which you want to count
 * and return as wholes.
 * (Hack by Paul.Russell@rustcorp.com.au)
 * 2) Set *start = an address within the buffer.
 * Put the data of the requested offset at *start.
 * Return the number of bytes of data placed there.
 * If this number is greater than zero and you
 * didn't signal eof and the reader is prepared to
 * take more data you will be called again with the
 * requested offset advanced by the number of bytes
 * absorbed.
 */
```

Даже в самом комментарии отмечается, что реализация выполнена уже на грани хакинга, она перегружена возможностями, и в литературе многократно отмечается её неадекватное поведение при чтении данных, превышающих объёмом 3072 байт (вариант 1 и 2 комментария). Поэтому рассмотрим только вариант (0 в комментарии) чтения только данных, не превышающих страницу памяти. Здесь тоже могут быть различные реализации (они показаны как варианты в дополнительном архиве `variants.tgz`), далее показаны варианты исполнения. Для каждого сравнительного варианта реализации создаётся свой модуль (вида `mod_proc_*`), который создаёт в `/proc` индивидуальное имя (вида `/proc/mod_node_*`):

```
$ ./insm
```

13 Далее идёт детализация хакерских способов, когда-то применённых в ядре, поэтому этот раздел можно пропустить без ущерба для остального содержания.

```
Module                  Size  Used by
mod_proc_3              1662  0
mod_proc_2              1638  0
mod_proc_1              1642  0
mod_proc_0              1610  0
vfat                    6740  1
/proc/mod_node_0  /proc/mod_node_1  /proc/mod_node_2  /proc/mod_node_3
```

Ниже мы сравним варианты реализации, показаны будут варианты исполнения кода и кратко протокол выполнения (число вызовов и объёмы копируемых данных на каждом вызове), запрос чтения (в цикле по 20 байт) во всех случаях одинаков, осуществляется программой `mcat`, вызов программы показан в листинге для первого варианта:

1. Копирование полного объёма на любой вызов, как обсуждалось ранее (6 операций, 6 копирований по 61 байт на каждой операции):

```
static ssize_t proc_node_read( char *buffer, char **start, off_t off,
                               int count, int *eof, void *data ) {
   unsigned long len = 0;
   LOG( "read: %d (buffer=%p, off=%ld, start=%p)", count, buffer, off, *start );
  // тупо копируем весь буфер, пока return не станет <= off
   memcpy( buffer, buf_msg, length );
   LOG( "copy bytes: %ld", length );
   len = length;
   LOG( "return bytes: %ld%s", len, *eof != 0 ? " ... EOF" : "" );
   return len;
};
$ ./mcat 20 /proc/mod_node_0
...
$ dmesg | tail -n30 | grep !
! read: 20 (buffer=f30ab000, off=0, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f30ab000, off=20, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f30ab000, off=40, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f30ab000, off=60, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 19 (buffer=f30ab000, off=61, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f30ab000, off=61, start=(null))
! copy bytes: 61
! return bytes: 61
```

2. То же самое, но принудительно устанавливается признак конца файла (исчерпания данных), запросов данных становится на единицу меньше (5 операций, 4 копирования по 61 байт на каждой операции):

```
static ssize_t proc_node_read( char *buffer, char **start, off_t off,
                               int count, int *eof, void *data ) {
   unsigned long len = 0;
   LOG( "read: %d (buffer=%p, off=%ld, start=%p)", count, buffer, off, *start );
   // ... на 1 шаг раньше ( return == off ) установлен eof
   if( off < length ) {
      memcpy( buffer, buf_msg, length );
      LOG( "copy bytes: %d", length );
```

```
      len = length;
   }
   *eof = off + count >= length ? 1 : 0;
   LOG( "return bytes: %ld%s", len, *eof != 0 ? " ... EOF" : "" );
   return len;
};
$ dmesg | tail -n14 | grep !
! read: 20 (buffer=f67e8000, off=0, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f67e8000, off=20, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f67e8000, off=40, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f67e8000, off=60, start=(null))
! copy bytes: 61
! return bytes: 61 ... EOF
! read: 20 (buffer=f67e8000, off=61, start=(null))
! return bytes: 0 ... EOF
```

3. Но можно осуществлять только одно полное копирование на самом первом вызове, при всех последующих вызовах будет использоваться промежуточный буфер ядра (5 операций, 1 копирование 61 байт на первой операции). Предполагается, что между вызовами `read()` данные пространства ядра остаются неизменными, но это неявно предполагается и во всех прочих вариантах:

```
static ssize_t proc_node_read( char *buffer, char **start, off_t off,
                               int count, int *eof, void *data ) {
   unsigned long len = 0;
   LOG( "read: %d (buffer=%p, off=%ld, start=%p)", count, buffer, off, *start );
   // 1-но копирование всего буфера на 1-м запросе
   len = length;
   if( 0 == off ) {
      memcpy( buffer, buf_msg, length );
      LOG( "copy bytes: %d", length );
   }
   *eof = off + count >= length ? 1 : 0;
   LOG( "return bytes: %ld%s", len, *eof != 0 ? " ... EOF" : "" );
   return len;
};
$ dmesg | tail -n11 | grep !
! read: 20 (buffer=c4775000, off=0, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=c4775000, off=20, start=(null))
! return bytes: 61
! read: 20 (buffer=c4775000, off=40, start=(null))
! return bytes: 61
! read: 20 (buffer=c4775000, off=60, start=(null))
! return bytes: 61 ... EOF
! read: 20 (buffer=c4775000, off=61, start=(null))
! return bytes: 61 ... EOF
```

4. Копирование ровно запрошенной длины при каждом вызове (5 операций, 4 копирования по 20 байт на каждой операции):

```
static ssize_t proc_node_read( char *buffer, char **start, off_t off,
                               int count, int *eof, void *data ) {
   unsigned long len = 0;
```

```
    LOG( "read: %d (buffer=%p, off=%ld, start=%p)", count, buffer, off, *start );
    // копирование только count байт на каждом запросе - аккуратно меняем return!
    if( off >= length ) {
       *eof = 1;
    }
    else {
       int cp;
       len = length - off;
       cp = min( count, len );
       memcpy( buffer + off, buf_msg + off, cp );
       LOG( "copy bytes: %d", cp );
       len = off + cp;
       *eof = off + cp >= length ? 1 : 0;
    }
    LOG( "return bytes: %ld%s", len, *eof != 0 ? " ... EOF" : "" );
    return len;
};
$ dmesg | tail -n14 | grep !
! read: 20 (buffer=e5633000, off=0, start=(null))
! copy bytes: 20
! return bytes: 20
! read: 20 (buffer=e5633000, off=20, start=(null))
! copy bytes: 20
! return bytes: 40
! read: 20 (buffer=e5633000, off=40, start=(null))
! copy bytes: 20
! return bytes: 60
! read: 20 (buffer=e5633000, off=60, start=(null))
! copy bytes: 1
! return bytes: 61 ... EOF
! read: 20 (buffer=e5633000, off=61, start=(null))
! return bytes: 0 ... EOF
```

Как видно, даже для таких простейших целей у реализатора имеется обширный простор для выбора. Но все варианты отличаются только по показателям производительности. Учитывая, что через имена `/proc` обычно читаются весьма ограниченные объёмы данных, главным образом диагностических, варианты становятся эквивалентными, и предпочтение, возможно, следует отдавать из соображений простоты (простое копирование).

Запись данных

С записью данных всё гораздо проще. Операция записи не имеет возможности позиционирования, и осуществляется по принципу «всё за раз» копированием из пространства пользователя в пространство ядра (тот же дополнительный архив `variants.tgz`):

proc_node_write.c :

```
static int parameter = 999;

static ssize_t proc_node_write( struct file *file, const char __user *buffer,
                                unsigned long count, void *data ) {
   unsigned long len = min( count, size );
   LOG( "write: %ld (buffer=%p)", count, buffer );
   memset( buf_msg, NULL_CHAR, length );        // обнуление прежнего содержимого
   length = len;
   if( copy_from_user( buf_msg, buffer, len ) )
      return -EFAULT;
   sscanf( buf_msg, "%d", &parameter );
   LOG( "parameter set to %d", parameter );
```

```
   return count;
}
```

Общий механизм файловых операций

Другой, альтернативный механизм выполнения обменных операций (чтение, запись) в `/proc`, а в ядре начиная с 3.10 и единственный — это использование традиционного механизм работы с любыми именами файловой системы, основанный на таблице файловых операций (он подробно рассматривался при рассмотрении драйверов символьных устройств). Следующий пример показывает модуль, который создаёт имя в `/proc`, это имя может и читаться и писаться, в этом случае используется структура указателей файловых операций в таблице операций (аналогично тому, как это делалось в драйверах интерфейса `/dev`):

mod_proc.c :

```
#include "mod_proc.h"
#include "fops_rw.c"         // чтение-запись для /proc/mod_node

static const struct file_operations node_fops = {
   .owner = THIS_MODULE,
   .read  = node_read,
   .write  = node_write
};

static int __init proc_init( void ) {
   int ret;
   struct proc_dir_entry *own_proc_node;
#if LINUX_VERSION_CODE < KERNEL_VERSION(3,10,0)
   own_proc_node = create_proc_entry( NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO, NULL );
#else
   own_proc_node = proc_create( NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO, NULL, &node_fops );
#endif
   if( NULL == own_proc_node ) {
      ret = -ENOENT;
      ERR( "can't create /proc/%s\n", NAME_NODE );
      goto err_node;
   }
#if LINUX_VERSION_CODE < KERNEL_VERSION(3,10,0)
   own_proc_node->uid = own_proc_node->gid = 0;
   own_proc_node->proc_fops = &node_fops;
#endif
   LOG( "/proc/%s installed\n", NAME_NODE );
   return 0;
err_node:
   return ret;
}

static void __exit proc_exit( void ) {
   remove_proc_entry( NAME_NODE, NULL );
   LOG( "/proc/%s removed\n", NAME_NODE );
}
```

Здесь всё знакомо, и напоминает предыдущие примеры, за исключением заполнения таблицы файловых операций `node_fops`. Реализующие функции, как и ранее, вынесены в отдельный файл (они используются не один раз):

fops_rw.c :

```
static char *get_rw_buf( void ) {
   static char buf_msg[ LEN_MSG + 1 ] =
          ".........1.........2.........3.........4.........5\n";
```

```
   return buf_msg;
}

// чтение из /proc/mod_proc :
static ssize_t node_read( struct file *file, char *buf,
                          size_t count, loff_t *ppos ) {
   char *buf_msg = get_rw_buf();
   int res;
   LOG( "read: %ld bytes (ppos=%lld)\n", (long)count, *ppos );
   if( *ppos >= strlen( buf_msg ) ) {     // EOF
      *ppos = 0;
      LOG( "EOF" );
      return 0;
   }
   if( count > strlen( buf_msg ) - *ppos )
      count = strlen( buf_msg ) - *ppos;  // это копия
   res = copy_to_user( (void*)buf, buf_msg + *ppos, count );
   *ppos += count;
   LOG( "return %ld bytes\n", (long)count );
   return count;
}

// запись в /proc/mod_proc :
static ssize_t node_write( struct file *file, const char *buf,
                           size_t count, loff_t *ppos ) {
   char *buf_msg = get_rw_buf();
   int res;
   uint len = count < LEN_MSG ? count : LEN_MSG;
   LOG( "write: %ld bytes\n", (long)count );
   res = copy_from_user( buf_msg, (void*)buf, len );
   buf_msg[ len ] = '\0';
   LOG( "put %d bytes\n", len );
   return len;
}
```

В отличие от упрощенных реализаций функций, с которыми мы упражнялись в `/dev`, здесь показана более «зрелая» реализация операции чтения, отслеживающая позиционирование указателя чтения, и допускающая произвольную длину данных в запросе `read()`. Обращаем внимание на то, что функция чтения `node_read()` в этом примере **принципиально** отличается от функции аналогичного назначения `proc_node_read()` в предыдущих примерах: не только своей реализацией, но и прототипом вызова, тем, что она непосредственно работает (копирует) данные в пространство пользователя, и тем, как она возвращает свои результаты.

Повторяем испытательный цикл того, что у нас получилось:

```
$ sudo insmod mod_proc.ko
$ ls -l /proc/mod_*
-rw-rw-rw- 1 root root 0 Июл  2 20:47 /proc/mod_node
$ cat /proc/mod_dir/mod_node
.........1.........2.........3.........4.........5
$ dmesg | tail -n30 | grep -v ^audit
! /proc/mod_node installed
! read: 32768 bytes (ppos=0)
! return 51 bytes
! read: 32768 bytes (ppos=51)
! EOF
$ echo new string > /proc/mod_dir/mod_node
$ cat /proc/mod_dir/mod_node
new string
```

```
$ ./mcat 3
read + 03 bytes, input buffer: new
read + 03 bytes, input buffer: new st
read + 03 bytes, input buffer: new strin
read + 02 bytes, input buffer: new string
read + 00 bytes, input buffer: new string
$ sudo rmmod mod_proc
$ cat /proc/mod_node
cat: /proc/mod_node: Нет такого файла или каталога
```

Начальное содержимое буфера модуля сделано отличающимся от предыдущих случаев как по содержанию, так и по длине (51 байт вместо 61), чтобы визуально легко различать какой из методов работает. Хорошо видно, как изменилась длина запрашиваемых данных утилитой `cat` (32767). Детальный анализ происходящего:

```
$ ./mcat 20
read + 20 bytes, input buffer: .........1.........2
read + 20 bytes, input buffer: .........1.........2.........3.........4
read + 11 bytes, input buffer: .........1.........2.........3.........4.........5
read + 00 bytes, input buffer: .........1.........2.........3.........4.........5
$ dmesg | tail -n30 | grep -v ^audit
! /proc/mod_node installed
! read: 20 bytes (ppos=0)
! return 20 bytes
! read: 20 bytes (ppos=20)
! return 20 bytes
! read: 20 bytes (ppos=40)
! return 11 bytes
! read: 20 bytes (ppos=51)
! EOF
$ ./mcat 80
read + 51 bytes, input buffer: .........1.........2.........3.........4.........5
read + 00 bytes, input buffer: .........1.........2.........3.........4.........5
$ dmesg | tail -n5 | grep -v ^audit
! EOF
! read: 80 bytes (ppos=0)
! return 51 bytes
! read: 80 bytes (ppos=51)
! EOF
$ ./mcat 5
read + 05 bytes, input buffer: .....
read + 05 bytes, input buffer: .........1
read + 05 bytes, input buffer: .........1.....
read + 05 bytes, input buffer: .........1.........2
read + 05 bytes, input buffer: .........1.........2.....
read + 05 bytes, input buffer: .........1.........2.........3
read + 05 bytes, input buffer: .........1.........2.........3.....
read + 05 bytes, input buffer: .........1.........2.........3.........4
read + 05 bytes, input buffer: .........1.........2.........3.........4.....
read + 05 bytes, input buffer: .........1.........2.........3.........4.........5
read + 01 bytes, input buffer: .........1.........2.........3.........4.........5
read + 00 bytes, input buffer: .........1.........2.........3.........4.........5
```

Теперь мы получили возможность не только считывать диагностику со своего модуля, но и передавать ему управляющие воздействия, записывая в модуль новые значения. Ещё раз обратите внимание на размер блока запроса на чтение (в системном журнале), и сравните с предыдущими случаями.

Ну а если нам захочется создать в `/proc` не отдельное имя, а собственную развитую иерархию имён? Как мы наблюдаем это, например, для любого системного каталога:

```
$ tree /proc/driver
/proc/driver
```

```
├── nvram
├── rtc
└── snd-page-alloc
0 directories, 3 files
```

Пожалуйста! Для этого придётся только слегка расширить функцию инициализации предыдущего модуля (ну, и привести ему в соответствие функцию выгрузки). Таким образом, по образу и подобию, вы можете создавать иерархию произвольной сложности и любой глубины вложенности :

mod_proct.c :

```
#include "mod_proc.h"
#include "fops_rw.c" // чтение-запись для /proc/mod_dir/mod_node

static const struct file_operations node_fops = {
   .owner  = THIS_MODULE,
   .read   = node_read,
   .write  = node_write
};

static struct proc_dir_entry *own_proc_dir;

static int __init proc_init( void ) {
   int ret;
   struct proc_dir_entry *own_proc_node;
#if LINUX_VERSION_CODE < KERNEL_VERSION(3,10,0)
   own_proc_dir = create_proc_entry( NAME_DIR, S_IFDIR | S_IRWXUGO, NULL );
   if( NULL == own_proc_dir ) {
      ret = -ENOENT;
      ERR( "can't create directory /proc/%s\n", NAME_DIR );
      goto err_dir;
   }
   own_proc_dir->uid = own_proc_dir->gid = 0;
   own_proc_node = create_proc_entry( NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO, own_proc_dir );
   if( NULL == own_proc_node ) {
      ret = -ENOENT;
      ERR( "can't create node /proc/%s/%s\n", NAME_DIR, NAME_NODE );
      goto err_node;
   }
   own_proc_node->uid = own_proc_node->gid = 0;
   own_proc_node->proc_fops = &node_fops;
#else
   own_proc_dir = proc_mkdir( NAME_DIR, NULL );
   if( NULL == own_proc_dir ) {
      ret = -ENOENT;
      ERR( "can't create directory /proc/%s\n", NAME_NODE );
      goto err_dir;
   }
   own_proc_node = proc_create( NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO, own_proc_dir,
&node_fops );
   if( NULL == own_proc_node ) {
      ret = -ENOENT;
      ERR( "can't create node /proc/%s/%s\n", NAME_DIR, NAME_NODE );
      goto err_node;
   }
#endif
   LOG( "/proc/%s/%s installed\n", NAME_DIR, NAME_NODE );
   return 0;
err_node:
   remove_proc_entry( NAME_DIR, NULL );
```

```
err_dir:
   return ret;
}

static void __exit proc_exit( void ) {
   remove_proc_entry( NAME_NODE, own_proc_dir );
   remove_proc_entry( NAME_DIR, NULL );
   LOG( "/proc/%s/%s removed\n", NAME_DIR, NAME_NODE );
}
```

Здесь обращаем внимание на то, как по разному создаются каталоги и терминальные имена до и после версии ядра 3.9.

Эксперименты с `mod_proct.ko` аналогичны предыдущему, но с отличающимся путевым именем (2-х уровневым), которое создаёт модуль в `/proc`:

```
$ sudo insmod mod_proct.ko
$ tree /proc/mod_dir/
/proc/mod_dir/
└── mod_node
0 directories, 1 file
$ ls -l /proc/mod_dir/mod_node
-rw-rw-rw-. 1 root root 0 февр. 13 22:20 /proc/mod_dir/mod_node
$ cat /proc/mod_dir/mod_node
.........1.........2.........3.........4.........5
$ echo 12345 > /proc/mod_dir/mod_node
$ cat /proc/mod_dir/mod_node
12345
$ echo новая строка > /proc/mod_dir/mod_node
$ cat /proc/mod_dir/mod_node
новая строка
$ sudo rmmod mod_proct.ko
```

Отметим, что таким же способом без всяких изменений, может быть создана иерархия имён **любой глубины вложенности** (это легко видно из кода).

В итоге, было показано, создание структур имен в `/proc` из модулей даёт путь как получения диагностической информации из модуля, так и передачу управляющей информации в модуль. С момента получения таких возможностей модуль становится управляемым (и это зачастую замещает необходимость в операциях `ioctl`, которые очень плохо контролируемы с точки зрения возможных ошибок).

API ядра предоставляет два альтернативных набора функций для реализации ввода вывода.[14] Возникает последний и закономерный вопрос: какая функция из этих двух альтернатив будет отрабатываться, если определены обе? Кто из методов обладает приоритетом? Для ответа был сделан модуль `mod_2.c` (он является линейной комбинацией показанного, и не приводится, но включён в архив примеров к тексту). Вот что показывает его использование:

- определена таблица файловых операций:

```
$ sudo insmod mod_2.ko mode=1
$ cat /proc/mod_node
.........1.........2.........3.........4.........5
$ dmesg | tail -n30 | grep -v ^audit
! /proc/mod_node installed
! read: 32768 bytes
! return 51 bytes
! read: 32768 bytes
```

14 Всё дальнейшее относится только к старому способу чтения-запись, и если вас не интересуют ядра ранее 3.9, может быть опущено.

```
! EOF
$ sudo rmmod mod_2
```

– таблица файловых операций не определялась, чтение функцией чтения `/proc` (отличается строка вывода):

```
$ sudo insmod mod_2.ko mode=2
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
$ dmesg | tail -n30 | grep -v ^audit
! /proc/mod_node installed
! read: 3072 (buffer=f1629000, off=0)
! return bytes: 61  ... EOF
! read: 3072 (buffer=f1629000, off=61)
! return bytes: 0  ... EOF
$ sudo rmmod mod_2
```

– определены **одновременно** и таблица файловых операций и функция чтения `/proc`, операция выполняется через таблицу файловых операций:

```
$ sudo insmod mod_2.ko mode=3
$ cat /proc/mod_node
.........1.........2.........3.........4.........5
$ dmesg | tail -n30 | grep -v ^audit
! /proc/mod_node installed
! read: 32768 bytes
! return 51 bytes
! read: 32768 bytes
! EOF
```

Задачи

1. Напишите модуль, который создаёт иерархию имён в **/proc** как минимум глубиной больше или равной 2 (задавайте путь строкой в параметре загрузки), чтобы в это имя можно было писать целочисленное значение, и считывать его оттуда. Запись **не численного значения** должна возвращать ошибку и не изменять ранее записанное значение.

2. Создайте **модель** системы авторегулирования в /proc:

   - Есть некоторое **регулируемое** значение (это одна переменная, `/proc/xxx/value`, где `xxx` — некоторая промежуточная иерархия имён каталогов, глубиной 1 или более);

   - Есть **корректирующая** переменная (это ещё одна переменная, `/proc/xxx/increment`) — при записи в эту переменную регулируемое значение корректируется: корректирующее значение умножается на коэффициент петлевого усиления (константу) и суммируется с записанным прежде значением `/proc/xxx/value`;

   - Коэффициент петлевого усиления считаем ещё одним именем в `/proc`, по чтению-записи: `/proc/xxx/multiply`;

   - Последовательно считывая (`cat`) `/proc/xxx/value` — формируем значение коррекции как разницу желаемого и имеемого, и записываем (`echo`) в `/proc/xxx/increment`, с целью привести `/proc/xxx/value` к условному значению 0 (оно могло бы быть любым, но для простоты возьмём значение 0).

- Значение `/proc/xxx/value` должно допускать запись для начальной инициализации очередного **возмущения**, коэффициент петлевого усиления задаём как параметр при старте модуля..., но его можно редактировать и записью в `/proc/xxx/multiply`;

- Если в результате записи `/proc/xxx/increment` и реакции модуля на это воздействие регулируемое значение всё ещё не нулевое, то повторяем всю процедуру в цикле...

3. Для предыдущей задачи сделать приложение (пользовательского пространства), которое будет **циклически**, с фиксированным интервалом времени, формировать разницу и записывать корректирующее значение в `/proc/xxx/increment`, до тех пор, пока `/proc/xxx/value` не установится в 0. Приложение должно быть **диалоговым**, позволяющим наблюдать динамику переходных процессов.

   Подсказка: Для упрощения обеих предыдущих задач, контоля и тестирования — сделайте сначала модель циклического авторегулирования в отдельном локальном приложении пользовательского пространства. Наблюдайте и сравнивайте **параллельно** динамику в 2-х приложениях.

4. Проанализируйте, как такие именованные «точки данных» могут использоваться в качестве базового механизма для построения специализированных SCADA системы.

5. (*) Модель работы и API для файловой системы `/proc` (`procfs`) радикально поменялись с ядра 3.10. Сделайте простой модуль (аналогичный примерам в тексте), создающий простейшую иерархию в `/proc` а). для новой модели, б). независимо от версии ядра Linux.

7. Интерфейс /sys

Одно из главных «приобретений» ядра, начинающееся от версий 2.6 — это появление единой унифицированной модели представления **устройств** в Linux. Главные составляющие, сделавшие возможным её существование, это файловая система sysfs и дуальный к ней (поддерживаемый ею) пакет пользовательского пространства udev. Модель устройств— это единый механизм для представления устройств и описания их топологии в системе. Декларируется множество преимуществ, которые обусловлены созданием единого представления устройств:

- Уменьшается дублирование кода.
- Используется механизм для выполнения общих, часто встречающихся функций, таких как счетчики использования.
- Возможность систематизации всех устройств в системе, возможность просмотра состояний устройств и определения, к какой шине то или другое устройство подключено.
- Обеспечивается возможность связывания устройств с их драйверами и наоборот.
- Появляется возможность разделения устройств на категории в соответствии с различными классификациями, таких как устройства ввода, без знания физической топологии устройств.
- Обеспечивается возможность просмотра иерархии устройств от листьев к корню и выключения питания устройств в правильном порядке.

Файловая система sysfs возникла первоначально из нужды поддерживать последовательность действий в динамическом управлении электропитанием (иерархия устройств при включении-выключении) и для поддержки горячего подключения устройств (то есть в обеспечение последнего пункта перечисления). Но позже модель оказалась гораздо плодотворнее. Сама по себе эта система является весьма сложной и объёмной, и о ней одной можно и нужно писать отдельную книгу. Но в контексте нашего рассмотрения нас интересует, в первую голову, возможность **ручного** создания интерфейса из модуля к файловым именам, в файловой системе /sys. Эта возможность весьма напоминает то, как модуль создаёт файловые имена в подсистеме /proc.

Базовым понятием модели представления устройств являются объекты struct kobject (определяется в файле <linux/kobject.h>). Тип struct kobject по смыслу аналогичен абстрактному базовому классу Object в объектно-ориентированных языков программирования, таких как C# и Java. Этот тип определяет общую функциональность, такую как счетчик ссылок, имя, указатель на родительский объект, что позволяет создавать объектную иерархию.

Зачастую объекты struct kobject сами по себе не создаются и не используются, они встраиваются в другие структуры данных, после чего те приобретают свойства, присущие struct kobject, например, такие, как встраиваемость в иерархию объектов. Вот как это выражается в определении уже известной нам структуры представления символьного устройства:

```
struct cdev {
   struct kobject           kobj;
   struct module           *owner;
   struct file_operations  *ops;
   ...
};
```

Во внешнем представлении в каталоге /sys, в интересующем нас смысле, каждому объекту struct kobject соответствует **каталог**, что видно и из самого определения структуры (показано для ядра 3.10):

```
struct kobject {
      const char           *name;
   ...
      struct kobject       *parent;
      struct kobj_type     *ktype;
      struct sysfs_dirent  *sd
   ...
```

```
};
```

Но это вовсе не означает, что каждый инициализированный объект `struct kobject` автоматически экспортируется в файловую систему `/sys`. Для того, чтобы объект сделать видимым в `/sys`, необходимо вызвать:

```
int kobject_add( struct kobject *kobj );
```

Но это не придётся делать явно нам в примерах ниже, просто по той простой причине, что используемые для регистрации имён в `/sys` высокоуровневые вызовы API (`class_create()`) делают это за нас.

Таким образом, объекты `struct kobject` естественным образом отображаются в **каталоги** пространства имён `/sys`, которые увязываются в иерархии. Файловая система `sysfs` это дерево каталогов без файлов. А как создать файлы в этих каталогах, в содержимое которых отображаются данные ядра? Каждый объект `struct kobject` (каталог) содержит (через свой компонент `struct kobj_type`) массив (указателей) структур `struct attribute`:

```
struct kobj_type {
...
   struct sysfs_ops  *sysfs_ops;
   struct attribute **default_attrs;
}
```

А вот каждая такая структура `struct attribute` (определена в `<linux/sysfs.h>`) и является определением одного **файлового имени**, содержащегося в рассматриваемом каталоге:

```
struct attribute {
...
   char   *name        /* имя атрибута-файла */;
   mode_t  mode struct /* права доступа к файлу */;
}
```

Показанный там же **массив структур** таблиц операций (`struct sysfs_ops`, определение также в `<linux/sysfs.h>`) содержит два поля — определения функций `show()` и `store()`, соответственно, чтения и записи символьного поля данных ядра, отображаемых этим файлом:

```
struct sysfs_ops {
   ssize_t (*show)( struct kobject*, struct attribute*, char *buf );
   ssize_t (*store)( struct kobject*, struct attribute*, const char *buf, size_t count );
...
}
```

И сами эти функции и их использование показаны в примере ниже. Их прототипы весьма часто модифицируются от версии к версии ядра (показано для 3.10), поэтому следует обязательно справиться в их текущих определениях, это сэкономит много времени.

Примечание: Показанное поле `sysfs_ops` в составе `struct kobj_type` — это именно **массив структур**, хоть это и не очевидно из описаний. Таким образом, для каждого атрибута (файлового имени) может быть связан **свой** набор функций чтения и записи этого файлового имени извне.

Этих сведений о `sysfs` нам должно быть достаточно для создания интерфейса модуля в пространство имён `/sys`, но перед тем, как переходить к примеру, остановимся в два слова на аналогиях и различиях `/proc` и `/sys` в качестве интерфейса для отображения модулем подконтрольных ему данных ядра. Различия систем `/proc` и `/sys` — складываются главным образом на основе негласных соглашений и устоявшихся традиций:

– информация терминальных имён `/proc` — комплексная, обычно содержит большие объёмы текстовой информации, иногда это таблицы, и даже с заголовками, проясняющими смысл столбцов таблицы;

– информацию терминальных имён `/sys` (атрибутов) рекомендуется оформлять в виде а). простых, б). символьных значений, в). представляющих значения, соответствующие скалярным типам данных языка С (`int, long, char[]`);

Сравним:

```
$ cat /proc/partitions  | head -n5
major minor  #blocks  name
  33     0   10022040 hde
  33     1    3783276 hde1
  33     2          1 hde2
$ cat /sys/devices/audio/dev
14:4
$ cat /sys/bus/serio/devices/serio0/set
2
```

В первом случае это (потенциально) обширная таблица, с сформированным заголовком таблицы, разъясняющим смысл колонок, а во втором — представление целочисленных значений.

А теперь мы готовы перейти к рассмотрению возможного вида модуля (архив `sys.tgz`), читающего и пишущего из/в атрибута-имени, им созданного в `/sys` (большая часть происходящего в этом модуле, за исключения регистрации имён в `/sys` нам уже известно):

xxx.c :

```
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/parport.h>
#include <asm/uaccess.h>
#include <linux/pci.h>
#include <linux/version.h>

#define LEN_MSG 160
static char buf_msg[ LEN_MSG + 1 ] = "Hello from module!\n";

/* <linux/device.h>
LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
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);
};
LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,32)
struct class_attribute {
   struct attribute attr;
   ssize_t (*show)(struct class *class, char *buf);
   ssize_t (*store)(struct class *class, const char *buf, size_t count);
};
*/

/* sysfs show() method. Calls the show() method corresponding to the individual sysfs file */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t x_show( struct class *class, struct class_attribute *attr, char *buf ) {
#else
static ssize_t x_show( struct class *class, char *buf ) {
#endif
   strcpy( buf, buf_msg );
   printk( "read %d\n", strlen( buf ) );
   return strlen( buf );
}

/* sysfs store() method. Calls the store() method corresponding to the individual sysfs file */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t x_store( struct class *class, struct class_attribute *attr,
                        const char *buf, size_t count ) {
```

```
#else
static ssize_t x_store( struct class *class, const char *buf, size_t count ) {
#endif
   printk( "write %d\n" , count );
   strncpy( buf_msg, buf, count );
   buf_msg[ count ] = '\0';
   return count;
}

/* <linux/device.h>
#define CLASS_ATTR(_name, _mode, _show, _store) \
struct class_attribute class_attr_##_name = __ATTR(_name, _mode, _show, _store) */
CLASS_ATTR( xxx, ( S_IWUSR | S_IRUGO ), &x_show, &x_store );

static struct class *x_class;

int __init x_init( void ) {
   int res;
   x_class = class_create( THIS_MODULE, "x-class" );
   if( IS_ERR( x_class ) ) printk( "bad class create\n" );
   res = class_create_file( x_class, &class_attr_xxx );
/* <linux/device.h>
extern int __must_check class_create_file(struct class *class, const struct class_attribute
*attr); */
   printk( "'xxx' module initialized\n" );
   return 0;
}

void x_cleanup( void ) {
/* <linux/device.h>
extern void class_remove_file(struct class *class, const struct class_attribute *attr); */
   class_remove_file( x_class, &class_attr_xxx );
   class_destroy( x_class );
   return;
}

module_init( x_init );
module_exit( x_cleanup );
MODULE_LICENSE( "GPL" );
```

В коде показанного модуля:

- Создаётся класс `struct class` (соответствующая ему структура `struct kobject`), отображаемый вызовом `class_create()` в создаваемый **каталог** с именем `"x-class"`;

- Создаётся атрибутная запись `struct class_attribute`, с именем этой **переменной** `class_attr_xxx`. Эта переменная создаётся макросом `CLASS_ATTR()`, поэтому имя переменной образуется (текстовой конкатенацией) из параметра макровызова. Это может казаться необычным без привычки, но всё, что делается вокруг `/sys` — выполняется в подобной технике: этот и ряд подобных макросов.

- Этот же макровызов определяет и имена (адреса) функций обработчиков `show()` и `store()` для этого атрибута.

- Создаётся атрибутная запись увязывается с ранее созданным классом вызовом `class_create_file()`. Именно на этом этапе будет создано **файловое** имя в каталоге `"x-class"`. Само имя, под которым будет представляться файл, определилось на предыдущем шаге в вызове `CLASS_ATTR()`.

– Функция обработчик `show()` будет вызывается при выполнении запросов на чтение файлового имени `/sys/class/x-class/xxx`, а `store()`, соответственно, на запись.

– Обычной практикой является запись целой последовательности макровызовов `CLASS_ATTR()`, которые определят несколько атрибутных записей, которые позже последовательностью вызовов `class_create_file()` увяжутся с единым классом (создаётся целая группа файлов в одном каталоге).

– При таком групповом создании **имена** обрабатывающих функций (3-й и 4-й параметры макровызовов `CLASS_ATTR()`) также формируются как конкатенация параметров вызова. Другой практикой уточнения к какому атрибуту относится запрос, является динамический анализ параметра `attr`, полученного функциями `show()` и `store()` (только в относительно поздних версиях ядра, после 2.6.32).

Особенностями кода, работающего с подсистемой `/sys`, является высокая волатильность всего API, доступного для использования в коде: прототипы функций, структуры данных, макроопределения и всё прочее. Это подчёркнуто тем, что комментарии относительно различий версий включены в текст примера выше, чтобы подчеркнуть наиболее «версиеопасные направления» (определения взяты из хэдер-файлов).

Теперь мы готовы рассмотреть работу кода:

```
$ sudo insmod xxx.ko
$ lsmod | head -n2
Module                  Size  Used by
xxx                     1047  0
$ ls -lR /sys/class/x-class
/sys/class/x-class:
итого 0
-rw-r--r--. 1 root root 4096 Янв 27 23:34 xxx
$ tree /sys/class/x-class
/sys/class/x-class
└── xxx
0 directories, 1 file
$ ls -l /sys/module/xxx/
итого 0
drwxr-xr-x 2 root root    0 Янв 27 23:57 holders
-r--r--r-- 1 root root 4096 Янв 27 23:57 initstate
drwxr-xr-x 2 root root    0 Янв 27 23:57 notes
-r--r--r-- 1 root root 4096 Янв 27 23:57 refcnt
drwxr-xr-x 2 root root    0 Янв 27 23:57 sections
-r--r--r-- 1 root root 4096 Янв 27 23:57 srcversion
$ dmesg | tail -n18 | grep -v ^audit
'xxx' module initialized
 $ cat /sys/class/x-class/xxx
Hello from module!
$ dmesg | tail -n15 | grep -v ^audit
read 19
```

К этому месту мы убеждаемся (по форме вывода), что операция чтения со стороны пользователя действительно выполняется функцией `show()`, и названия функций `show()` и `store()` отражают направления передачи данных именно **со стороны внешнего наблюдателя** (из пространства пользователя). Для разработчика кода модуля они носят в точности противоположный смысл. Смотрит дальнейшие операции:

```
$ echo 12345 >  /sys/class/x-class/xxx
bash: /sys/class/x-class/xxx: Отказано в доступе
$ sudo echo 12345 >  /sys/class/x-class/xxx
bash: /sys/class/x-class/xxx: Отказано в доступе
```

Этот номер не проходит — из за прав досту аимени

Этот номер не проходит — из-за прав доступа имени `xxx`, и в версиях ядра начиная с 3.15 параметр `mode` (2-й) макроса `CLASS_ATTR()` контролируется жёстко. Но запись можно осуществить (проверить) либо с терминала с регистрацией `root`, либо вот таким трюком:

```
$ echo 12345 | sudo tee /sys/class/x-class/xxx
12345
$ cat /sys/class/x-class/xxx
12345
$ dmesg | tail -n6
[ 3793.398782] perf interrupt took too long (2506 > 2500), lowering
kernel.perf_event_max_sample_rate to 50000
[39631.990792] xxx: module verification failed: signature and/or  required key missing - tainting
kernel
[39631.990997] 'xxx' module initialized
[39694.685078] read 19
[39975.563266] write 6
[39983.048538] read 6
$ sudo rmmod xxx.ko
$ cat /sys/class/x-class/xxx
cat: /sys/class/x-class/xxx: Нет такого файла или каталога
```

В тех случаях (а это как правило), когда стремятся создать целую группу файловых имён, связанных с модулем, поведение которых обычно сходно и отличается лишь деталями (например, привязкой к **различным** переменным внутри модуля), обычно используют параметризуемые макроопределения для определения функций `show()` и `store()`. Читать и отлаживать это практически невозможно, но это работает и массово используется. Для конкретизации сказанного перепишем предыдущий модуль так, чтобы он создавал три независимых точки входа (показаны только отличия от `xxx.c`):

xxm.c :

```
...
#define LEN_MSG 160

// определения функций обработчиков
#define IOFUNCS( name )                                                    \
static char buf_##name[ LEN_MSG + 1 ] = "не инициализировано "#name"\n";   \
static ssize_t SHOW_##name( struct class *class, struct class_attribute *attr,  \
                            char *buf ) {                                  \
   strcpy( buf, buf_##name );                                              \
   printk( "read %d\n", strlen( buf ) );                                   \
   return strlen( buf );                                                   \
}                                                                          \
static ssize_t STORE_##name( struct class *class, struct class_attribute *attr, \
                             const char *buf, size_t count );              \
   printk( "write %d\n", count );                                          \
   strncpy( buf_##name, buf, count );                                      \
   buf_##name[ count ] = '\0';                                             \
   return count;                                                           \
}
IOFUNCS( data1 );
IOFUNCS( data2 );
IOFUNCS( data3 );

// определение атрибутных записей
#define OWN_CLASS_ATTR( name ) \
struct class_attribute class_attr_##name = \
__ATTR( name, 0666, &SHOW_##name, &STORE_##name )
static OWN_CLASS_ATTR( data1 ); // создаётся class_attr_data1
static OWN_CLASS_ATTR( data2 ); // создаётся class_attr_data2
static OWN_CLASS_ATTR( data3 ); // создаётся class_attr_data3
...

int __init x_init(void) {
```

```
   ...
   res = class_create_file( x_class, &class_attr_data1 );
   res = class_create_file( x_class, &class_attr_data2 );
   res = class_create_file( x_class, &class_attr_data3 );
   ...
}

void x_cleanup(void) {
   class_remove_file( x_class, &class_attr_data1 );
   class_remove_file( x_class, &class_attr_data2 );
   class_remove_file( x_class, &class_attr_data3 );
   ...
}
```

В подобных конструкциях могут создаваться весьма обширные коллекции файловых имён, в обсуждаемом примере их три:

```
$ sudo insmod xxm.ko
$ tree /sys/class/x-class
/sys/class/x-class
├── data1
├── data2
└── data3
0 directories, 3 files
$ cat /sys/class/x-class/data*
не инициализировано data1
не инициализировано data2
не инициализировано data3
$ echo строка 1 | sudo tee 1>/dev/null /sys/class/x-class/data1
$ echo строка 2 | sudo tee 1>/dev/null /sys/class/x-class/data2
$ echo строка 3 | sudo tee 1>/dev/null /sys/class/x-class/data3
$ cat /sys/class/x-class/data*
строка 1
строка 2
строка 3
$ sudo rmmod xxm.ko
```

Часто задаваемый вопрос: а можно ли в `/sys` создать **иерархию** имён произвольной глубины и структуризации ... так как это возможно сделать в `/proc`? В первом приближении ответ: **нет**. Потому что `/sys` — это каталог управления и отображения **подсистемы ввода-вывода**, устройств, со своей структурой, а `/sys/class`, в котором мы создаём имена для связи с модулем — это каталог **классов** устройств.

Но если это окажется уж особо необходимым, мы можем уже внутри собственного класса устройств создать фиктивное устройство (каталог), по типу (в продолжение предыдущих кодов):

```
ssize_t device_show_ulong( struct device *dev, struct device_attribute *attr, char *buf) {
... };
ssize_t device_store_ulong( struct device *dev, struct device_attribute *attr,
                            const char *buf, size_t count ) {
...};
// #define DEVICE_ATTR(_name, _mode, _show, _store) \
//         struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
static OWN_DEVICE_ATTR( data, 0666, &device_show_ulong, &device_store_ulong );
   ...
   x_class = class_create( THIS_MODULE, "x-class" );
   dev_t dev = MKDEV( 10, 1 ); // произвольно, для примера
   struct device *pdev = device_create( x_class, NULL, dev, NULL, "zzz" );
   int res = device_create_file( pdev, &dev_attr_data );
   ...
```

(Все названные вызовы описаны в <linux/device.h>, как легко заметить, все они подобны обсуждавшимся выше с заменой префиксов class_* на device_*). При этом будет создано фиктивное устройство:

```
$ ls -l /dev/zzz
crw------- 1 root root 10, 1 май 16 20:18 /dev/zzz
```

А в /sys/class будет создан подкаталог (ссылка) x_class/zzz, с характерной структурой устройства, но с дополнительным терминальным именем data. Но, конечно, подобная схема очень искусственна.

Ошибки обменных операций

А что, если по какой-то причине **созданное** путевое имя в /sys не может быть сосчитано, если пользователю нужно сообщить о ошибке операции ввода вывода? Для этого в собственных функциях чтения-записи возвращаем **отрицательный** код ошибки (показаны только отличия от примера xxx.c).

xxe.c :

```
...
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t x_show( struct class *class, struct class_attribute *attr, char *buf ) {
#else
static ssize_t x_show( struct class *class, char *buf ) {
#endif
   ...
   return -EIO;
}

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t x_store( struct class *class, struct class_attribute *attr,
                        const char *buf, size_t count ) {
#else
static ssize_t x_store( struct class *class, const char *buf, size_t count ) {
#endif
   ...
   return -EIO;
}
...
```

Уточнённые символьные коды ошибок черпаем из заголовочных файлах: errno-base.h и errno.h каталога <uapi/asm-generic> (и ещё несколько с более специфичными кодами):

```
#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Argument list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
...
```

А вот как выглядит выполнение с ошибками с точки зрения пользователя:

```
$ sudo insmod xxe.ko
$ lsmod | head -n3
Module                  Size  Used by
xxe                    12626  0
fuse                   91410  3
$ ls -l /sys/class/x-class/xxx
```

```
-rw-r--r--. 1 root root 4096 фев  9 21:23 /sys/class/x-class/xxx
$ cat /sys/class/x-class/xxx
cat: /sys/class/x-class/xxx: Ошибка ввода/вывода
$ echo 11111111 | sudo tee 1>/dev/null /sys/class/x-class/xxx
tee: /sys/class/x-class/xxx: Ошибка ввода/вывода
$ sudo rmmod xxe.ko
```

На этом мы и остановимся в рассмотрении подсистемы `/sys`. Потому, как сейчас функции `/sys` в Linux расширились настолько, что об этой файловой подсистеме одной можно и нужно писать отдельную книгу: все устройства в системе (сознательно стараниями автора модуля для него, или даже помимо его воли) — находят отображения в `/sys`, а сопутствующая ей подсистема пользовательского пространства `udev` динамически управляет правилами создания имён и полномочия доступа к ним. Но это — совершенно другая история. Мы же в кратком примере рассмотрели совершенно частную задачу: как из собственного модуля создать интерфейс к именам в `/sys`, для создания диагностических или управляющих интерфейсов этого модуля.

Задачи

1. Проделайте в `/sys` аналогичные действия как для /proc (создать иерархию имени по записи-чтению).

   Подсказка: Для этой задачи используйте `struct class` и макрос `CLASS_ATTR()` (это тот способ, который чаще показывается в учебниках). Создавайте собственные каталоги и терминальные имена в `/sys/class/...`

2. Проделайте то же самое, что в предыдущей задаче, но теперь используя непосредственно `struct kobject`, а также работу с группой атрибутов (файловых имён) `struct attribute_group`. Для большей общности сразу предусмотрите возможность создания нескольких идентичных по назначению имён в `/sys/kernel/...`. Как сделать это число имён наиболее динамичный (наиболее безболезненно изменяемым). Сравните сложность этого и предыдущего способов.

3. Создайте модель системы авторегулирования, так как это делалось для `/proc`, но используя в качестве базового каталога «точек данных» `/sys/class/*` или `/sys/kernel/*`: там, где вы чувствуете себя увереннее.

8. Сетевой стек

Сетевая подсистема является гораздо более разветвлённая чем интерфейс устройств Linux. Но, несмотря на обилие возможностей (например, если судить по числу обслуживающих сетевых утили т: `ifconfig`, `ip`, `netstat`, `route` ... и до нескольких десятков иных) — сетевая подсистема Linux, с позиции разработчика ядра, логичнее и прозрачнее, чем, например, тот же интерфейс устройств.

Существует ещё один дополнительный мотив, согласно которому сетевая подсистема должна быть для программиста разработчика особенно близка и интересна: в связи с взрывным расширением спектра протоколов и устройств коммуникаций, разработки драйверов специфических (проприетарных) коммуникационных устройств выпадают в задачи гораздо чаще, чем для любых других видов устройств. И предоставляют расширенные возможности для творчества...

Сетевая подсистема Linux ориентирована в большей степени на обслуживание протоколов Ethernet на канальном уровне и TCP/IP на уровне транспортном, но эта модель расширяется с равным успехом и на другие типы протоколов, таким образом покрывая весь спектр возможностей. На сегодня сетевая подсистема Linux обеспечивает поддержку широчайшего спектра протоколов и устройств:

- проводные соединения Ethernet (LAN);
- беспроводные соединения WiFi (LAN) и Bluetooth (PAN);
- разнообразные проводные устройства «последней мили» (WAN): E1/T1/J1, различные модификации DSL, ...
- беспроводные модемы (WAN), относящиеся к разнообразным протоколам, сетям и модификациям: GSM, GPRS, EDGE, CDMA, EV-DO (EVDO), WiMAX, LTE, ...

Весь этот спектр (и ещё некоторые классы менее употребляемых технологий) поддерживается **единой** сетевой подсистемой.

Сеть TCP/IP, как известно, очень условно согласуется (или вовсе не согласуется) с 7-ми уровневой моделью OSI взаимодействия открытых систем (она и разработана раньше модели OSI, и, естественно, они не соответствуют друг другу). В Linux сложилась такая терминология разделения на подуровни, которая соответствует [24]:

- всё, что относится к поддержке оборудования и канальному уровню — описывается как сетевые **интерфейсы**, и обозначается как L2, преимущественно это Ethernet, но и другие сетевые протоколы канального уровня (Token Ring, ArcNet, ...) тоже;
- протоколы сетевого уровня OSI (IP/IPv4/IPv6 , IPX, RIP, OSPF, ARP, ...) — как **сетевой** уровень стека протоколов, или уровень L3;
- всё, что выше (ICMP, UDP, TCP, SCTP ...) - как протоколы **транспортного** уровня, или уровень L4;
- всё же то, что относится к ещё выше лежащим уровням (сеансовый, представительский, прикладной) модели OSI (например: SSH, SIP, RTP, ...) — никаким образом не проявляется в ядре, и относится уже только к области клиентских и серверных утилит пространства пользователя.

Такая сложившаяся числовая нумерация сетевых слоёв (layers: L2, L3, L4) соответствует **аналогиям** (не более) из модели OSI (обратите внимание, что в принятой в Linux терминологии нет слоя L1 — это физический уровень передачи данных, который не попадает в круг интересов разработчиков системы). Точно таким же образом, в круг интересов разработчиков **ядра** Linux не попадают все сетевые слои, которые лежат выше L4 — это уже протоколы и приложения пользовательского уровня (абстракция сетевых сокетов), но именно там создаются и потребляются сетевые пакеты.

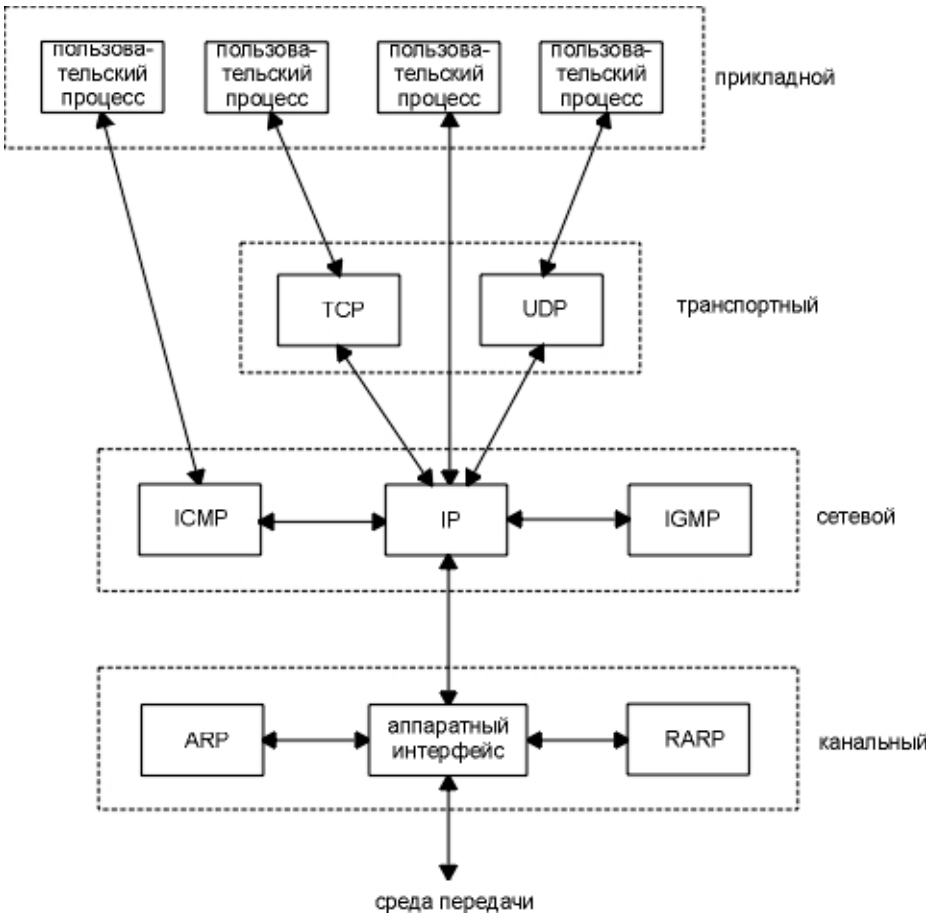


Рис. : Сетевой стек Linux (заимствовано из легендарной книги У. Р. Стивенса «TCP/IP Illustrated»).

Сетевые инструменты

Но прежде, чем заняться созданием модулей сетевых интерфейсов, нам придётся коротко восстановить в памяти те инструменты, которыми мы будем наблюдать и управлять сетевыми состояниями в системе.

Сетевые интерфейсы

В отличие от всех прочих **устройств** в системе (символьных или блочных), которым соответствуют имена устройств в каталоге `/dev`, сетевые устройства создают сетевые **интерфейсы**, которые не отображаются как именованные устройства, но каждый из которых имеет набор своих характеристических параметров (MAC адрес, IP адрес, маска сети, ...). Интерфейсы могут быть физическими (отображающими реальные аппаратные сетевые устройства, например, `eth0` — адаптер Ethernet), или логическими, виртуальными (отражающими некоторые моделируемые понятия, например, `tap0` — туннельный интерфейс).

Имена сетевых интерфейсов, как мы увидим вскоре, могут быть **произвольными** и определяются **модулем** ядра, реализующим интерфейс для устройств такого типа. В системе не может быть интерфейсов с совпадающими именами, поэтому поддерживающий модуль будет как-то модифицировать имена интерфейсов однотипных устройств, в соответствии с принятой схемой именований.

Примечание: Существует предрассудок, что имена проводных устройств Ethernet — это, например, `eth0`, `eth1`, `eth2`, ..., а WiFi интерфейсы — это `wlan0`, `wlan1`, `wlan2`, ... соответственно, и так далее. Это не так. Когда то это соответствовало

действительности, но в современном Linux имена интерфейсов могут быть **произвольными**. Более того, на одной и той же аппаратной конфигурации, при установке различных версий даже одного и того же дистрибутива, имена интерфейсов и схема именования могут различаться.

Посмотреть текущие существующие сетевые интерфейсы можно, например, так:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp2s14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group
default qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wlp8s0: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN mode DORMANT group default qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
```

Этим же интерфейсам соответствуют подкаталоги с соответствующими именами в `/proc`, каждый такой каталог содержит псевдофайлы-параметры (по диагностике или управлению) соответствующего интерфейса:

```
$ ls /proc/sys/net/ipv4/conf
all  default  enp2s14  lo  wlp8s0
$ ls -w80 /proc/sys/net/ipv4/conf/enp2s14/
accept_local       disable_xfrm                    proxy_arp_pvlan
accept_redirects   force_igmp_version              route_localnet
accept_source_route forwarding                     rp_filter
arp_accept         igmpv2_unsolicited_report_interval  secure_redirects
arp_announce       igmpv3_unsolicited_report_interval  send_redirects
arp_filter         log_martians                    shared_media
arp_ignore         mc_forwarding                   src_valid_mark
arp_notify         medium_id                       tag
bootp_relay        promote_secondaries
disable_policy     proxy_arp
```

Не менее важной, чем набор сетевых интерфейсов, характеристикой сетевой подсистемы является таблица роутинга ядра, которая полностью и однозначно определяет направления (между интерфейсами) распространения трафика, например:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: em1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group
default qlen 1000
    link/ether a0:1d:48:f4:93:5c brd ff:ff:ff:ff:ff:ff
3: wlo1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DORMANT group
default qlen 1000
    link/ether 34:23:87:d6:85:0d brd ff:ff:ff:ff:ff:ff
4: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode
DEFAULT group default
    link/ppp
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    1024   0        0 em1
80.255.73.34    0.0.0.0         255.255.255.255 UH    0      0        0 ppp0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 em1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 wlo1
```

Несоответствие таблицы роутинга состояниям сетевых интерфейсов (что весьма часто случается при экспериментах и отладке сетевых модулей ядра) — наиболее частая причина отличия поведения сети от ожидаемого (картина восстанавливается соответствующими командами `route`, добавляющими или удаляющими направления в таблицу). Самое краткое и **исчерпывающее** описание работы TCP/IP сети (из известных автору) дал У. Р. Стивенс:

1. IP-пакеты (создающиеся на хосте или приходящие на него снаружи), если они не предназначены

данному хосту, ретранслируются в соответствии с одной из строки таблицы роутинга на основе IP адреса **получателя**.

2. Если ни одна строка таблицы не соответствует адресу получателя (подсеть или хост), то пакеты ретранслируются в интерфейс, который обозначен как интерфейс по умолчанию, который **всегда** присутствует в таблице роутинга (интерфейс с Destination равным 0.0.0.0 в примере показанном выше).

3. Пакет, пришедший с некоторого интерфейса, **никогда** не ретранслируется в этот же интерфейс.

По этому алгоритму всегда можно разобрать картину происходящего в системе с любой самой сложной конфигурацией интерфейсов.

Одному аппаратному сетевому устройству (физическому интерфейсу) может соответствовать один (наиболее частый случай) или несколько различных сетевых интерфейсов. Новые (логические) сетевые интерфейсы могут **надстраиваться** (соответствующими модулями ядра) над уже существующими (физическими или виртуальными). По тексту далее, мы станем называть такие надстроенные интерфейсы виртуальными — построенными над другими ранее существующими. Задача надстройки виртуальных сетевых интерфейсов зачастую и является самой распространённой задачей, стоящей перед практическим разработчиком. Простейшим примером множественности логических сетевых интерфейсов могут служить **сетевые алиасы**, когда существующему интерфейсу дополнительно присваиваются адрес-маска, делающие его представленным ещё в одной подсети:

```
$ ifconfig
enp2s14: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.5  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::215:60ff:fec4:ee02  prefixlen 64  scopeid 0x20<link>
        ether 00:15:60:c4:ee:02  txqueuelen 1000  (Ethernet)
        RX packets 16943  bytes 22978143 (21.9 MiB)
        RX errors 0  dropped 1  overruns 0  frame 0
        TX packets 10437  bytes 851740 (831.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 16
...
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    1024   0        0 enp2s14
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 enp2s14
```

К этому моменту у нас присутствует единственный Ethernet интерфейс `enp2s14`, в LAN 192.168.1.0/24 (он же интерфейс по умолчанию).

```
$ sudo ifconfig enp2s14:1 192.168.5.5
$ ifconfig
enp2s14: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.5  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::215:60ff:fec4:ee02  prefixlen 64  scopeid 0x20<link>
        ether 00:15:60:c4:ee:02  txqueuelen 1000  (Ethernet)
        RX packets 17105  bytes 22992037 (21.9 MiB)
        RX errors 0  dropped 1  overruns 0  frame 0
        TX packets 10544  bytes 864868 (844.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 16

enp2s14:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.5.5  netmask 255.255.255.0  broadcast 192.168.5.255
        ether 00:15:60:c4:ee:02  txqueuelen 1000  (Ethernet)
        device interrupt 16
...
```

Теперь в системе создан (таким простым способом) дополнительный **алиасный** сетевой интерфейс (синоним) в подсеть 192.168.5.0/24, и трафик с хостами этой подсети будет направляться через интерфейс `enp2s14:1` (хотя физически будет всё так же проходить через `enp2s14`):

```
$ ip addr
...
2: enp2s14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.5/24 brd 192.168.1.255 scope global enp2s14
       valid_lft forever preferred_lft forever
    inet 192.168.5.5/24 brd 192.168.5.255 scope global enp2s14:1
       valid_lft forever preferred_lft forever
    inet6 fe80::215:60ff:fec4:ee02/64 scope link
       valid_lft forever preferred_lft forever
...
$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         192.168.1.1     0.0.0.0         UG    1024   0        0 enp2s14
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 enp2s14
192.168.5.0     0.0.0.0         255.255.255.0   U     0      0        0 enp2s14
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    1024   0        0 enp2s14
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 enp2s14
192.168.5.0     0.0.0.0         255.255.255.0   U     0      0        0 enp2s14
$ ping 192.168.5.5
PING 192.168.5.5 (192.168.5.5) 56(84) bytes of data.
64 bytes from 192.168.5.5: icmp_seq=1 ttl=64 time=0.526 ms
64 bytes from 192.168.5.5: icmp_seq=2 ttl=64 time=0.323 ms
^C
--- 192.168.5.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.323/0.424/0.526/0.103 ms
```

Техника создания и работы с сетевыми алиасами является в высшей степени полезной при отработке кодов модулей ядра, обсуждаемых далее.

Создание **виртуальных** сетевых интерфейсов, обладающих некоторыми дополнительными качествами относительно базовых интерфейсов, над которыми они надстроены (например, шифрование трафика по определённому алгоритму), и является главным предметом следующего ниже обзора реализующих модулей ядра.

Инструменты наблюдения

При рассмотрении драйверов блочных устройств ранее, должно было броситься в глаза то обстоятельство, что эти драйвера не столько отличаются от драйверов других классов устройств (символьных, например), сколько тем, что они потребовали для своего испытания, тестирования и отладки привлечения совершенно особого и широкого спектра специфических программного инструментария: `fdisk`, `gparted`, `gdisk`, `mkfs.*`, `mount` и других. Ещё более это выявляется при работе с сетевыми интерфейсами и протоколами — успех здесь определяется тем, каким арсеналом инструментов мы обладаем и как гибко можем его использовать. Объясняется это тем, что здесь мы работаем с **совершенно иным** инструментарием, неприменимым к устройствам, и наоборот — инструментарий для работы с устройствами неприменим к сетевой сфере.

И, поскольку представление сетевых интерфейсов принципиально отличается от устройств, то при отработке модулей ядра поддержки сетевых средств используется совершенно особое множество **команд-утилит**. Их мы используем для контроля, диагностики и управления сетевыми интерфейсами. Набор сетевых утилит, используемых в сетевой разработке — огромен! Но ниже мы только назовём некоторые из них, без которых такая работа просто невозможна...

Простейшими инструментами **диагностики** в нашей отработке примеров будет посылка «пульсов» — тестирующих ICMP пакетов `ping` (был показан ранее) и `traceroute` (задержка прохождения промежуточных хостов на трассе маршрута):

```
$ traceroute 80.255.64.23
traceroute to 80.255.64.23 (80.255.64.23), 30 hops max, 60 byte packets
 1  192.168.1.1 (192.168.1.1)  1.052 ms  1.447 ms  1.952 ms
 2  * * *
 3  10.50.21.14 (10.50.21.14)  32.584 ms  34.609 ms  34.828 ms
 4  umc-10G-gw.ix.net.ua (195.35.65.50)  37.521 ms  38.751 ms  39.052 ms
 5  * * *
```

Команды `ping` и `traceroute` имеют множество опций (показываемых по `--help` и описанных в `man`), например, из числа самых необходимых при отработке сетевых модулей, возможность направить поток ICMP в указанный сетевой интерфейс из числа нескольких наличествующих:

```
$ ping -I wlo1 192.168.1.1
PING 192.168.1.1 (192.168.1.1) from 192.168.1.21 wlo1: 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=2.12 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=1.99 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=64 time=2.00 ms
...
```

Самым известным из инструментов **управления** сетевыми интерфейсами является утилита:

```
$ ifconfig
...
cipsec0   Link encap:Ethernet  HWaddr 00:0B:FC:F8:01:8F
          inet addr:192.168.27.101  Mask:255.255.255.0
          inet6 addr: fe80::20b:fcff:fef8:18f/64 Scope:Link
          UP RUNNING NOARP  MTU:1356  Metric:1
          RX packets:4 errors:0 dropped:3 overruns:0 frame:0
          TX packets:18 errors:0 dropped:5 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:538 (538.0 b)  TX bytes:1670 (1.6 KiB)
...
wlan0     Link encap:Ethernet  HWaddr 00:13:02:69:70:9B
          inet addr:192.168.1.21  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::213:2ff:fe69:709b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10863 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11768 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3274108 (3.1 MiB)  TX bytes:1727121 (1.6 MiB)
```

Здесь показаны два сетевых интерфейса: физическая беспроводная сеть Wi-Fi (`wlan0`) и виртуальный интерфейс (виртуальная частная сеть, VPN) созданный программными средствами (Cisco Systems VPN Client) от Cisco Systems (`cipsec0`), работающий через один и тот же (`wlan0` в показанном случае) физический канал (что подтверждает сказанное выше о возможности нескольких сетевых интерфейсов над одним физическом каналом). Для управления создаваемым сетевым интерфейсом (например, операции `up` или `down`), в отличие от диагностики, утилита `ifconfig` потребует прав `root`.

Примечание: Интересно, что если тот же VPN-канал создать «родными» Linux средствами OpenVPN (вместо Cisco Systems VPN Client) к тому же удалённому серверу-хосту, то мы получим совершенно другой (и даже, если нужно, ещё один **дополнительно**, параллельно) сетевой интерфейс — различия в интерфейсах обусловлены **модулями ядра**, которые их создавали:

```
$ ifconfig
...
tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          inet addr:192.168.27.112  P-t-P:192.168.27.112  Mask:255.255.255.0
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1412  Metric:1
```

```
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:1905 (1.8 KiB)  TX bytes:883 (883.0 b)
```

Несколько менее известным[15], но более развитым инструментом **управления**, является более поздняя утилита `ip` (в некоторых дистрибутивах может потребоваться отдельная установка как пакета, известного под именем `iproute2`), вот результаты выполнения для той же конфигурации:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
4: vboxnet0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether 0a:00:27:00:00:00 brd ff:ff:ff:ff:ff:ff
5: pan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
    link/ether ae:4c:18:a0:26:1b brd ff:ff:ff:ff:ff:ff
6: cipsec0: <NOARP,UP,LOWER_UP> mtu 1356 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:0b:fc:f8:01:8f brd ff:ff:ff:ff:ff:ff
$ ip addr show dev cipsec0
6: cipsec0: <NOARP,UP,LOWER_UP> mtu 1356 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:0b:fc:f8:01:8f brd ff:ff:ff:ff:ff:ff
    inet 192.168.27.101/24 brd 192.168.27.255 scope global cipsec0
    inet6 fe80::20b:fcff:fef8:18f/64 scope link
       valid_lft forever preferred_lft forever
```

Утилита `ip` имеет очень разветвлённый синтаксис (и возможности), но, к счастью, и такую же разветвлённую (древовидную) систему подсказок:

```
$ ip help
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
       ip [ -force ] -batch filename
where  OBJECT := { link | addr | addrlabel | route | rule | neigh | ntable |
                   tunnel | maddr | mroute | monitor | xfrm }
       OPTIONS := { -V[ersion] | -s[tatistics] | -d[etails] | -r[esolve] |
                    -f[amily] { inet | inet6 | ipx | dnet | link } |
                    -o[neline] | -t[imestamp] | -b[atch] [filename] }
$ ip addr help
Usage: ip addr {add|change|replace} IFADDR dev STRING [ LIFETIME ]
                                                      [ CONFFLAG-LIST]
       ip addr del IFADDR dev STRING
       ip addr {show|flush} [ dev STRING ] [ scope SCOPE-ID ]
                            [ to PREFIX ] [ FLAG-LIST ] [ label PATTERN ]
...
```

Этими утилитами работы с сетевыми интерфейсами мы будем пользоваться при отработке модулей ядра, создающих такие интерфейсы.

Ещё одним инструментом **управления** работой сетевой системы, не пересекающимся с перечисленными, является утилита `route` — управление таблицей роутинга ядра (и диагностика содержимого). Простейший пример использования утилиты `route` для изменений в таблице роутинга будет показан ниже, в примере переименований сетевого интерфейса.

Ещё один инструмент, который непременно понадобится при тщательном тестировании созданного модуля-драйвера — это утилита разрешения сетевых имён в IP адреса и наоборот:

15 `ifconfig` является очень давним, универсальным инструментом управления сетью, присутствующим во всех POSIX операционных системах (Solaris, *BSD, QNX, MINIX, ...), а утилита `ip` появилась намного позже, и является, главным образом «Linux изобретением», хотя позже начала с успехом использоваться и в других системах.

```
$ nslookup yandex.ru 192.168.1.1
Server:        192.168.1.1
Address:       192.168.1.1#53
Non-authoritative answer:
Name:  yandex.ru
Address: 213.180.204.11
Name:  yandex.ru
Address: 93.158.134.11
Name:  yandex.ru
Address: 213.180.193.11
```

Вторым параметром команды (необязательным) является IP адрес DNS-сервера, через который требуется выполнить разрешение имён (при его отсутсвии будет использована последовательность DNS, конфигурированных в системе по умолчанию).

Для **анализа трафика** разрабатываемого сетевого интерфейса вам безусловно потребуются что-то из числа известных утилит **сетевых сниферов**, таких как `tcpdump` (http://www.tcpdump.org/), или её GUI эквивалент `Wireshark` (http://www.wireshark.org/). Посмотрим как для одного из сетевых интерфейсов (`p7p1`) выглядит результат `tcpdump`:

```
$ ip addr show dev p7p1
3: p7p1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:9e:02:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global p7p1
    inet6 fe80::a00:27ff:fe9e:202/64 scope link
       valid_lft forever preferred_lft forever
```

Вот как может выглядеть полученный в `tcpdump` протокол (показано только начало) выполнения операции `ping` на этот интерфейс с внешнего хоста LAN:

```
$ sudo tcpdump -i p7p1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on p7p1, link-type EN10MB (Ethernet), capture size 65535 bytes
08:57:53.070217 ARP, Request who-has 192.168.56.101 tell 192.168.56.1, length 46
08:57:53.070271 ARP, Reply 192.168.56.101 is-at 08:00:27:9e:02:02 (oui Unknown), length 28
08:57:53.070330 IP 192.168.56.1 > 192.168.56.101: ICMP echo request, id 2478, seq 1, length 64
08:57:53.070373 IP 192.168.56.101 > 192.168.56.1: ICMP echo reply, id 2478, seq 1, length 64
08:57:54.071415 IP 192.168.56.1 > 192.168.56.101: ICMP echo request, id 2478, seq 2, length 64
08:57:54.071464 IP 192.168.56.101 > 192.168.56.1: ICMP echo reply, id 2478, seq 2, length 64
...
```

Видно работу ARP механизма разрешения IP адресов в локальной сети (начало протокола), и приём и передачу IP пакетов (тип протокола ICMP).

Примечание: В примере специально показано имя (`p7p1`) проводного (Ethernet) сетевого интерфейса в том виде, который он может иметь в некоторых (Fedora 16, 17) дистрибутивах Linux (вместо `eth0, eth1, ...`). Такое (новое) обозначение увязывает сетевой интерфейс с адресом реализующего его реального сетевого адаптера на шине PCI.

Инструменты тестирования

Ранее обсуждалось, что для тестирования драйверов устройств оптимальными тестерами являются стандартные утилиты POSIX/GNU, такие как `echo`, `cat`, `cp` и другие. Для тестирования и отладки сетевых модулей ядра также хорошо бы предварительно определиться с набором тестовых инструментов (утилит), которые также были бы относительно стандартизованы (или широко используемые), и которые позволяли бы проводить тестирование наиболее быстро, с наименьшими затратами.

Один из удачных вариантов тестеров могут быть утилиты передачи файлов по протоколу SSH — `sftp` и `scp`. Обе утилиты копируют указанный (по URL) **сетевой файл**. Разница состоит в том, что `sftp` требует указания только источника и копирует его в текущий каталог, а для `scp` указывается и источник и приёмник (и каждый из них может быть сетевым URL, таким образом допускается выполнение копирования и из 3-го, стороннего узла):

```
$ sftp olej@192.168.1.9:/home/olej/YYY
olej@192.168.1.9's password:
Connected to 192.168.1.9.
Fetching /home/olej/YYY to YYY
/home/olej/YYY                            100% 98MB 10.9MB/s 00:09

$ scp olej@192.168.1.137:/boot/initramfs-3.6.11-5.fc17.i686.img img1
olej@192.168.1.137's password:
initramfs-3.6.11-5.fc17.i686.img          100%   18MB  17.6MB/s   00:01
```

Ещё одним эффективным инструментом тестирования и отладки может быть утилита nc (network cat). У этой утилиты есть множество возможностей (описанных в man), в частности она позволяет как передавать данные в сеть (клиент), так и принимать эти данные из сети (сервер):

```
$ echo 0123456789 > dg.txt
$ cat dg.txt | nc -l 12345

$ nc 192.168.56.1 12345 > file.txt
$ cat file.txt
0123456789
```

Здесь в первой группе команд nc запускается как **клиент**, во второй — как **сервер**, а параметр 12345 — это согласованный номер порта TCP, через который происходит передача, а 192.168.56.1 — IP-адрес узла, **с которого** осуществляет приём сервер. По умолчанию используется протокол передачи TCP, но его можно изменить на UDP, используя для nc опцию -u.

В отношении сетевых портов транспортного уровня L4 (TCP, UDP, а также менее известных транспортных протоколов SCTP и DCCP) напомним следующее... Количество портов ограничено с учётом 16-битной адресации (2^{16}=65536, начало — «0»). Все порты разделены на три диапазона — **общеизвестные** (или **системные**, 0—1023), **зарегистрированные** (или **пользовательские**, 1024—49151) и **динамические** (или **частные**, 49152—65535).

Динамические и/или приватные порты — от 49152 до 65535. Эти порты динамические в том смысле, что они могут быть использованы любым процессом и с любой целью. Часто, программа, работающая на зарегистрированном порту (от 1024 до 49151) порождает другие процессы, которые затем используют эти динамические порты.

Самая свежая информация о регистрации номеров портов может быть найдена здесь: http://www.iana.org/numbers.htm#P

Структуры данных сетевого стека

Сетевая реализация построена так, чтобы не зависеть от конкретики протоколов. Основной структурой данных описывающей **сетевой интерфейс** (устройство) является struct net_device, к ней мы вернёмся позже, описывая устройство.

А вот **основной** структурой обмениваемых **данных** (между сетевыми уровнями), на движении экземпляров данных которой между сетевыми уровнями построена работа всей подсистемы — это есть буферы сокетов (определения в <linux/skbuff.h>). Буфер сокетов состоит их двух частей: данные управления struct sk_buff, и данные пакета (указываемые в struct sk_buff указателями head и data). Буферы сокетов всегда увязываются в очереди (struct sk_queue_head) посредством своих двух первых полей next и prev. Вот некоторые поля структуры, которые позволяют представить её структуру:

```
typedef unsigned char *sk_buff_data_t;
struct sk_buff {
   struct sk_buff *next; /* These two members must be first. */
   struct sk_buff *prev;
...
   sk_buff_data_t  transport_header;
```

```
    sk_buff_data_t  network_header;
    sk_buff_data_t  mac_header;
...
    unsigned char *head,
                  *data;
...
};
```

Структура вложенности заголовков сетевых уровней в точности соответствует структуре инкапсуляции сетевых протоколов протоколов внутри друг друга, это позволяет обрабатывающему слою получать доступ к информации, относящейся только к нужному ему слою.

Экземпляры данных типа `struct sk_buff`:

- Возникают при поступлении очередного сетевого пакета (здесь нужно принимать во внимание возможность сегментации пакетов) из внешней физической среды распространения данных. Об этом событии извещает прерывание (IRQ), генерируемое сетевым адаптером. При этом создаётся (чаще извлекается из пула использованных) экземпляр буфера сокета, заполняется данными из поступившего пакета и далее этот экземпляр передаётся **вверх** от сетевого слоя к слою, до приложения **прикладного уровня**, которое является получателем пакета. На этом экземпляр данных буфера сокета уничтожается (утилизируется).

- Возникают в среде приложения **прикладного уровня**, которое является отправителем пакета данных. Пакет отправляемых данных помещается в созданный буфер сокета, который начинает перемещаться вниз от сетевого слоя к слою, до достижения канального уровня L2. На этом уровне осуществляется физическая передача данных пакета через сетевой адаптер в среду распространения. В случае успешного завершения передачи (что подтверждается прерыванием, генерируемым сетевым адаптером, часто по той же линии IRQ, что и при приёме пакета) буфер сокета уничтожается (утилизируется). При отсутствии подтверждения отправки (IRQ) обычно делается несколько повторных попыток, прежде, чем принять решение об ошибке канала.

Прохождение экземпляра данных буфера сокета сквозь стек сетевых протоколов будет детально проанализировано далее.

Драйверы: сетевой интерфейс

Задача сетевого интерфейса — быть тем местом, в котором:

- создаются экземпляры структуры `struct sk_buff`, по каждому принятому из интерфейса пакету (здесь нужно принимать во внимание возможность сегментации IP пакетов), далее созданный экземпляр структуры продвигается по стеку протоколов вверх, до получателя пользовательского пространства, где он и уничтожается;
- исходящие экземпляры структуры `struct sk_buff`, порождённые где-то на верхних уровнях протоколов пользовательского пространства, должны отправляться (чаще всего каким-то аппаратным механизмом), а сами экземпляры структуры после этого — уничтожаться.

Более детально эти вопросы мы рассмотрим чуть позже, при обсуждении прохождения пакетов сквозь стек сетевых протоколов. А пока наша задача — **создание** той концевой точки (интерфейса), где эти последовательности действий начинаются и завершаются.

Создание сетевых интерфейсов

Ниже показан пример простого создания и регистрации в системе нового сетевого интерфейса (многие примеры этого раздела заимствованы из [6] с небольшими модификациями, и находятся в архиве `net.tgz`):

network.c :

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

static struct net_device *dev;

static int my_open( struct net_device *dev ) {
   printk( KERN_INFO "Hit: my_open(%s)\n", dev->name );
   /* start up the transmission queue */
   netif_start_queue( dev );
   return 0;
}

static int my_close( struct net_device *dev ) {
   printk( KERN_INFO "Hit: my_close(%s)\n", dev->name );
   /* shutdown the transmission queue */
   netif_stop_queue( dev );
   return 0;
}

/* Note this method is only needed on some; without it
   module will fail upon removal or use. At any rate there is a memory
   leak whenever you try to send a packet through in any case*/
static int stub_start_xmit( struct sk_buff *skb, struct net_device *dev ) {
   dev_kfree_skb( skb );
   return 0;
}

static struct net_device_ops ndo = {
   .ndo_open = my_open,
   .ndo_stop = my_close,
   .ndo_start_xmit = stub_start_xmit,
};

static void my_setup( struct net_device *dev ) {
   int j;
   /* Fill in the MAC address with a phoney */
   for( j = 0; j < ETH_ALEN; ++j )
      dev->dev_addr[ j ] = (char)j;
   ether_setup( dev );
   dev->netdev_ops = &ndo;
}

static int __init my_init( void ) {
   printk( KERN_INFO "Loading stub network module:...." );
   dev = alloc_netdev( 0, "fict%d", my_setup );
   if( register_netdev( dev ) ) {
      printk( KERN_INFO " Failed to register\n" );
      free_netdev( dev );
      return -1;
   }
   printk( KERN_INFO "Succeeded in loading %s!\n", dev_name( &dev->dev ) );
   return 0;
}

static void __exit my_exit( void ) {
   printk( KERN_INFO "Unloading stub network module\n" );
   unregister_netdev( dev );
```

```
    free_netdev( dev );
}

module_init( my_init );
module_exit( my_exit );

MODULE_AUTHOR( "Bill Shubert" );
MODULE_AUTHOR( "Jerry Cooperstein" );
MODULE_AUTHOR( "Tatsuo Kawasaki" );
MODULE_DESCRIPTION( "LDD:1.0 s_24/lab1_network.c" );
MODULE_LICENSE( "GPL v2" );
```

Здесь нужно обратить внимание на вызов `alloc_netdev()`, который в качестве параметра получает шаблон (`%d`) имени нового интерфейса: мы задаём префикс имени интерфейса (`fict`), а система присваивает сама первый свободный номер интерфейса с таким префиксом. Обратите также внимание как в цикле заполнился фиктивным значением `00:01:02:03:04:05` MAC-адрес интерфейса, что мы увидим вскоре в диагностике.

Новая схема и детальнее о создании

Описанная выше схема использовалась в неизменном виде на протяжении, как минимум, 5-ти последних лет, и в таком виде описана везде в литературе и обсуждениях. Но начиная с ядра 3.17 прототип **макроса** создания интерфейса меняется (`<linux/netdevice.h>`)...

- было:
```
#define alloc_netdev( sizeof_priv, name, setup )
```
- стало:
```
#define alloc_netdev( sizeof_priv, name, name_assign_type, setup )
```

Как легко видеть, теперь вместо 3-х параметров 4, 3-й из которых — константа, определяющая порядок нумерации создаваемых интерфейсов, описанная в том же файле определений:
```
/* interface name assignment types (sysfs name_assign_type attribute) */
#define NET_NAME_UNKNOWN     0 /* unknown origin (not exposed to userspace) */
#define NET_NAME_ENUM        1 /* enumerated by kernel */
#define NET_NAME_PREDICTABLE 2 /* predictably named by the kernel */
#define NET_NAME_USER        3 /* provided by user-space */
#define NET_NAME_RENAMED     4 /* renamed by user-space */
```

Похоже, что пока (на уровне ядра 3.17) эти различные схемы ещё не дифференцируются, но начата активная работа по модернизации схемы. Следствием же является, что все сетевые драйверы, особенно проприетарные или от производителей, **должны быть переписаны** с учётом новой схемы.

Для иллюстрации и изучения этой схемы приведём модифицированный вариант предыдущего модуля (показаны только существенно отличающиеся части):

mulnet.c :

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

static int num = 1;              // число создаваемых интерфейсов
module_param( num, int, 0 );
static char* title;              // префикс имени интерфейсов
module_param( title, charp, 0 );
static int digit = 1;            // числовые суфиксы (по умолчанию)
module_param( digit, int, 0 );
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(3, 17, 0))
static int mode = 1;             // режим нумерации интерфейсов
module_param( mode, int, 0 );
#endif
```

```
static struct net_device *adev[] = { NULL, NULL, NULL, NULL };

...

static int ipos;

static void __init my_setup( struct net_device *dev ) {
   /* Fill in the MAC address with a phoney */
   int j;
   for( j = 0; j < ETH_ALEN; ++j )
      dev->dev_addr[ j ] = (char)( j + ipos );
   ether_setup( dev );
   dev->netdev_ops = &ndo;
}

static int __init my_init( void ) {
   char prefix[ 20 ];
   if( num > sizeof( adev ) / sizeof( adev[ 0 ] ) ) {
      printk( KERN_INFO "! link number error" );
      return -EINVAL;
   }
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(3, 17, 0))
   if( mode < 0 || mode > NET_NAME_RENAMED ) {
      printk( KERN_INFO "! unknown name assign mode" );
      return -EINVAL;
   }
#endif
   printk( KERN_INFO "! loading network module for %d links", num );
   sprintf( prefix, "%s%s", ( NULL == title ? "fict" : title ), "%d" );
   for( ipos = 0; ipos < num; ipos++ ) {
      if( !digit )
         sprintf( prefix, "%s%c", ( NULL == title ? "fict" : title ), 'a' + ipos );
#if (LINUX_VERSION_CODE < KERNEL_VERSION(3, 17, 0))
      adev[ ipos ] = alloc_netdev( 0, prefix, my_setup );
#else
      adev[ ipos ] = alloc_netdev( 0, prefix, NET_NAME_UNKNOWN + mode, my_setup );
#endif
      if( register_netdev( adev[ ipos ] ) ) {
         int j;
         printk( KERN_INFO "! failed to register" );
         for( j = ipos; j >= 0; j-- ) {
            if( j != ipos ) unregister_netdev( adev[ ipos ] );
            free_netdev( adev[ ipos ] );
         }
         return -ELNRNG;
      }
   }
   printk( KERN_INFO "! succeeded in loading %d devices!", num );
   return 0;
}

static void __exit my_exit( void ) {
   int i;
   printk( KERN_INFO "! unloading network module" );
   for( i = 0; i < num; i++ ) {
      unregister_netdev( adev[ i ] );
      free_netdev( adev[ i ] );
   }
}
```

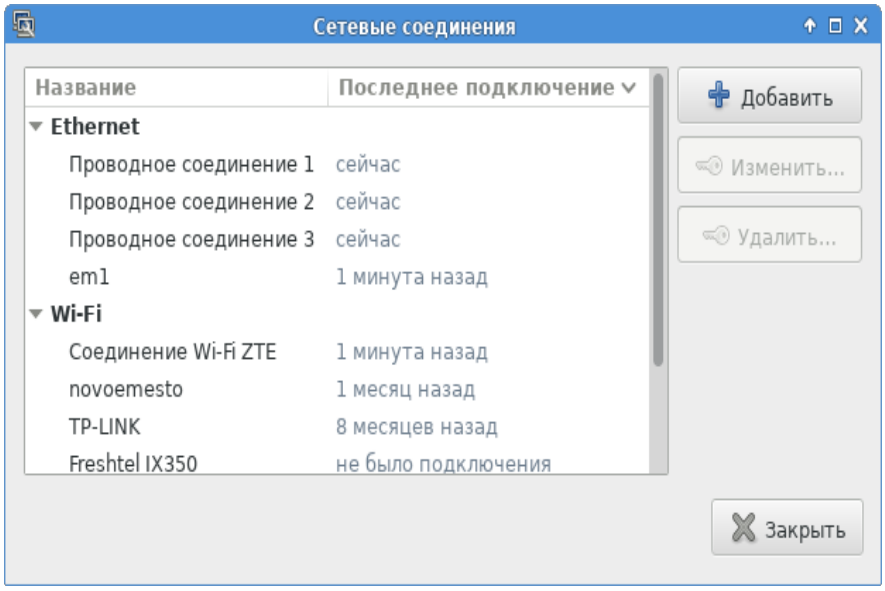
```
...
```

В дополнение, здесь показано создание нескольких однотипных сетевых интерфейсов (параметр `num=...`) и возможность «ручного» присвоения (параметр `digit=0`) им произвольных имён (без формата "%d"):

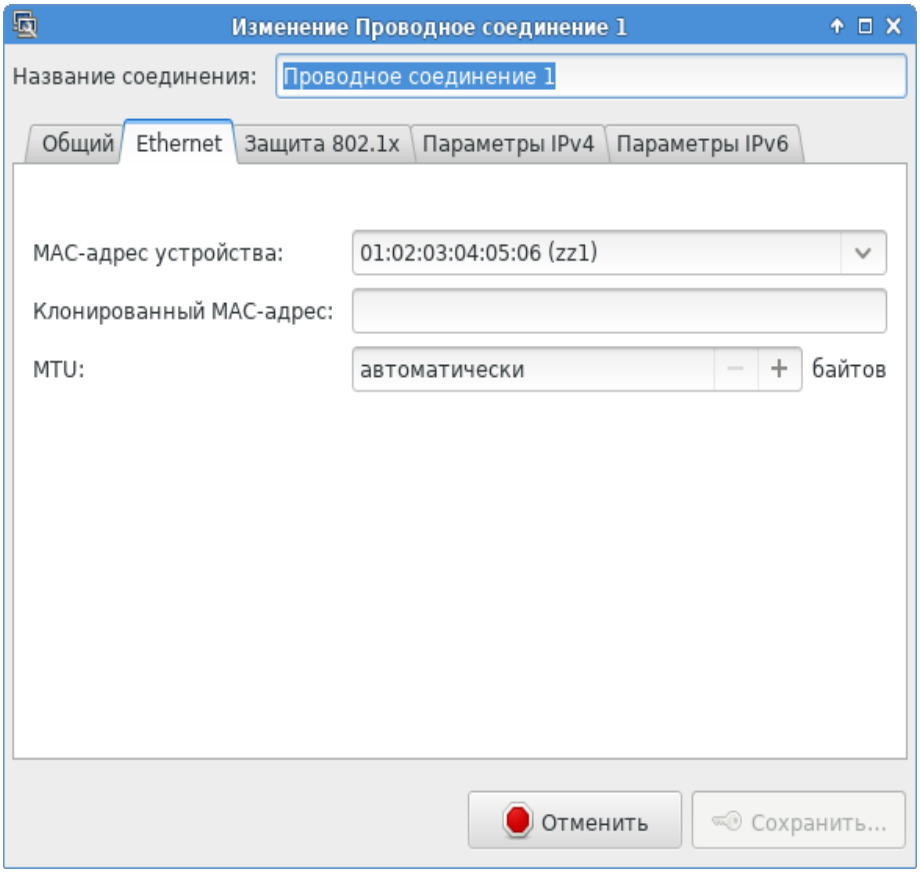
```
$ sudo insmod mulnet.ko num=3 title=zz mode=2
$ ip link
...
    link/ether 34:23:87:d6:85:0d brd ff:ff:ff:ff:ff:ff
7: zz0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 00:01:02:03:04:05 brd ff:ff:ff:ff:ff:ff
8: zz1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 01:02:03:04:05:06 brd ff:ff:ff:ff:ff:ff
9: zz2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 02:03:04:05:06:07 brd ff:ff:ff:ff:ff:ff
```

А вот как выглядят интерфейсы, если они создаются без числовых суффиксов, в режиме ручного формирования имён:

```
$ sudo insmod mulnet.ko num=3 digit=0
$ ip link
...
4: ficta: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 00:01:02:03:04:05 brd ff:ff:ff:ff:ff:ff
5: fictb: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 01:02:03:04:05:06 brd ff:ff:ff:ff:ff:ff
6: fictc: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 02:03:04:05:06:07 brd ff:ff:ff:ff:ff:ff
```



Если в вашей Linux системе для управления сетевой подсистемой используется аплет Network Manager, то, непосредственно после запусков показанных выше модулей, созданные интерфейсы сразу же отобразятся в нём...



Операции сетевого интерфейса

Для того, чтобы «придать жизнь» созданному сетевому интерфейсу, нужно реализовать для него набор обменных операций, чему и будет посвящена оставшаяся часть этого раздела... Вся связь сетевого интерфейса с выполняемыми на нём **операциями** осуществляется через таблицу функций операций сетевого интерфейса (**n**et **d**evice **o**perations):

```
struct net_device_ops {
        int                     (*ndo_init)(struct net_device *dev);
        void                    (*ndo_uninit)(struct net_device *dev);
        int                     (*ndo_open)(struct net_device *dev);
        int                     (*ndo_stop)(struct net_device *dev);
        netdev_tx_t             (*ndo_start_xmit) (struct sk_buff *skb,
                                                   struct net_device *dev);
        ...
        struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
        ...
}
```

В ядре 3.09, например, определено 39 операций в `struct net_device_ops`, (и около 50-ти операций в ядре 3.14), но реально разрабатываемые модули реализуют только некоторую малую часть из них.

Характерно, что в таблице операций интерфейса присутствует операция **передачи** сокетного буфера `ndo_start_xmit` в физическую среду, но вовсе нет операции **приёма** пакетов (сокетных буферов). Это совершенно естественно, как мы увидим вскоре: принятые пакеты (например в обработчике аппаратеного прерывания IRQ) тут же передаются в очередь (ядра) принимаемых пакетов, и далее уже обрабатываются сетевым стеком. А вот выполнять операцию `ndo_start_xmit` — обязательно, хотя бы, как минимум, для вызова API ядра `dev_kfree_skb()`, который утилизирует (уничтожает) сокетный буфер после успешной (да и безуспешной тоже) операции передачи пакета. Если этого не делать, в системе возникнет слабо выраженная утечка памяти (с каждым пакетом), которая, в конечном итоге, рано или поздно приведёт к краху системы.

Теперь созданное нами выше (пока фиктивное) сетевое устройство уже можно установить в системе:

```
$ sudo insmod ./network.ko
$ dmesg | tail -n4
[ 7355.005588] Loading stub network module:....
[ 7355.005597] my_setup()
[ 7355.006703] Succeeded in loading fict0!
$ ip link show dev fict0
5: fict0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether 00:01:02:03:04:05 brd ff:ff:ff:ff:ff:ff
$ sudo ifconfig fict0 192.168.56.50
$ dmesg | tail -n6
[ 7355.005588] Loading stub network module:....
[ 7355.005597] my_setup()
[ 7355.006703] Succeeded in loading fict0!
[ 7562.604588] Hit: my_open(fict0)
[ 7573.442094] fict0: no IPv6 routers present
$ ping 192.168.56.50
PING 192.168.56.50 (192.168.56.50) 56(84) bytes of data.
64 bytes from 192.168.56.50: icmp_req=1 ttl=64 time=0.253 ms
64 bytes from 192.168.56.50: icmp_req=2 ttl=64 time=0.056 ms
64 bytes from 192.168.56.50: icmp_req=3 ttl=64 time=0.057 ms
64 bytes from 192.168.56.50: icmp_req=4 ttl=64 time=0.056 ms
^C
--- 192.168.56.50 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.056/0.105/0.253/0.085 ms
```

Но показанное это вовсе не означает какую-то реальную работоспособность созданного интерфейса — создаваемый поток сокетных буферов замыкается в петлю внутри самого сетевого стека:

```
$ ifconfig fict0
fict0     Link encap:Ethernet  HWaddr 00:01:02:03:04:05
          inet addr:192.168.56.50  Bcast:192.168.56.255  Mask:255.255.255.0
          inet6 addr: fe80::201:2ff:fe03:405/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

Обратите внимание, как совершенно **произвольное значение** заполняется в структуре `net_device`, и устанавливается в качестве MAC (аппаратного) адреса созданного интерфейса (в функции `my_setup()`).

Как уже отмечалось выше, основу структуры описания сетевого интерфейса составляет структура `struct net_device`, описанная в `<linux/netdevice.h>`. При работе с сетевыми интерфейсами эту структуру стоит изучить весьма тщательно. Это очень крупная структура, содержащая не только описание аппаратных средств, но и конфигурационные параметры сетевого интерфейса по отношению к выше лежащим протоколам (пример взят из ядра 3.09):

```
struct net_device {
   char  name[ IFNAMSIZ ] ;
...
   unsigned long  mem_end;   /* shared mem end     */
   unsigned long  mem_start; /* shared mem start   */
   unsigned long  base_addr; /* device I/O address */
   unsigned int   irq;       /* device IRQ number  */
...
   unsigned       mtu;       /* interface MTU value     */
   unsigned short type;      /* interface hardware type */
...
   struct net_device_stats stats;
```

```
    struct list_head dev_list;
...
    /* Interface address info. */
    unsigned char  perm_addr[ MAX_ADDR_LEN ]; /* permanent hw address    */
    unsigned char  addr_len;                  /* hardware address length */
...
}
```

Здесь поле `type`, например, определяет тип аппаратного адаптера с точки зрения ARP-механизма разрешения MAC адресов (`<linux/if_arp.h>`):

```
...
#define ARPHRD_ETHER       1    /* Ethernet 10Mbps           */
...
#define ARPHRD_IEEE802     6    /* IEEE 802.2 Ethernet/TR/TB */
#define ARPHRD_ARCNET      7    /* ARCnet                    */
...
#define ARPHRD_IEEE1394   24    /* IEEE 1394 IPv4 - RFC 2734 */
...
#define ARPHRD_IEEE80211 801    /* IEEE 802.11               */
```

Здесь же заносятся такие совершенно аппаратные характеристики интерфейса (реализующего его физического адаптера), как, например, адрес базовой области ввода-вывода (`base_addr`), используемая линия аппаратного прерывания (`irq`), максимальная длина пакета для данного интерфейса (`mtu`)...

Детальный разбор огромного числа полей `struct net_device` (этой и любой другой сопутствующей) или их возможных значений — бессмысленный, хотя бы потому, что эта структура радикально изменяется от подверсии к подверсии ядра; такой разбор должен проводиться «по месту» на основе изучения названных выше заголовочных файлов.

Со структурой сетевого интерфейса обычно создаётся и связывается (кодом модуля) **приватная структура данных**, в которой пользователь может размещать произвольные собственные данные любой сложности, ассоциированные с интерфейсом. Это обычная практика ядра Linux, и не только сетевой подсистемы. Указатель такой приватной структуры помещается в структуру сетевого интерфейса. Это особо актуально, если предполагается, что драйвер может создавать несколько сетевых интерфейсов (например, несколько идентичных сетевых адаптеров). Доступ к приватной структуре данных должен определяться **исключительно** специально определённой для того функцией `netdev_priv()`. Ниже показан возможный вид функции — это определение из ядра 3.09, но никто не даст гарантий, что в другом ядре оно радикально не поменяется:

```
/*     netdev_priv - access network device private data
 * Get network device private data
 */
static inline void *netdev_priv( const struct net_device *dev ) {
        return (char *)dev + ALIGN( sizeof( struct net_device ), NETDEV_ALIGN );
}
```

Примечание: Как легко видеть из определения, приватная структура данных дописывается **непосредственно в хвост** `struct net_device` - это обычная практика создания структур переменного размера, принятая в языке C начиная с стандарта C89 (и в C99). Но именно из-за этого, за счёт эффектов **выравнивания** данных (и его возможного изменения в будущем), не следует адресоваться к приватным данным непосредственно, а следует использовать `netdev_priv()`.

При начальном размещении интерфейса размер определённой пользователем приватной структуры передаётся первым параметром функции размещения, например так:

```
child = alloc_netdev( sizeof( struct priv ), "fict%d", &setup );
```

После успешного выполнения размещения интерфейса приватная структура также будет размещена («в хвост» структуре `struct net_device`), и будет доступна по вызову `netdev_priv()`.

Все структуры `struct net_device`, описывающие доступные сетевые интерфейсы в системе, увязаны в единый связный список.

Примечание: В ядре Linux **все и любые** списочные связные структуры строятся на основе API кольцевых двухсвязных списков, структур данных `struct list_head` (`поле dev_list` в `struct net_device`). Техника связных списков в ядре Linux будет подробно рассмотрена позже, в части API ядра.

Следующий пример визуализирует содержимого списка сетевых интерфейсов:

devices.c :

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

static int __init my_init( void ) {
   struct net_device *dev;
   printk( KERN_INFO "Hello: module loaded at 0x%p\n", my_init );
   dev = first_net_device( &init_net );
   printk( KERN_INFO "Hello: dev_base address=0x%p\n", dev );
   while ( dev ) {
      printk( KERN_INFO
              "name = %6s irq=%4d trans_start=%12lu last_rx=%12lu\n",
              dev->name, dev->irq, dev->trans_start, dev->last_rx );
      dev = next_net_device( dev );
   }
   return -1;
}

module_init( my_init );
```

Выполнение (предварительно для убедительности загрузим ранее созданный модуль `network.ko`):

```
$ sudo insmod network.ko
$ sudo insmod devices.ko
insmod: error inserting 'devices.ko': -1 Operation not permitted
$ dmesg | tail -n8
Hello: module loaded at 0xf8853000
Hello: dev_base address=0xf719c400
name =     lo  irq=   0  trans_start=           0  last_rx=           0
name =   eth0  irq=  16  trans_start=  4294693516  last_rx=           0
name =  wlan0  irq=   0  trans_start=  4294693412  last_rx=           0
name =   pan0  irq=   0  trans_start=           0  last_rx=           0
name = cipsec0 irq=   0  trans_start=     2459232  last_rx=           0
name = mynet0  irq=   0  trans_start=           0  last_rx=           0
```

Переименование сетевого интерфейса

Только-что, в примере выше, было показано, как имя сетевого интерфейса задаётся модулем, создающим этот интерфейс. Но имя сетевого интерфейса (созданного модулем, или присутствующего в системе) не есть совершенно константное значение, которое должно оставаться неизменным во всё время работы интерфейса в системе. Сравните:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN mode DEFAULT qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
$ ip link show wlan0
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN mode DEFAULT qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
```

```
$ sudo ip link set dev wlan0 name wln2
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wln2: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN mode DEFAULT qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
```

Здесь исходный WiFi-интерфейс `wlan0` был переименован командой `ip` в новое имя `wln2`. Далее с этим новым именем интерфейса работают все сетевые утилиты и протоколы.

Это не имеет прямого отношения к программированию модулей ядра, но оказывается чрезвычайно **мощным** инструментом при тестировании модулей, при экспериментах, или при конфигурировании программных пакетов, включающих модули ядра сетевых устройств, которые создают свои новые сетевые интерфейсы. Но для использования этого инструмента нужно учесть несколько факторов...

- пусть мы имеем первоначально систему с двумя интерфейсами (`p2p1` и `p7p1`):

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 p2p1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 p2p1
192.168.56.1    0.0.0.0         255.255.255.255 UH    0      0        0 p7p1
```

- изменить имя можно только для остановленного интерфейса, в противном случае интерфейс будет «занят» для операции (в первоначальном примере `wlan0` и был как-раз в остановленном состоянии):

```
$ sudo ip link set dev p7p1 name crypt0
RTNETLINK answers: Device or resource busy
```

- интерфейс нужно остановить для выполнения переименования, после чего снова поднять:

```
$ sudo ifconfig p7p1 down
$ sudo ip link set dev p7p1 name crypt0
$ sudo ifconfig crypt0 192.168.56.4
$ ip address show dev crypt0
3: crypt0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:08:9a:bd brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.3/32 brd 192.168.56.3 scope global crypt0
    inet6 fe80::a00:27ff:fe08:9abd/64 scope link
       valid_lft forever preferred_lft forever
```

- но после остановки и перезапуска интерфейса разрушается соответствующая ему запись в таблице роутинга (такое произошло бы если бы мы и не переименовывали интерфейс):

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 p2p1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 p2p1
$ sudo route add -net 192.168.56.0 netmask 255.255.255.0 dev crypt0
$ route -n
Kernel IP routing table
```

- при работающем интерфейсе `crypt0` с присвоенным ему IP создаётся впечатление его неработоспособности; для восстановления работы нужно добавить запись об интерфейсе в таблицу роутинга:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 p2p1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 p2p1
192.168.56.0    0.0.0.0         255.255.255.0   U     0      0        0 crypt0
```

- после чего можно убедиться в полной работоспособности интерфейса с новым именем:

```
$ ping 192.168.56.1
PING 192.168.56.1 (192.168.56.1) 56(84) bytes of data.
64 bytes from 192.168.56.1: icmp_req=1 ttl=64 time=0.422 ms
64 bytes from 192.168.56.1: icmp_req=2 ttl=64 time=0.239 ms
^C
--- 192.168.56.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.239/0.330/0.422/0.093 ms
```

Путь пакета сквозь стек протоколов

Теперь у нас достаточно деталей, чтобы проследить путь пакетов (буферов сокетов) сквозь сетевой стек, проследить то, как буфера сокетов возникают в системе, и когда они её покидают, а также ответить на вопрос, почему вышележащие протокольные уровни (будут рассмотрены чуть ниже) никогда не порождают и не уничтожают буферов сокетов, а только обрабатывают (или модифицируют) содержащуюся в них информацию (работают как фильтры). Итак, последовательность связей мы можем разложить в таком порядке:

Приём: традиционный подход

Традиционный подход состоит в том, что каждый приходящий сетевой пакет порождает аппаратное прерывание по линии IRQ адаптера, что и служит сигналом на приём очередного сетевого пакета и создание буфера сокета для его сохранения и обработки принятых данных. Порядок действий модуля сетевого интерфейса при этом следующий:

1. Читая конфигурационную область PCI адартера сети при инициализации модуля, определяем линию прерывания IRQ, которая будет обслуживать сетевой обмен:

```
char irq;
pci_read_config_byte( pdev, PCI_INTERRUPT_LINE, &byte );
```

Точно таким же манером будет определена и область адресов ввода-адресов адаптера, скорее всего, через DMA ... - всё это рассматривается позже, при рассмотрении аппаратных шин.

2. При инициализации сетевого интерфейса, для этой линии IRQ устанавливается обработчик прерывания `my_interrupt()` :

```
request_irq( (int)irq, my_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id );
```

3. В обработчике прерывания, по приёму нового пакета из сети (то же прерывание может происходить и при завершении отправки пакета в сеть, здесь нужен анализ причины), создаётся (или запрашивается из пула используемых) новый экземпляр буфера сокетов:

```
static irqreturn_t my_interrupt( int irq, void *dev_id ) {
   ...
   struct sk_buff *skb = kmalloc( sizeof( struct sk_buff ), ... );
   // заполнение данных *skb чтением из портов сетевого адаптера
   netif_rx( skb );
   return IRQ_HANDLED;
}
```

Все эти действия выполняются не в самом обработчике верхней половины прерываний от сетевого адаптера, а в обработчике отложенного прерывания `NET_RX_SOFTIRQ` для этой линии. Последним действием является передача заполненного сокетного буфера вызову `netif_rx()` (или `netif_receive_skb()`) который и запустит процесс движения его (буфера) вверх по структуре сетевого стека (отметит отложенное программное прерывание `NET_RX_SOFTIRQ` для исполнения).

Приём: высокоскоростной интерфейс

Особенность природы сетевых интерфейсов состоит в том, что их активность носит взрывной характер: после весьма продолжительных периодов молчания возникают интервалы пиковой активности, когда сетевые пакеты (сегментированные на IP пакеты объёмы передаваемых данных) следуют сплошной плотной чередой. После такого пика активности могут снова наступать значительные промежутки полного отсутствия активности, или вялой активности на интерфейсе (обмен ARP пакетами для обновления информации разрешения локальных адресов и подобные виды активности). Современные Ethernet сетевые карты используют скорости обмена до 10Gbit/s, но уже даже при значительно ниже интенсивностях традиционный подход становится нецелесообразным: в периоды высокой плотности поступления пакетов:

- новые приходящие пакеты создают вложенные запросы IRQ нескольких уровней при ещё не обслуженном приёме текущего IRQ;
- асинхронное обслуживание каждого IRQ в плотном потоке создаёт слишком большие накладные расходы;

Поэтому был добавлен набор API для обработки таких плотных потоков пакетов, поступающих с высокоскоростных интерфейсов, который и получил название NAPI (New API[16]). Идея состоит в том, чтобы приём пакетов осуществлять не методом аппаратного прерывания, а методом **программного опроса** (polling), точнее, комбинацией этих двух возможностей:

- при поступлении **первого** пакета «пачки» инициируется прерывание IRQ адаптера (всё начинается как в традиционном методе)...
- в обработчике прерывания **запрещается** поступление дальнейших запросов прерывания с этой линии IRQ по приёму пакетов, IRQ с этой же линии по отправке пактов могут продолжать поступать, таким образом, этот запрет происходит не программным запретом линии IRQ со стороны процессора, а записью управляющей информации в **аппаратные регистры** сетевого адаптера, адаптер должен предусматривать такое раздельное управление поступлением прерываний по приёму и передаче, но для современных высокоскоростных адаптеров это, обычно, соблюдается;
- после прекращения прерываний по приёму обработчик переходит в режим циклического считывания и обработки принятых из сети пакетов, сетевой адаптер при этом накапливает поступающие пакеты во внутреннем кольцевом буфере приёма, а считывание производится либо до полного исчерпания кольцевого буфера, либо до опередёлённого порогового числа считанных пакетов (10, 20, ...), называемого бюджетом функции полинга;
- естественно, это считывание и обработка пакетов происходит не в собственно обработчике прерывания (верхней половине), а в его отсроченной части;
- по каждому принятому в опросе пакету генерируется сокетный буфер для продвижения его по стеку сетевыз протоколов вверх;
- после **завершения цикла** программного опроса, по его результатам устанавливается состояние завершения `NAPI_STATE_DISABLE` (если не осталось больше не сосчитанных пакетов в кольцевом буфере адаптера), или `NAPI_STATE_SCHED` (что говорит, что устройство адаптера должно продолжать опрашиваться когда ядро следующий раз перейдёт к циклу опросов в отложенном обработчике прерываний).
- если результатом является `NAPI_STATE_DISABLE`, то после завершения цикла программного опроса восстанавливается разрешение генерации прерываний по линии IRQ приёма пакетов (записью в порты сетевого адаптера);

В реализующем коде модуля это укрупнённо должно выглядеть подобно следующему (при условии, что линия IRQ связана с аппаратным адаптером, как это описано для традиционного метода):

1. Реализатор обязан предварительно создать и зарегистрировать специфичную для модуля функцию опроса (poll-функцию), используя вызов (`<netdevice.h>`):

```
static inline void netif_napi_add( struct net_device *dev,
                                   struct napi_struct *napi,
```

16 Естественно, до какого времени он будет «новым» неизвестно — до появления ещё более нового.

```
                                    int (*poll)( struct napi_struct *, int ),
                                    int weight );
```

– где:

  dev — это рассмотренная раньше структура зарегистрированного сетевого интерфейса;

  poll — регистрируемая функция программного опроса, о которой ниже;

  weight — относительный вес, приоритет, который придаёт разработчик этому интерфейсу, для 10Mb и 100Mb адаптеров здесь часто указано значение 16, а для 10Gb и 100Gb — значение 64;

  napi — дополнительный параметр, указатель на специальную структуру, которая будет передаваться в каждый вызов функции poll, и где будет, по результату выполнения этой функции, заполняться поле state значениями NAPI_STATE_DISABLE или NAPI_STATE_SCHED, вид этой структуры должен быть (<netdevice.h>):

```
struct napi_struct {
   struct list_head poll_list;
   unsigned long state;
   int weight;
   int (*poll)( struct napi_struct *, int );
};
```

2. Зарегистрированная функция программного опроса (полностью зависимая от задачи и реализуемая в коде модуля) имеет подобный вид:

```
static int my_card_poll( struct napi_struct *napi, int budget ) {
   int work_done; // число реально обработанных в цикле опроса сетевых пакетов
   work_done = my_card_input( budget, ... ); // реализационно специфический приём пакетов
   if( work_done < budget ) {
      netif_rx_complete( netdev, napi );
      my_card_enable_irq( ... );             // разрешить IRQ приёма
   }
   return work_done;
}
```

Здесь пользовательская функция my_card_input() в цикле пытается аппаратно сосчитать budget сетевых пакетов, и для каждого считанного сетевого пакета создаёт сокетный буфер и вызывает netif_receive_skb(), после чего этот буфер начинает движение по стеку протоколов вверх. Если кольцевой буфер сетевого адаптера исчерпался ранее budget пакетов (нет более наличных пакетов), то адаптеру разрешается возбуждать прерывания по приёму, а ядро вызовом netif_rx_complete() уведомляется, что отменяется отложенное программное прерывание NET_RX_SOFTIRQ для дальнейшего вызова функции опроса. Если же удалось сосчитать budget пакетов (в буфере адаптера, видимо, есть ещё не обработанные пакеты), то опрос продолжится при следующем цикле обработки отложенного программного прерывания NET_RX_SOFTIRQ.

3. Обработчик аппаратного прерывания линии IRQ сетевого адаптера (активирующий при приходе **первого** сетевого пакета «пачки» активности) должен выполнять примерно следующее:

```
static irqreturn_t my_interrupt( int irq, void *dev_id ) {
   struct net_device *netdev = dev_id;
   if( likely( netif_rx_schedule_prep( netdev, ... ) ) ) {
      my_card_disable_irq( ... );             // запретить IRQ приёма
      __netif_rx_schedule( netdev, ... );
   }
   return IRQ_HANDLED;
}
```

Здесь ядро должно быть уведомлено, что новая порция сетевых пакетов готова для обработки. Для этого

– вызов netif_rx_schedule_prep() подготавливает устройство для помещение в список для программного опроса, устанавливая состояние в NAPI_STATE_SCHED;

– если предыдущий вызов успешен (а противное возникает только если NAPI уже активен), то вызовом __netif_rx_schedule() устройство помещается в список для программного опроса, в цикле обработки отложенного программного прерывания NET_RX_SOFTIRQ.

Вот, собственно, и всё относительно новой модели приёма сетевых пакетов. Здесь нужно держать в виду, что бюджет, разово устанавливаемый в функции опроса (локальный бюджет), не должен быть чрезмерно большим. По крайней мере:

– Опрос не должен должен потреблять более одного системного тика (глобальная переменная `jiffies`), иначе это будет искажать диспетчеризацию потоков ядра;

– Бюджет не должен быть больше глобально установленного ограничения:

```
$ cat /proc/sys/net/core/netdev_budget
300
```

После каждого цикла опроса число обработанных пакетов (возвращаемых функцией опроса) вычитается из этого глобального бюджета, и если остаток меньше нуля, то обработчик программного прерывания `NET_RX_SOFTIRQ` останавливается.

Передача пакетов

Описанными выше действиями инициируется создание и движение сокетного буфера вверх по стеку. Движение же вниз (при отправке в сеть) обеспечивается по другой цепочке:

1. При инициализации сетевого интерфейса (это момент, который уже был назван выше в п.2), создаётся таблица операций сетевого интерфейса, одно из полей которой `ndo_start_xmit` определяет функцию передачи пакета в сеть:

```
struct net_device_ops ndo = {
   .ndo_open = my_open,
   .ndo_stop = my_close,
   .ndo_start_xmit = stub_start_xmit,
};
```

2. При вызове `stub_start_xmit()` должна обеспечить аппаратную передачу полученного сокета в сеть, после чего уничтожает (возвращает в пул) буфер сокета:

```
static int stub_start_xmit( struct sk_buff *skb, struct net_device *dev ) {
   // ... аппартное обслуживание передачи
   dev_kfree_skb( skb );
   return 0;
}
```

Реально чаще уничтожение отправляемого буфера будет происходить не при инициализации операции, а при её (успешном) завершении, что отслеживается **по той же линии IRQ**, что и приём пакетов из сети.

Часто задаваемый вопрос: а где же в этом процессе место (код), где реально создаётся содержательная информация, **помещаемая** в сокетный буфер, или где **потребляется** информация из принимаемых сокетных буферов? Ответ: не ищите такого места в пределах сетевого стека ядра — любая информация для отправки в сеть, или потребляемая из сети, возникает в поле зрения только на прикладных уровнях, в приложениях пространства пользователя, таких, например, как `ping, ssh, telnet` и великое множество других. Интерфейс из этого прикладного уровня в стек протоколов ядра обеспечивается известным POSIX API сокетов прикладного уровня.

Статистики интерфейса

Процессы, происходящие на сетевом интерфейс, сложно явно наблюдать (в сравнении, скажем, с интерфейсами `/dev` или `/proc`). Поэтому очень важной характеристикой интерфейса становится накопленная статистика происходящих на нём процессов. Для накопления статистики работы сетевого интерфейса описана специальная структура (достаточно большая, определена там же в `<linux/netdevice.h>`, показано только начало структуры) :

```
struct net_device_stats {
```

```
   unsigned long  rx_packets;     /* total packets received    */
   unsigned long  tx_packets;     /* total packets transmitted */
   unsigned long  rx_bytes;       /* total bytes received      */
   unsigned long  tx_bytes;       /* total bytes transmitted   */
   unsigned long  rx_errors;      /* bad packets received      */
   unsigned long  tx_errors;      /* packet transmit problems  */
...
}
```

Поля такой структуры структура должны заполняться кодом модуля статистическими данными проходящих пакетов (при передаче пакета, например, инкрементируя `tx_packets`).

В пространство пользователя эту структуру возвращает функция `ndo_get_stats` в таблице операций `struct net_device_ops` (выше эти поля были специально показаны). Модуль должен реализовать такую собственную функцию и поместить её в `struct net_device_ops`. Это делается, если вы хотите получать статистики сетевого интерфейса пользователем вызовом `ifconfig`, или через интерфейс файловой системы `/proc`, как это ожидаемо и происходит для всех других сетевых интерфейсов:

```
$ ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr 00:13:02:69:70:9B
          inet addr:192.168.1.22  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::213:2ff:fe69:709b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8658 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9070 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4240425 (4.0 MiB)  TX bytes:1318733 (1.2 MiB)
```

Где обычно размещается структура `net_device_stats`, которую мы предполагаем возвращать пользователю? Часто встречаются несколько вариантов:

1. Если модуль обслуживает только один конкретный сетевой интерфейс, то структура может размещаться на глобальном уровне кода модуля.

```
static struct net_device_stats *stats;
...
static struct net_device_stats *my_get_stats( struct net_device *dev ) {
   return &stats;
}
...
static struct net_device_ops ndo = {
   ...
   .ndo_get_stats = my_get_stats,
};
```

2. Часто структура статистики размещается как составная часть структуры **приватных данных** (о которой была речь выше), которую разработчик связывает с сетевым интерфейсом.

```
static struct net_device *my_dev = NULL;
struct my_private {
   struct net_device_stats stats;
   ...
};
...
static struct net_device_stats *my_get_stats( struct net_device *dev ) {
   struct my_private *priv = (my_private*)netdev_priv( dev );
   return &priv->stats;
}
...
static struct net_device_ops ndo = {
   ...
```

```
   .ndo_get_stats = my_get_stats,
}
...
void my_setup( struct net_device *dev ) {
   memset( netdev_priv( dev ), 0, sizeof( struct my_private ) );
   dev->netdev_ops = &ndo;
}
int __init my_init( void ) {
    my_dev = alloc_netdev( sizeof( struct my_private ), "my_if%d", my_setup );
}
```

3. Наконец, может использоваться структура, включённая (имплементированная) непосредственно **в состав** определения интерфейса `struct net_device`.

```
...
static struct net_device_stats *my_get_stats( struct net_device *dev ) {
   return &dev->stats;
}
```

Все эти три варианта использования показаны (для сравнения: файлы `virt.c`, `virt1.c` и `virt2.c` в архиве `virt.tgz`).

Виртуальный сетевой интерфейс

В предыдущих примерах мы создавали сетевые интерфейсы, но они не осуществляли реально с физической средой передачи и приёма. Для выполнения такого уровня проработки нужно бы иметь реальное коммуникационное оборудование на PCI шине, что не всегда доступно. Но мы можем создать интерфейс, который будет перехватывать трафик сетевого ввода-вывода с другого, реально существующего в системе, интерфейса, и обеспечивать обработку этих потоков (архив `virt.tgz`).

virt.c :

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/moduleparam.h>
#include <net/arp.h>

#define ERR(...) printk( KERN_ERR "! "__VA_ARGS__ )
#define LOG(...) printk( KERN_INFO "! "__VA_ARGS__ )

static char* link = "eth0";
module_param( link, charp, 0 );

static char* ifname = "virt";
module_param( ifname, charp, 0 );

static struct net_device *child = NULL;

struct priv {
   struct net_device_stats stats;
   struct net_device *parent;
};

static rx_handler_result_t handle_frame( struct sk_buff **pskb ) {
   struct sk_buff *skb = *pskb;
   if( child ) {
```

```
      struct priv *priv = netdev_priv( child );
      priv->stats.rx_packets++;
      priv->stats.rx_bytes += skb->len;
      LOG( "rx: injecting frame from %s to %s", skb->dev->name, child->name );
      skb->dev = child;
      return RX_HANDLER_ANOTHER;
   }
   return RX_HANDLER_PASS;
}

static int open( struct net_device *dev ) {
   netif_start_queue( dev );
   LOG( "%s: device opened", dev->name );
   return 0;
}

static int stop( struct net_device *dev ) {
   netif_stop_queue( dev );
   LOG( "%s: device closed", dev->name );
   return 0;
}

static netdev_tx_t start_xmit( struct sk_buff *skb, struct net_device *dev ) {
   struct priv *priv = netdev_priv( dev );
   priv->stats.tx_packets++;
   priv->stats.tx_bytes += skb->len;
   if( priv->parent ) {
      skb->dev = priv->parent;
      skb->priority = 1;
      dev_queue_xmit( skb );
      LOG( "tx: injecting frame from %s to %s", dev->name, skb->dev->name );
      return 0;
   }
   return NETDEV_TX_OK;
}

static struct net_device_stats *get_stats( struct net_device *dev ) {
   return &( (struct priv*)netdev_priv( dev ) )->stats;
}

static struct net_device_ops crypto_net_device_ops = {
   .ndo_open = open,
   .ndo_stop = stop,
   .ndo_get_stats = get_stats,
   .ndo_start_xmit = start_xmit,
};

static void setup( struct net_device *dev ) {
   int j;
   ether_setup( dev );
   memset( netdev_priv(dev), 0, sizeof( struct priv ) );
   dev->netdev_ops = &crypto_net_device_ops;
   for( j = 0; j < ETH_ALEN; ++j ) // fill in the MAC address with a phoney
      dev->dev_addr[ j ] = (char)j;
}

int __init init( void ) {
   int err = 0;
   struct priv *priv;
```

```
   char ifstr[ 40 ];
   sprintf( ifstr, "%s%s", ifname, "%d" );
#if (LINUX_VERSION_CODE < KERNEL_VERSION(3, 17, 0))
   child = alloc_netdev( sizeof( struct priv ), ifstr, setup );
#else
   child = alloc_netdev( sizeof( struct priv ), ifstr, NET_NAME_UNKNOWN, setup );
#endif
   if( child == NULL ) {
      ERR( "%s: allocate error", THIS_MODULE->name ); return -ENOMEM;
   }
   priv = netdev_priv( child );
   priv->parent = __dev_get_by_name( &init_net, link ); // parent interface
   if( !priv->parent ) {
      ERR( "%s: no such net: %s", THIS_MODULE->name, link );
      err = -ENODEV; goto err;
   }
   if( priv->parent->type != ARPHRD_ETHER && priv->parent->type != ARPHRD_LOOPBACK ) {
      ERR( "%s: illegal net type", THIS_MODULE->name );
      err = -EINVAL; goto err;
   }
   /* also, and clone its IP, MAC and other information */
   memcpy( child->dev_addr, priv->parent->dev_addr, ETH_ALEN );
   memcpy( child->broadcast, priv->parent->broadcast, ETH_ALEN );
   if( ( err = dev_alloc_name( child, child->name ) ) ) {
      ERR( "%s: allocate name, error %i", THIS_MODULE->name, err );
      err = -EIO; goto err;
   }
   register_netdev( child );
   rtnl_lock();
   netdev_rx_handler_register( priv->parent, &handle_frame, NULL );
   rtnl_unlock();
   LOG( "module %s loaded", THIS_MODULE->name );
   LOG( "%s: create link %s", THIS_MODULE->name, child->name );
   LOG( "%s: registered rx handler for %s", THIS_MODULE->name, priv->parent->name );
   return 0;
err:
   free_netdev( child );
   return err;
}

void __exit exit( void ) {
   struct priv *priv = netdev_priv( child );
   if( priv->parent ) {
      rtnl_lock();
      netdev_rx_handler_unregister( priv->parent );
      rtnl_unlock();
      LOG( "unregister rx handler for %s\n", priv->parent->name );
   }
   unregister_netdev( child );
   free_netdev( child );
   LOG( "module %s unloaded", THIS_MODULE->name );
}

module_init( init );
module_exit( exit );

MODULE_AUTHOR( "Oleg Tsiliuric" );
MODULE_AUTHOR( "Nikita Dorokhin" );
MODULE_LICENSE( "GPL v2" );
```

```
MODULE_VERSION( "2.1" );
```

Перехват **входящего** трафика родительского интерфейса здесь осуществляется установкой нового обработчика (функция `handle_frame()`) входящих пакетов для созданного интерфейса, вызовом `netdev_rx_handler_unregister()`, который появился в API ядра начиная с 2.6.36 (ранее это приходилось делать другими способами). При передаче **исходящего** сокетного буфера в сеть (функция `start_xmit()`) мы просто подменяем в структуре сокетного буфера интерфейс, через который физически должна производиться отправка.

Работа с таким интерфейсом выглядит примерно следующим образом:

- на **любой** существующий и работоспособный сетевой интерфейс:

```
$ ip addr show dev p7p1
3: p7p1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:9e:02:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global p7p1
    inet6 fe80::a00:27ff:fe9e:202/64 scope link
    valid_lft forever preferred_lft forever
```

- устанавливаем новый виртуальный интерфейс и конфигурируем его (на IP подсеть, отличную от исходной подсети интерфейса `p7p1`):

```
$ sudo insmod virt2.ko link=p7p1
$ sudo ifconfig virt0 192.168.50.2
$ ifconfig virt0
virt0     Link encap:Ethernet  HWaddr 08:00:27:9E:02:02
          inet addr:192.168.50.2  Bcast:192.168.50.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe9e:202/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:27 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:5027 (4.9 KiB)
```

- самый простой способ создать **ответный** конец (вам ведь нужно как-то тестировать свою работу?) для такой (192.168.50.2/24) подсети **на другом хосте** LAN, это создать **алиасный IP** для сетевого интерфейса этого удалённого хоста, по типу:

```
$ sudo ifconfig vboxnet0:1 192.168.50.1
$ ifconfig
...
vboxnet0  Link encap:Ethernet  HWaddr 0A:00:27:00:00:00
          inet addr:192.168.56.1  Bcast:192.168.56.255  Mask:255.255.255.0
          inet6 addr: fe80::800:27ff:fe00:0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:223 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:36730 (35.8 KiB)
vboxnet0:1 Link encap:Ethernet  HWaddr 0A:00:27:00:00:00
          inet addr:192.168.50.1  Bcast:192.168.50.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

(Здесь показан сетевой интерфейс гипервизора виртуальных машин VirtualBox, но точно то же можно проделать и с интерфейсом любого физического устройства).

- теперь из вновь созданного виртуального интерфейса мы можем проверить прозрачность сети посылкой ICMP:

```
$ ping 192.168.50.1
PING 192.168.50.1 (192.168.50.1) 56(84) bytes of data.
64 bytes from 192.168.50.1: icmp_req=1 ttl=64 time=0.371 ms
64 bytes from 192.168.50.1: icmp_req=2 ttl=64 time=0.210 ms
64 bytes from 192.168.50.1: icmp_req=3 ttl=64 time=0.184 ms
64 bytes from 192.168.50.1: icmp_req=4 ttl=64 time=0.242 ms
^C
```

```
--- 192.168.50.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 0.184/0.251/0.371/0.074 ms
```

- и далее создать для удалённого хоста сессию ssh (по протоколу TCP) через новый виртуальный интерфейс:

```
$ ssh 192.168.50.2
Nasty PTR record "192.168.50.2" is set up for 192.168.50.2, ignoring
olej@192.168.50.2's password:
Last login: Tue Apr  3 10:21:28 2012 from 192.168.1.5
[olej@fedora16vm ~]$ uname -a
Linux fedora16vm.localdomain 3.3.0-8.fc16.i686 #1 SMP Thu Mar 29 18:33:55 UTC 2012 i686 i686 i386
GNU/Linux
[olej@fedora16vm ~]$ exit
logout
Connection to 192.168.50.2 closed.
$
```

Примечание: Чтобы не увеличивать сложность обсуждения, выше показан упрощённый пример модуля, который полностью **перехватывает** трафик родительского интерфейса, при этом он **замещает** родительский интерфейс (для этого нужно раздельно обрабатывать пакеты IP и пакеты разрешения адресов ARP). Среди архива примеров размещён и архив (virt-full.tgz), который корректно анализирует адреса получателей.

Виртуальный сетевой интерфейс (созданный тем или иным способом) это очень мощный инструмент периода создания и отладки сетевых модулей, поэтому он заслуживает отдельного рассмотрения.

Протокол сетевого уровня

На этом уровне (L3) обеспечивается обработка таких протоколов, как: IP/IPv4/IPv6, IPX, ICMP, RIP, OSPF, ARP, или добавление оригинальных пользовательских протоколов. Для установки обработчиков сетевого уровня предоставляется API сетевого уровня (<linux/netdevice.h>):

```
struct packet_type {
   __be16  type; /* This is really htons(ether_type). */
   struct net_device *dev; /* NULL is wildcarded here    */
   int (*func) (struct sk_buff *, struct net_device *, struct packet_type *, struct net_device *);
...
   struct list_head list;
};
extern void dev_add_pack( struct packet_type *pt );
extern void dev_remove_pack( struct packet_type *pt );
```

Фактически, в протокольных модулях, как здесь, так и далее на транспортном уровне — мы должны **добавить фильтр**, через который проходят буфера сокетов **из входящего потока** интерфейса (исходящий поток реализуется проще, как показано в примерах ранее). Функция dev_add_pack() добавляет ещё один новый обработчик для пакетов заданного типа, реализуемый функцией func(). Функция **добавляет, но не замещает** существующий обработчик (в том числе и обработчик по умолчанию сетевой системы Linux). На обработку в функцию отбираются (попадают) те буфера сокетов, которые удовлетворяют критерием, заложенным в структуре struct packet_type (по типу протокола type, сетевому интерфейсу dev)

Примеры добавления собственных обработчиков сетевых протоколов находятся в архиве netproto.tgz. Вот так может быть добавлен обработчик нового протокола сетевого уровня:

net_proto.c :

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

```
#define ERR(...) printk( KERN_ERR "! "__VA_ARGS__ )
#define LOG(...) printk( KERN_INFO "! "__VA_ARGS__ )

int test_pack_rcv( struct sk_buff *skb, struct net_device *dev,
                   struct packet_type *pt, struct net_device *odev ) {
   LOG( "packet received with length: %u\n", skb->len );
   kfree_skb( skb );
   return skb->len;
};

#define TEST_PROTO_ID 0x1234
static struct packet_type test_proto = {
   __constant_htons( ETH_P_ALL ),  // may be: __constant_htons( TEST_PROTO_ID ),
   NULL,
   test_pack_rcv,
   (void*)1,
   NULL
};

static int __init my_init( void ) {
   dev_add_pack( &test_proto );
   LOG( "module loaded\n" );
   return 0;
}

static void __exit my_exit( void ) {
   dev_remove_pack( &test_proto );
   LOG( KERN_INFO "module unloaded\n" );
}

module_init( my_init );
module_exit( my_exit );

MODULE_AUTHOR( "Oleg Tsiliuric" );
MODULE_LICENSE( "GPL v2" );
```

Примечание: Самая большая сложность с подобными примерами — это то, какими средствами вы будете его тестировать для нестандартного протокола, когда операционная система, возможно, не знает такого сетевого протокола, и не имеет утилит обмена в таком протоколе...

Выполнение такого примера:

```
$ sudo insmod net_proto.ko
$ dmesg | tail -n6
module loaded
packet received with length: 74
packet received with length: 60
packet received with length: 66
packet received with length: 241
packet received with length: 52
$ sudo rmmod net_proto
```

В этом примере обработчик протокола перехватывает (фильтрует) **все** пакеты (константа `ETH_P_ALL`) на всех сетевых интерфейсах. В случае собственного протокола здесь должна бы быть константа `TEST_PROTO_ID` (но для такого случая нам нечем оттестировать модуль). Очень большое число идентификаторов протоколов (Ethernet Protocol ID's) находим в `<linux/if_ether.h>`, некоторые наиболее интересные из них, для примера:

```
#define ETH_P_LOOP   0x0060  /* Ethernet Loopback packet  */
...
#define ETH_P_IP     0x0800  /* Internet Protocol packet  */
```

```
...
#define ETH_P_ARP    0x0806  /* Address Resolution packet */
...
#define ETH_P_PAE    0x888E  /* Port Access Entity (IEEE 802.1X) */
...
#define ETH_P_ALL    0x0003  /* Every packet (be careful!!!) */
...
```

Здесь же находим описание заголовка Ethernet пакета, который помогает в заполнении структуры `struct packet_type` :

```
struct ethhdr {
   unsigned char h_dest[ETH_ALEN];   /* destination eth addr */
   unsigned char h_source[ETH_ALEN]; /* source ether addr    */
   __be16       h_proto;             /* packet type ID field */
} __attribute__((packed));
```

Этот пример порождает ряд вопросов:

- Можно ли установить несколько обработчиков потока пакетов (для одного или различающихся типов протоколов `type`)?
- В каком порядке будут вызываться функции-обработчики при поступлении пакета?
- Как специфицировать один отдельный сетевой интерфейс, к которому должен применяться фильтр?
- Какую роль играет вызов `kfree_skb()` (в функции-фильтре обработки протокола `test_pack_rcv()`), и какой сокетный буфер (или его копия) при этом освобождается?

Для уяснения этих вопросов нам необходимо собрать на базе предыдущего другой пример (показаны только отличающиеся фрагменты кода):

net_proto2.c :

```
...
static int debug = 0;
module_param( debug, int, 0 );

static char* link = NULL;
module_param( link, charp, 0 );

int test_pack_rcv_1( struct sk_buff *skb, struct net_device *dev,
                     struct packet_type *pt, struct net_device *odev ) {
   int s = atomic_read( &skb->users );
   kfree_skb( skb );
   if( debug > 0 )
      LOG( "function #1 - %p => users: %d->%d\n", skb, s, atomic_read( &skb->users ) );
   return skb->len;
};

int test_pack_rcv_2( struct sk_buff *skb, struct net_device *dev,
                     struct packet_type *pt, struct net_device *odev ) {
   int s = atomic_read( &skb->users );
   kfree_skb( skb );
   if( debug > 0 )
      LOG( "function #2 - %p => users: %d->%d\n", skb, s, atomic_read( &skb->users ) );
   return skb->len;
};

static struct packet_type
test_proto1 = {
   __constant_htons( ETH_P_IP ),
   NULL,
```

```
    test_pack_rcv_1,
    (void*)1,
    NULL
},
test_proto2 = {
    __constant_htons( ETH_P_IP ),
    NULL,
    test_pack_rcv_2,
    (void*)1,
    NULL
};

static int __init my_init( void ) {
    if( link != NULL ) {
        struct net_device *dev = __dev_get_by_name( &init_net, link );
        if( NULL == dev ) {
            ERR( "%s: illegal link", link );
            return -EINVAL;
        }
        test_proto1.dev = test_proto2.dev = dev;
    }
    dev_add_pack( &test_proto1 );
    dev_add_pack( &test_proto2 );
    if( NULL == test_proto1.dev ) LOG( "module %s loaded for all links\n", THIS_MODULE->name );
    else LOG( "module %s loaded for link %s\n", THIS_MODULE->name, link );
    return 0;
}

static void __exit my_exit( void ) {
    if( test_proto2.dev != NULL ) dev_put( test_proto2.dev );
    if( test_proto1.dev != NULL ) dev_put( test_proto1.dev );
    dev_remove_pack( &test_proto2 );
    dev_remove_pack( &test_proto1 );
    LOG( "module %s unloaded\n", THIS_MODULE->name );
}
```

Здесь, собственно, выполняется абсолютно то же, что и в предыдущем варианте, но мы устанавливаем одновременно **два** новых дополнительных обработчика для пакетов IPv4 (`ETH_P_IP`), причём устанавливаем именно в последовательности: `test_pack_rcv_1()`, а затем `test_pack_rcv_2()`. Смотрим что из этого получается... Загружаем модуль:

```
$ sudo insmod net_proto2.ko link=p7p1
$ dmesg | tail -n1
[  403.339591] ! module net_proto2 loaded for link p7p1
```

И далее (конфигурировав интерфейс на адрес 192.168.56.3) с другого хоста выполняем:

```
$ ping -c3 192.168.56.3
PING 192.168.56.3 (192.168.56.3) 56(84) bytes of data.
64 bytes from 192.168.56.3: icmp_req=1 ttl=64 time=0.668 ms
64 bytes from 192.168.56.3: icmp_req=2 ttl=64 time=0.402 ms
64 bytes from 192.168.56.3: icmp_req=3 ttl=64 time=0.330 ms
--- 192.168.56.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.330/0.466/0.668/0.147 ms
```

В системном журнале мы найдём:

```
$ dmesg | tail -n7
[  403.339591] ! module net_proto2 loaded for link p7p1
[  420.305824] ! function #2 - eb6873c0 => users: 2->1
[  420.305829] ! function #1 - eb6873c0 => users: 2->1
[  421.306302] ! function #2 - eb687c00 => users: 2->1
```

```
[  421.306308] ! function #1 - eb687c00 => users: 2->1
[  422.306289] ! function #2 - eb687180 => users: 2->1
[  422.306294] ! function #1 - eb687180 => users: 2->1
```

Отсюда видно, что обработчики срабатывают в порядке обратном их установке (установленный позже - срабатывает раньше), но срабатывают **все**. Они получают в качестве аргумента адрес одной и той же копии буфера сокета. Относительно `kfree_skb()` мы должны обратиться к исходному коду реализации ядра (файл `net/core/skbuff.c`):

```
void kfree_skb(struct sk_buff *skb) {
   if (unlikely(!skb))
      return;
   if (likely(atomic_read(&skb->users) == 1))
      smp_rmb();
   else if (likely(!atomic_dec_and_test(&skb->users)))
      return;
   trace_kfree_skb(skb, __builtin_return_address(0));
   __kfree_skb(skb);
}
```

Вызов `kfree_skb()` будет реально освобождать буфер сокета только в случае `skb->users == 1`, при всех остальных значениях он будет только декрементировать `skb->users` (счётчик использования). Для уточнения происходящего мы можем закомментировать вызовы `kfree_skb()` в двух обработчиках выше, и наблюдать при этом:

```
$ dmesg | tail -n7
[11373.754524] ! module net_proto2 loaded for link p7p1
[11398.930057] ! function #2 - ed3dfc00 => users: 2->2
[11398.930061] ! function #1 - ed3dfc00 => users: 3->3
[11399.929838] ! function #2 - ed3dfb40 => users: 2->2
[11399.929843] ! function #1 - ed3dfb40 => users: 3->3
[11400.929522] ! function #2 - ed3df480 => users: 2->2
[11400.929527] ! function #1 - ed3df480 => users: 3->3
```

Если функция обработчик не декрементирует `skb->users` по завершению вызовом `kfree_skb()`, то, после окончательного вызова обработки по умолчанию, буфер сокета не будет уничтожен, и в системе будет наблюдаться **утечка памяти**. Она незначительная, но неумолимая, и, в конце концов, приведёт к краху системы. Проверку на отсутствие утечки памяти (по этой причине, или по любой иной) предлагается проверить приёмом по сетевому каналу значительного объёма данных (несколько гигабайт), с фиксацией состояния памяти командой `free`. Для этого можно с успехом использовать утилиту `nc` (network cat):

- на тестируемом узле запустить скрипт `./client`:

```
LIMIT=1000000
for ((a=1; a <= LIMIT ; a++))
do
   rm -a z.txt
   nc 192.168.56.1 12345 > z.txt
   sleep 1
done
```

- на стороннем узле сети запустить скрипт `./server`

```
dd if=/dev/zero of=z.txt bs=1M count=1
LIMIT=1000000
for ((a=1; a <= LIMIT ; a++))
do
   cat z.txt | nc -l 12345
done
```

(здесь в скриптах: 192.168.56.1 — IP адрес узла где работает `server`, 12345 — номер TCP порта, через который `nc` предписывается передавать данные, при желании, можно использовать и UDP, добавив к `nc` опцию `-u`)

- и подождать некоторое время, порядка 10-20 минут:

```
$ free
             total       used       free     shared    buffers     cached
```

```
Mem:        768084     260636     507448          0      23392     129108
-/+ buffers/cache:     108136     659948
Swap:      1540092          0    1540092

$ ifconfig p7p1
p7p1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.56.3  netmask 255.255.255.255  broadcast 192.168.56.3
        inet6 fe80::a00:27ff:fe08:9abd  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:08:9a:bd  txqueuelen 1000  (Ethernet)
        RX packets 2017560  bytes 3048762624 (2.8 GiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 398156  bytes 26283474 (25.0 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Ещё раз о виртуальном интерфейсе

Ранее рассматривалось создание виртуального сетевого интерфейса (использующего трафик реального физического, родительского) средствами `netdev_rx_handler_register()`. Но такой способ не свободен от некоторых недостатков: он появился хронологически достаточно поздно, в API ядра начиная с версии 2.6.36, не со всяким ядром он может быть использован, кроме того, он в некоторой степени громоздкий и путанный. Но это же можно сделать и другим способом, не зависящим от версий используемого ядра, используя протокольные механизмы сетевого уровня (L3). Кроме того, это хороший реалистичный пример использования протокольных фильтров. Пример подобной реализации расположен в архиве `virt-proto.tgz`.

Примечание: В названном архиве представлены два варианта модуля: упрощённый модуль `virtl.ko` (lite вариант), интерфейс (`virt0`) которого **замещает** родительский сетевой интерфейс, и полный вариант `virt.ko`, который анализирует сетевые фреймы (и ARP и IP4 протоколов), и затрагивает только трафик, к его интерфейсу относящийся. Разница состоит в том, что на время загрузки упрощённого модуля работа родительского интерфейса прекращается, а при загрузке полного варианта оба интерфейса работают независимо. Но код полного модуля гораздо более громоздкий, а для понимания принципов он ничего не добавляет. Ниже детально рассмотрен упрощённый вариант, не скрывающий принципы, и только позже мы в пару слов коснёмся полного варианта: код его и протокол испытаний приведены в архиве, поэтому его детализация не вызывает сложностей.

virtl.c :

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/inetdevice.h>
#include <linux/moduleparam.h>
#include <net/arp.h>
#include <linux/ip.h>

#define ERR(...) printk( KERN_ERR "! "__VA_ARGS__ )
#define LOG(...) printk( KERN_INFO "! "__VA_ARGS__ )
#define DBG(...) if( debug != 0 ) printk( KERN_INFO "! "__VA_ARGS__ )

static char* link = "eth0";
module_param( link, charp, 0 );

static char* ifname = "virt";
module_param( ifname, charp, 0 );

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

static struct net_device *child = NULL;
static struct net_device_stats stats;  // статическая таблица статистики интерфейса
```

```
static u32 child_ip;

struct priv {
   struct net_device *parent;
};

static char* strIP( u32 addr ) {      // диагностика IP в точечной нотации
   static char saddr[ MAX_ADDR_LEN ];
   sprintf( saddr, "%d.%d.%d.%d",
            ( addr ) & 0xFF, ( addr >> 8 ) & 0xFF,
            ( addr >> 16 ) & 0xFF, ( addr >> 24 ) & 0xFF
          );
   return saddr;
}

static int open( struct net_device *dev ) {
   struct in_device *in_dev = dev->ip_ptr;
   struct in_ifaddr *ifa = in_dev->ifa_list;      /* IP ifaddr chain */
   LOG( "%s: device opened", dev->name );
   child_ip = ifa->ifa_address;
   netif_start_queue( dev );
   if( debug != 0 ) {
      char sdebg[ 40 ] = "";
      sprintf( sdebg, "%s:", strIP( ifa->ifa_address ) );
      strcat( sdebg, strIP( ifa->ifa_mask ) );
      DBG( "%s: %s", dev->name, sdebg );
   }
   return 0;
}

static int stop( struct net_device *dev ) {
   LOG( "%s: device closed", dev->name );
   netif_stop_queue( dev );
   return 0;
}

static struct net_device_stats *get_stats( struct net_device *dev ) {
   return &stats;
}

// передача фрейма
static netdev_tx_t start_xmit( struct sk_buff *skb, struct net_device *dev ) {
   struct priv *priv = netdev_priv( dev );
   stats.tx_packets++;
   stats.tx_bytes += skb->len;
   skb->dev = priv->parent;   // передача в родительский (физический) интерфейс
   skb->priority = 1;
   dev_queue_xmit( skb );
   DBG( "tx: injecting frame from %s to %s with length: %u",
        dev->name, skb->dev->name, skb->len );
   return 0;
}

static struct net_device_ops net_device_ops = {
   .ndo_open = open,
   .ndo_stop = stop,
   .ndo_get_stats = get_stats,
   .ndo_start_xmit = start_xmit,
};
```

```
// приём фрейма
int pack_parent( struct sk_buff *skb, struct net_device *dev,
                 struct packet_type *pt, struct net_device *odev ) {
   skb->dev = child;          // передача фрейма в виртуальный интерфейс
   stats.rx_packets++;
   stats.rx_bytes += skb->len;
   DBG( "tx: injecting frame from %s to %s with length: %u",
        dev->name, skb->dev->name, skb->len );
   kfree_skb( skb );
   return skb->len;
};

static struct packet_type proto_parent = {
   __constant_htons( ETH_P_ALL ), // перехватывать все пакеты: ETH_P_ARP & ETH_P_IP
   NULL,
   pack_parent,
   (void*)1,
   NULL
};

int __init init( void ) {
   void setup( struct net_device *dev ) { // вложенная функция (GCC)
      int j;
      ether_setup( dev );
      memset( netdev_priv( dev ), 0, sizeof( struct priv ) );
      dev->netdev_ops = &net_device_ops;
      for( j = 0; j < ETH_ALEN; ++j )     // заполнить MAC фиктивным адресом
         dev->dev_addr[ j ] = (char)j;
   }
   int err = 0;
   struct priv *priv;
   char ifstr[ 40 ];
   sprintf( ifstr, "%s%s", ifname, "%d" );
#if (LINUX_VERSION_CODE < KERNEL_VERSION(3, 17, 0))
   child = alloc_netdev( sizeof( struct priv ), ifstr, setup );
#else
   child = alloc_netdev( sizeof( struct priv ), ifstr, NET_NAME_UNKNOWN, setup );
#endif
   if( child == NULL ) {
      ERR( "%s: allocate error", THIS_MODULE->name ); return -ENOMEM;
   }
   priv = netdev_priv( child );
   priv->parent = __dev_get_by_name( &init_net, link ); // родительский интерфейс
   if( !priv->parent ) {
      ERR( "%s: no such net: %s", THIS_MODULE->name, link );
      err = -ENODEV; goto err;
   }
   if( priv->parent->type != ARPHRD_ETHER && priv->parent->type != ARPHRD_LOOPBACK ) {
      ERR( "%s: illegal net type", THIS_MODULE->name );
      err = -EINVAL; goto err;
   }
   memcpy( child->dev_addr, priv->parent->dev_addr, ETH_ALEN );
   memcpy( child->broadcast, priv->parent->broadcast, ETH_ALEN );
   if( ( err = dev_alloc_name( child, child->name ) ) ) {
      ERR( "%s: allocate name, error %i", THIS_MODULE->name, err );
      err = -EIO; goto err;
   }
   register_netdev( child );         // зарегистрировать новый интерфейс
```

```
    proto_parent.dev = priv->parent;
    dev_add_pack( &proto_parent );    // установить обработчик фреймов для родителя
    LOG( "module %s loaded", THIS_MODULE->name );
    LOG( "%s: create link %s", THIS_MODULE->name, child->name );
    return 0;
err:
    free_netdev( child );
    return err;
}

void __exit exit( void ) {
    dev_remove_pack( &proto_parent ); // удалить обработчик фреймов
    unregister_netdev( child );
    free_netdev( child );
    LOG( "module %s unloaded", THIS_MODULE->name );
    LOG( "============================================" );
}

module_init( init );
module_exit( exit );

MODULE_AUTHOR( "Oleg Tsiliuric" );
MODULE_LICENSE( "GPL v2" );
MODULE_VERSION( "3.6" );
```

От рассмотренных уже ранее примеров код отличается только тем, что после регистрации нового сетевого интерфейса (`virt0`) он выполняет вызов `dev_add_pack()`, предварительно установив в структуре `packet_type` поле `dev` на указатель родительского интерфейса: только с этого интерфейса входящий трафик будет перехватываться определённой в структуре функцией `pack_parent()`. Эта функция фиксирует статистику интерфейса и, самое главное, **подменяет** в сокетном буфере указатель родительского интерфейса на виртуальный. Обратная подмена (виртуального на физический) происходит в функции отправки фрейма `start_xmit()`, но это не отличается от того, что мы видели ранее. Вот как это работает:

- на тестируемом компьютере загружаем модуль и конфигурируем его:

```
$ ip address
...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:52:b9:e0 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.21/24 brd 192.168.1.255 scope global eth0
    inet6 fe80::a00:27ff:fe52:b9e0/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:0f:13:6d brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.102/24 brd 192.168.56.255 scope global eth1
    inet6 fe80::a00:27ff:fe0f:136d/64 scope link
       valid_lft forever preferred_lft forever
$ sudo insmod ./virt.ko link=eth1 debug=1
$ sudo ifconfig virt0 192.168.50.19
$ sudo ifconfig virt0
virt0     Link encap:Ethernet  HWaddr 08:00:27:0f:13:6d
          inet addr:192.168.50.19  Bcast:192.168.50.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe0f:136d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:46 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:8373 (8.1 KiB)
```

(показана статистика с нулевым числом принятых байт на интерфейсе).

- на тестирующем компьютере создаём алиасный IP для тестируемой подсети (192.168.50.0/24) и можем осуществлять трафик на созданный интерфейс:

```
$ sudo ifconfig vboxnet0:1 192.168.50.1
$ ping 192.168.50.19
PING 192.168.50.19 (192.168.50.19) 56(84) bytes of data.
64 bytes from 192.168.50.19: icmp_req=1 ttl=64 time=0.627 ms
64 bytes from 192.168.50.19: icmp_req=2 ttl=64 time=0.305 ms
64 bytes from 192.168.50.19: icmp_req=3 ttl=64 time=0.326 ms
^C
--- 192.168.50.19 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.305/0.419/0.627/0.148 ms
```

- на этом же (тестирующем) компьютере (ответной стороне) очень содержательно наблюдать трафик (в отдельном терминале), фиксируемый `tcpdump`:

```
$ sudo tcpdump -i vboxnet0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on vboxnet0, link-type EN10MB (Ethernet), capture size 65535 bytes
...
18:41:01.740607 ARP, Request who-has 192.168.50.19 tell 192.168.50.1, length 28
18:41:01.741104 ARP, Reply 192.168.50.19 is-at 08:00:27:0f:13:6d (oui Unknown), length 28
18:41:01.741116 IP 192.168.50.1 > 192.168.50.19: ICMP echo request, id 8402, seq 1, length 64
18:41:01.741211 IP 192.168.50.19 > 192.168.50.1: ICMP echo reply, id 8402, seq 1, length 64
18:41:02.741164 IP 192.168.50.1 > 192.168.50.19: ICMP echo request, id 8402, seq 2, length 64
18:41:02.741451 IP 192.168.50.19 > 192.168.50.1: ICMP echo reply, id 8402, seq 2, length 64
18:41:03.741163 IP 192.168.50.1 > 192.168.50.19: ICMP echo request, id 8402, seq 3, length 64
18:41:03.741471 IP 192.168.50.19 > 192.168.50.1: ICMP echo reply, id 8402, seq 3, length 64
18:41:06.747701 ARP, Request who-has 192.168.50.1 tell 192.168.50.19, length 28
18:41:06.747715 ARP, Reply 192.168.50.1 is-at 0a:00:27:00:00:00 (oui Unknown), length 28
```

Теперь коротко, в два слова, о том, как сделать полновесный виртуальный интерфейс, работающий только со своим трафиком, и не нарушающий работу родительского интерфейса (то, что делает полная версия модуля в архиве). Для этого необходимо:

- объявить **два** раздельных обработчика протоколов (для протоколов разрешения имён ARP и собственно для протокола IP):

```
// обработчик фреймов ETH_P_ARP
int arp_pack_rcv( struct sk_buff *skb, struct net_device *dev,
                  struct packet_type *pt, struct net_device *odev ) {
   ...
   return skb->len;
};

static struct packet_type arp_proto = {
   __constant_htons( ETH_P_ARP ),
   NULL,
   arp_pack_rcv,  // фильтр пртокола ETH_P_ARP
   (void*)1,
   NULL
};

// обработчик фреймов ETH_P_IP
int ip4_pack_rcv( struct sk_buff *skb, struct net_device *dev,
                  struct packet_type *pt, struct net_device *odev ) {
```

```
    ...
    return skb->len;
};

static struct packet_type ip4_proto = {
    __constant_htons( ETH_P_IP ),
    NULL,
    ip4_pack_rcv,    // фильтр пртокола ETH_P_IP
    (void*)1,
    NULL
};
```

- и оба их зарегистрировать в функции инициализации модуля:

```
    arp_proto.dev = ip4_proto.dev = priv->parent; // перехват только с родительского интерфейса
    dev_add_pack( &arp_proto );
    dev_add_pack( &ip4_proto );
```

- каждый из обработчиков должен осуществлять **подмену** интерфейса только для тех фреймов, IP получателя которых совпадает с IP интерфейса ...

- а два раздельных обработчика удобны тем, что заголовки фреймов ARP и IP имеют совершенно разный формат, и выделять IP назначения в них приходится по-разному (весь полный код показан в архиве примера).

Используя такой полновесный модуль, можно открыть к хосту, например, две параллельные сессии SSH на разные интерфейсы (использующие разные IP), которые будут в параллель реально использовать единый общий физический интерфейс:

```
$ ssh olej@192.168.50.17
olej@192.168.50.17's password:
Last login: Mon Jul 16 15:52:16 2012 from 192.168.1.9
...
$ ssh olej@192.168.56.101
olej@192.168.56.101's password:
Last login: Mon Jul 16 17:29:57 2012 from 192.168.50.1
...
$ who
olej     tty1         2012-07-16 09:29 (:0)
olej     pts/0        2012-07-16 09:33 (:0.0)
olej     pts/1        2012-07-16 12:22 (192.168.1.9)
olej     pts/4        2012-07-16 15:52 (192.168.1.9)
olej     pts/6        2012-07-16 17:29 (192.168.50.1)
olej     pts/7        2012-07-16 17:31 (192.168.56.1)
```

Последняя показанная команда (`who`) выполняется уже в сессии SSH, то есть на том самом удалённом хосте, к которому и фиксируется два **независимых** подключения из двух различных подсетей (последние две строки вывода), которые на самом деле представляют один хост.

Протокол транспортного уровня

На этом уровне (L4) обеспечивается обработка таких протоколов, как: UDP, TCP, SCTP, DCCP ... их число возрастает. Протоколы транспортного уровня (протоколы IP) описаны в `<linux/in.h>` :

```
/* Standard well-defined IP protocols.  */
enum {
  IPPROTO_IP =     0,   /* Dummy protocol for TCP               */
```

```
  IPPROTO_ICMP =   1,   /* Internet Control Message Protocol    */
  IPPROTO_IGMP =   2,   /* Internet Group Management Protocol   */
...
  IPPROTO_TCP =    6,   /* Transmission Control Protocol        */
...
  IPPROTO_UDP =   17,   /* User Datagram Protocol               */
...
  IPPROTO_SCTP = 132,   /* Stream Control Transport Protocol    */
...
  IPPROTO_RAW  = 255,   /* Raw IP packets                       */
}
```

Для установки обработчика протоколов транспортного уровня существует API `<net/protocol.h>` :

```
struct net_protocol {                 // This is used to register protocols
   int  (*handler)( struct sk_buff *skb );
   void (*err_handler)( struct sk_buff *skb, u32 info );
   int  (*gso_send_check)( struct sk_buff *skb );
   struct sk_buff *(*gso_segment)( struct sk_buff *skb, int features );
   struct sk_buff **(*gro_receive)( struct sk_buff **head, struct sk_buff *skb );
   int (*gro_complete)( struct sk_buff *skb );
   unsigned int no_policy:1,
                netns_ok:1;
};
int inet_add_protocol( const struct net_protocol *prot, unsigned char num );
int inet_del_protocol( const struct net_protocol *prot, unsigned char num );
```

Здесь 2-й параметр вызова функций (`num`) как раз и есть константа из числа `IPPROTO_*`.

Эта схема в общих чертах напоминает то, как это же делалось на сетевом уровне: каждый пакет проходит через функцию-фильтр, где мы можем анализировать или изменять отдельные поля сокетного буфера, отображающего пакет.

Пример модуля, устанавливающего протокол:

trn_proto.c :

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

int test_proto_rcv( struct sk_buff *skb ) {
   printk( KERN_INFO "Packet received with length: %u\n", skb->len );
   return skb->len;
};

static struct net_protocol test_proto = {
   .handler = test_proto_rcv,
   .err_handler = 0,
   .no_policy = 0,
};

//#define PROTO IPPROTO_ICMP
//#define PROTO IPPROTO_TCP
#define PROTO IPPROTO_RAW
static int __init my_init( void ) {
   int ret;
   if( ( ret = inet_add_protocol( &test_proto, PROTO ) ) < 0 ) {
      printk( KERN_INFO "proto init: can't add protocol\n");
      return ret;
   };
```

```
    printk( KERN_INFO "proto module loaded\n" );
    return 0;
}

static void __exit my_exit( void ) {
    inet_del_protocol( &test_proto, PROTO );
    printk( KERN_INFO "proto module unloaded\n" );
}

module_init( my_init );
module_exit( my_exit );

MODULE_AUTHOR( "Oleg Tsiliuric" );
MODULE_LICENSE( "GPL v2" );
```

Вот как будет выглядеть работа модуля для протокола `IPPROTO_RAW`:

```
$ sudo insmod trn_proto.ko
$ lsmod | head  -n2
Module                  Size  Used by
trn_proto               780  0
$ cat /proc/modules | grep proto
trn_proto 780 0 - Live 0xf9a26000
$ ls -R /sys/module/trn_proto
/sys/module/trn_proto:
holders  initstate  notes  refcnt  sections  srcversion
...
$ sudo rmmod trn_proto
$ dmesg | tail -n60 | grep -v ^audit
proto module loaded
proto module unloaded
```

Но если вы попытаетесь установить (добавить!) обработчик для уже **обрабатываемого** (установленного) протокола (например, `IPPROTO_TPC`), то получите ошибку:

```
$ sudo insmod trn_proto.ko
insmod: error inserting 'trn_proto.ko': -1 Operation not permitted
$ dmesg | tail -n60 | grep -v ^audit
proto init: can't add protocol
$ lsmod | grep proto
$
```

Здесь возникает уже названная раньше сложность:

- Если вы планируете отрабатывать новый (или не использующийся в системе) протокол, то для его тестирования в системе нет инструментов, и прежде нужно подумать о том, чтобы создать тестовые приложения прикладного уровня.

- Если пытаться моделировать работу нового протокола под видом уже существующего (например, `IPPROTO_UDP`), то вам прежде понадобится удалить существующий обработчик, чем можно радикально нарушить работоспособность системы (например, для `IPPROTO_UDP` разрушить систему разрешения доменных имён DNS).

Обсуждение

В порядке напоминания, хотелось бы посоветовать взять на заметку, что успех развития, отладки и тестирования сетевых модулей зависит не только от качественно прописанного кода и тщательно продуманного

плана тестирования. Важным элементом успеха становится грамотное управление роутингом в сетевой системе и добавление соответствующих записей в таблице маршрутизации. Таблица роутинга, демонстрируется вот такой командой:

```
$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.0     *               255.255.255.0   U     2      0        0 wlan0
default         192.168.1.1     0.0.0.0         UG    0      0        0 wlan0
```

До тех пор, пока в этой таблице не появится строка (строки) определяющие поведение нового интерфейса, над которым вы работаете, никакого положительного результата в поведении модуля добиться невозможно. А появиться эта строка может только если её добавить туда вручную той же командой `route`. Позже подобные действия могут выполняться синхронно с инсталляцией модуля при установке пакета. Но на этапе отработки гибкое управление роутингом становится залогом успеха. Эта особенность существенно отличает отработку сетевых модулей ядра от драйверов потоковых устройств в `/dev`, о которых мы говорили ранее.

И, конечно же, при отработке сетевых модулей ядра незаменимым инструментом становится такая утилита анализа трафика как `tcpdump`.

Замечание относительно технологии отладки (отработки) кода сетевых модулей:

- Отлаживать сетевые модули ядра, по крайней мере на начальных этапах разработки, в высшей степени, как никакой иной класс драйверов, плодотворно отрабатывать в виртуальной машине (например под управлением Virtual Box) ...

- И производить это с терминала (одного или многих одновременно) удалённого хоста, подключённого к отлаживаемой виртуальной машине сессиями SSH.

- Объясняются это удобство и продуктивность тем обстоятельством, что на виртуальной машине можно создать много (с запасом) сетевых интерфейсов разного свойства, таких как виртуальный сетевой адаптер, виртуальный интерфейс хоста и т.п.

- При этом SSH отработку (редактирование кода, компиляцию, тестирование, отладку, ...) разрабатываемого модуля производим **не через тот интерфейс**, для которого тестируем модуль (здесь могут быть самые разнообразные варианты: тестируем на реальном адаптере — связываемся через виртуальный, или напротив, тестируем на виртуальном адаптере — связываемся через реальный, и другие подобные варианты).

- При такой схеме, при «падении» тестируемого интерфейса (что происходит неизбежно и многократно) не разрывается связь и управление по SSH-интерфейсу, и мы можем легко перезапустить повреждённую конфигурацию без перезагрузки хоста и сети, что недостижимо при испытаниях на реальном оборудовании.

Использование такого стенда для отработки сетевых модулей показало снижение затрат времени на отработку **в разы**, что радикально ускоряет продвижение проекта.

Задачи

1. Сделайте модуль виртуального сетевого интерфейса, надстроив его над вашим реальным сетевым интерфейсом. Протестируйте сквозь него трафик. Модифицируйте (шифруйте, например по XOR) «на лету» его трафик.

2. Реализуйте виртуальный интерфейс подобно предыдущей задаче, но перенаправление в него трафика организуйте на сетевом уровне (на уровне протоколов).

3. (*) Попытайтесь реализовать свой собственный протокол транспортного уровня. В чём здесь состоит трудность?

4. Почему отработку модулей, работающих с сетевым стеком, **намного продуктивнее** отрабатывать именно в среде виртуальной машины (VirtualBox)?

9. Обработка прерываний

«Трудное – это то, что может быть сделано немедленно; невозможное – то, что потребует немного больше времени.»
Сантаяна.

Мы закончили рассмотрение механизмов параллелизмов, для случаев, когда это действительно параллельно выполняющиеся фрагменты кода (в случае SMP и наличия нескольких процессоров), или когда это квази-параллельность, и различные ветви асинхронно вытесняют друг друга, занимая единый процессор. Глядя на сложности, порождаемые параллельными вычислениями, можно было бы попытаться и вообще отказаться от параллельных механизмов в угоду простоте и детерминированности последовательного вычислительного процесса. И так и стараются поступить часто в малых и встраиваемых архитектурах. Можно было бы ..., если бы не один вид естественного асинхронного параллелизма, который возникает в любой, даже самой простой и однозадачной операционной системе, такой, например, как MS-DOS, и это — аппаратные прерывания. И наличие такого одного механизма сводит на нет попытки представить реальный вычислительный процесс как чисто последовательностный, как принято в сугубо теоретическом рассмотрении: параллелизм присутствует всегда!

Обмен с периферийным оборудованием, по существу, может происходить **только двумя** способами: циклическим программным опросом (пулингом) и по прерываниям от периферийного оборудования.

Примечание: Есть одна область практических применений средств компьютерной индустрии, которая развивается совершенно автономно, и в которой попытались уйти от асинхронного обслуживания аппаратных прерываний, относя именно к наличию этих механизмов риски отказов, снижения надёжности и живучести систем (утверждение, которое само по себе вызывает изрядные сомнения, или, по крайней мере, требующее доказательств, которые на сегодня не представлены). И область эта: промышленные программируемые логические контроллеры (PLC), применяемые в построении систем АСУ ТП экстремальной надёжности. Такие PLC строятся на абсолютно тех же процессорах общего применения, но обменивающиеся с многочисленной периферией не по прерываниям, а методами циклического программного опроса (пулинга), часто с периодом опроса миллисекундного диапазона или даже ниже. Не взирая на некоторую обособленность этой ветви развития, она занимает (в финансовых объёмах) весьма существенную часть компьютерной индустрии, где преуспели такие мировые бренды как: Modicon (ныне Schneider Electric), Siemens, Allen-Bradley и ряд других. Примечательно, что целый ряд известных моделей PLC работают, в том числе, и под операционной системой Linux, но работа с данными в них основывается на совершенно иных принципах, что, собственно, и делает их PLC. Вся эта отрасль стоит особняком, и к её особенностям мы не будем больше обращаться.

Могут быть комбинированное использование способов, остроумное решение в этом направлении являет собой сетевая подсистема Linux: в периоды «затишья» сетевой активности подсистема ожидает прихода сетевых пакетов по прерыванию. Но после получения первого пакета серии, ожидая всплеск активнсти, сетевая подсистема переходит в режим пулинга, циклически опрашивая сетевой адаптер. Как только сетевая активность затихает, пулинг прекращается, и подсистема возвращается в режим ожидания прерывания.

Общая модель обработки прерывания

Схема обработки аппаратных прерываний — это принципиально архитектурно зависимое действие, связанное с непосредственным взаимодействием с контроллером прерываний. Но схема в основных чертах остаётся неизменной, независимо от архитектуры. Вот как она выглядела, к примеру, в системе MS-DOS для процессоров x86 и «старого» контроллера прерываний (чип 8259) - на уровне ассемблера это нечто подобное последовательности действий:

- После возникновения аппаратного прерывания управление асинхронно получает функция (ваша функция!), адрес которой записан в векторе (вентиле) прерывания.
- Обработку прерывания функция обработчика выполняет при запрещённых следующих прерываниях.
- После завершения обработки прерывания функция-обработчик восстанавливает контроллер прерываний (чип 8259), посылая сигнал о завершении прерывания. Это осуществляется отправкой команды EOI (End Of Interrupt — код 20h — конец прерывания) в командный регистр микросхемы

8259. Это однобайтовый регистр адресуется через порт ввода/вывода 20h.

– Функция-обработчик завершается, возвращая управление командой `iret` (не `ret`, как все прочие привычные нам функции, вызываемые синхронно!).

Показанная схема слишком неэффективна, ненадёжна (в отношении пропущенных прерываний), и просто не может быть распространена на многозадачное окружение и на SMP архитектуры, использующие более современные чипы APIC контроллера прерываний (принципиально отличающиеся работой). В Linux обработка прерывания разделяется на две последовательные фазы и трансфомируется в следующую схему:

– Для линии IRQ регистрируется функция обработчика «верхней половины» (это та же ISR функция по смыслу, «верхняя половина» обработчика), который выполняется **при запрещённых прерываниях локального** процессора. Именно этой функции передаётся управление при возникновении аппаратного прерывания.

– Синтаксически функция-обработчик должна иметь строго описанный функциональный тип `irq_handler_t`, и возвращает управление **ядру системы** традиционным `return`, с возвращаемым значением `IRQ_NONE` или `IRQ_HANDLED`.

– При возникновении аппаратного прерывания по линии IRQ функция-обработчик получит управление. Эта функция выполняется **в контексте прерывания** — это одно из самых важных ограничений, накладываемых Linux, мы не раз будем возвращаться к нему. Перед своим завершением функция-обработчик **регистрирует** для последующего выполнения функцию нижней половины обработчика, которая и завершит позже начатую работу по обработке этого прерывания...

– В этой точке (после `return` из обработчика верхней половины) ядро завершает всё взаимодействие с аппаратурой контроллера прерываний, разрешает последующие прерывания, восстанавливает контроллер командой завершения обработки прерывания (посылает EOI) и возвращает управление из прерывания (из ядра!) уже именно командой `iret`. После этого будет восстановлен **контекст прерванного** процесса (потока).

– А вот запланированная выше к выполнению функция нижней половины будет вызвана ядром в некоторый момент позже (часто это может быть и непосредственно после завершения `return` из верхней половины, но это непредсказуемо), тогда, когда удобнее будет ядру системы. Принципиально важное отличие функции нижней половины состоит в том, что она выполняется уже **при разрешённых прерываниях**.

Исторически в Linux сменялось несколько разнообразных API реализации этой схемы (сами названия «верхняя половина» и «нижняя половина» - это дословно названия одной из старых схем, которая сейчас не присутствует в ядре). С появлением параллелизмов в ядре Linux, все новые схемы реализации обработчиков нижней половины (рассматриваются далее) построены на выполнении такого обработчика **отдельным потоком ядра**.

Такого весьма поверхностного изложения схемы обработки аппаратных прерываний нам уже достаточно для всего дальнейшего экскурса в детали прерываний. Но прежде нужно очень тщательно сформулировать ту самую отличительную сторону прерываний, делающую работу с ними сложной: что же такое есть контекст выполнения в Linux. В Linux все синхронные ветви последовательно выполняющегося кода могут принадлежать к категориям:

1. Пользовательский **процесс** (приложение).
2. Отдельный **поток** (POSIX `pthread_t`) в многопоточном пользовательском процессе. (Как мы увидим, в Linux ядро диспетчирует именно потоки, **задачи**, поэтому в отношении предыдущего пункта было бы правильнее сказать: главный **поток** пользовательского приложения, представленный функцией `main()`.)
3. Linux — операционная система с **вытеснением в ядре**. Поэтому в ядре существуют автономные **потоки ядра**, они и представляют третью категорию.

Не взирая на различия в предназначении, выполняемые единицы всех этих категорий представляются в ядре (`<linux/sched.h>`) **одинаковой** структурой **задачи**: `struct task_struct`. Более того, для диспетчирования ядром **все** они включены в **единую** систему приоритетных циклических очередей. Структура задачи `struct task_struct` — это очень крупная структура, содержащая, кроме прочего, полный набор текущих параметров в состоянии, например, когда задача прерывается (совершенно не обязательно асинхронно и аппаратным

прерыванием — просто у задачи может истечь выделенный ей квант времени, и она будет прервана диспетчером ядра). Таким образом, `struct task_struct` содержит полный **необходимый и достаточный** набор всех параметров, чтобы приостановленную задачу активировать в сколь угодно отдалённом будущем. Называют, что все эти категории «активностей» выполняются **в контексте задачи**.

При возникновении аппаратного прерывания, текущая задача прерывается, её контекст сохраняется, и управление передаётся функции ISR (верхней половины). Но для выполняющего ISR кода **не создаётся** `struct task_struct`. И такое выполнение называют: **контекст прерывания**. И стоит функции ISR вызвать любую другую функцию API ядра, вызывающею блокирование вызывающего кода (начиная с такой безобидной паузы как `msleep()`) и управление будет передано коду какой-то из других задач (`struct task_struct`), но оно **никогда не возвратится обратно**, потому, что у ISR нет контекста задачи, в который можно возвратиться. И произойдёт подобное не только при прямом вызове таких блокирующих функций API, но и при вызове в сколь угодно длинной последовательной цепочке вызовов (некоторые из которых могут быть писаны другим автором). А последствием такого неосмотрительного действия программиста станет **общий крах ядра системы**. В этом состоит, пожалуй, главная сложность обработки прерываний в модуле ядра: непрерывное отслеживание того, в каком контексте отрабатывает код, который вы создаёте.

Наблюдение прерываний в /proc

Но, прежде чем дальше углубляться в организацию обработки прерывания, коротко остановимся на том, как мы можем наблюдать и контролировать то, что происходит с прерываниями. Всякий раз, когда аппаратное прерывание обрабатывается процессором, внутренний счётчик прерываний увеличивается, предоставляя возможность контроля за подсистемой прерываний. Счётчики отображаются в `/proc/interrupts` (последняя колонка это и есть имя обработчика, зарегистрированное (как это будет показано далее) параметром `name` при регистрации в вызове `request_irq()`). Ниже показана «раскладка» прерываний в архитектуре x86 при использовании старого, стандартного программируемого контроллера прерываний PC 8259 (если быть точнее, то показана схема с двумя каскодно объединёнными по линии IRQ2 контроллерами 8259, которая была классикой более 20 лет — чип контроллера прерываний 8259 создавался ещё под 8-бит процессор 8080). Такую картину можно наблюдать только на компьютере с одним процессором :

```
$ cat /proc/interrupts
           CPU0
  0:   33675789          XT-PIC  timer
  1:      41076          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  5:         18          XT-PIC  uhci_hcd:usb1, CS46XX
  6:          3          XT-PIC  floppy
  7:          0          XT-PIC  parport0
  8:          1          XT-PIC  rtc
  9:          0          XT-PIC  acpi
 11:    2153158          XT-PIC  ide2, eth0, mga@pci:0000:01:00.0
 12:     347114          XT-PIC  i8042
 14:         38          XT-PIC  ide0
```

Те линии IRQ, для которых не установлены текущие обработчики прерываний, не отображаются в `/proc/interrupts`. Здесь уже хорошо видно разделение линии IRQ 11 между тремя различными PCI устройствами.

То же самое, но на существенно более новом компьютере с несколькими процессорами (ядрами), когда источником прерываний является усовершенствованный контроллер прерываний IO-APIC (отслеживаются прерывания как **по фронту** — `IO-APIC-edge`, так и **по уровню** — `IO-APIC-level/IO-APIC-fasteoi`):

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:         49          0          0          0   IO-APIC-edge      timer
  1:          8          0          0          0   IO-APIC-edge      i8042
  4:          2          0          0          0   IO-APIC-edge
```

```
  7:          0          0          0          0   IO-APIC-edge      parport0
  8:          1          0          0          0   IO-APIC-edge      rtc0
  9:          0          0          0          0   IO-APIC-fasteoi   acpi
 12:        144          0          0          0   IO-APIC-edge      i8042
 14:          0          0          0          0   IO-APIC-edge      ata_piix
 15:          0          0          0          0   IO-APIC-edge      ata_piix
 16:         30          0          0          0   IO-APIC-fasteoi   uhci_hcd:usb5, i915
 18:          0          0          0          0   IO-APIC-fasteoi   uhci_hcd:usb4
 19:       5620       7184          0          0   IO-APIC-fasteoi   ata_piix, uhci_hcd:usb3
 22:        572          0          0          0   IO-APIC-fasteoi   HDA Intel
 23:          0          0          0          0   IO-APIC-fasteoi   ehci_hcd:usb1, uhci_hcd:usb2
 27:         83          0        157          0   PCI-MSI-edge      eth0
NMI:          0          0          0          0   Non-maskable interrupts
LOC:       6646       8926       5769       5409   Local timer interrupts
...
```

(Обратим внимание на прерывания по линии IRQ 27, когда прерывания от сетевого адаптера `eth0` чередуясь обрабатываются на разных процессорах.)

Чрезвычайно важным является то, что различные линии IRQ могут быть запрограммированы в контроллере IO-APIC на разные механизмы срабатывание: **по фронту** — `IO-APIC-edge`, или **по уровню** — `IO-APIC-fasteoi`.) Механизм срабатывания вы, как разработчик проекта, можете выбирать по собственному усмотрению при установке обработчика прерывания (ISR), будет показано немного далее. Это может иметь принципиальное значение: обработка по фронту может пропускать прерывания, происходящие на одной линии IRQ от 2-х разных устройств, а обработка по уровню — приводить к ложным срабатываниям при неправильной обработке. Для нескольких устройств на шине PCI, разделяющих одну линию IRQ, чаще всего выбирают обработкку прерываний по уровню. (Смысл и процесс обработки по уровню и по фронту лучше всех показаны Р.Кёртеном в [14]).

Ещё одним источником (динамической) информации о произошедших (обработанных) прерываниях является файл `/proc/stat`:

```
$ cat /proc/stat
cpu  960 352 3120 226661 670 9 60 0 0
cpu0 265 103 367 56843 304 8 47 0 0
cpu1 246 102 824 56630 204 1 12 0 0
cpu2 224 51 863 56717 80 0 0 0 0
cpu3 224 94 1065 56470 80 0 0 0 0
intr 45959 49 8 0 0 2 0 0 0 1 0 0 0 144 0 0 0 30 0 0 12825 0 0 572 0 0 0 0 249 0 0 ...
```

Здесь строка, начинающаяся с `intr`, содержит (начиная со 2-го числового значения) **суммарное** число обработанных прерываний **по всем процессорам**, для всех последовательных номеров линий IRQ (IRQ 0, IRQ 1, IRQ 2 ... — сравните с предыдущим показанным выводом, они сделаны **почти** одновременно).

Мы можем не только наблюдать, но и управлять (что понадобится вскоре) распределением прерываний по процессорам:

```
# cat /proc/irq/27/smp_affinity
4
# cat /proc/interrupts | grep eth0
 27:          83          0       4102          0   PCI-MSI-edge      eth0
```

Видим, что уже после загрузки афинити-маска (битовая маска разрешённых процессоров) для IRQ 27 была установлена системой в 4(100). Мы можем изменить это распределение на 2(10):

```
# echo 2 > /proc/irq/27/smp_affinity
# cat /proc/irq/27/smp_affinity
2
# watch -n1 'cat /proc/interrupts | grep eth0'
 27:          83        117       4111          0   PCI-MSI-edge      eth0
```

Регистрация обработчика прерывания

Все функции и определения, реализующие интерфейс обработчика прерывания, объявлены в `<linux/interrupt.h>`. Первое, что мы должны всегда сделать — это зарегистрировать функцию обработчик (ISR) прерываний (все прототипы этого раздела взяты из ядра 2.6.37):

```
typedef irqreturn_t (*irq_handler_t)( int, void* );
int request_irq( unsigned int irq, irq_handler_t handler, unsigned long flags,
                 const char *name, void *dev );
extern void free_irq( unsigned int irq, void *dev );
```

- где:

`irq` - номер линии запрашиваемого прерывания.

`handler` - указатель на функцию-обработчик.

`flags` - битовая маска опций (описываемая далее), связанная с управлением прерыванием.

`name` - символьная строка, используемая в `/proc/interrupts`, для отображения владельца прерывания.

`dev` - указатель на уникальный идентификатор устройства на линии IRQ, для не разделяемых прерываний (например шины ISA) может указываться `NULL`. Данные по указателю `dev` требуются для удаления только специфицируемого устройства на разделяемой линии IRQ. Первоначально накладывалось единственное требование, чтобы этот указатель был уникальным, например, при размещении-освобождении N однотипных устройств вполне допустимым могла бы быть конструкция:

```
for( int i = 0; i < N; i++ ) request_irq( irq, handler, 0, const char *name, (void*)i );
...
for( int i = 0; i < N; i++ ) free_irq( irq, (void*)i );
```

Здесь указатель `(void*)i` совершенно бессмысленный **в качестве указателя**, но совершенно нормальный по назначению своего использования. В некоторых случаях показанный фрагмент — это совершенно пригодный фрагмент кода. Но позже оказалось целесообразным и удобным использовать именно в качестве `*dev` — указатель на специфическую для устройства структуру, которая содержит все характерные данные экземпляра устройства: поскольку для каждого экземпляра размещается своя копия структуры, то указатели на них и будут уникальными, что и требовалось. Этот же указатель будет передаваться в функцию-обработчик `handler()` вторым параметром при **каждом** прерывании, тем самым передавая уникальный идентификатор источника произошедшего прерывания. На сегодня это общеупотребимая практика увязывать обработчик прерывания со структурами данных устройства.

Примечание: прототипы `irq_handler_t` и флаги установки обработчика существенно меняются от версии к версии, например, радикально поменялись после 2.6.19, все флаги, именуемые сейчас `IRQF_*` до этого именовались `SA_*`. В результате этого можно встретиться с невозможностью компиляции даже относительно недавно разработанных модулей-драйверов.

Флаги установки обработчика:
- группа флагов установки обработчика по уровню (level-triggered) или фронту (edge-triggered):

```
#define IRQF_TRIGGER_NONE    0x00000000
#define IRQF_TRIGGER_RISING  0x00000001
#define IRQF_TRIGGER_FALLING 0x00000002
#define IRQF_TRIGGER_HIGH    0x00000004
#define IRQF_TRIGGER_LOW     0x00000008
#define IRQF_TRIGGER_MASK ( IRQF_TRIGGER_HIGH | IRQF_TRIGGER_LOW |
                            IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING )
#define IRQF_TRIGGER_PROBE   0x00000010
```

- другие (не все, только основные, часто используемые) флаги:
`IRQF_SHARED` — разрешить разделение (совместное использование) линии IRQ с другими устройствами (PCI шина и устройства).
`IRQF_PROBE_SHARED` — устанавливается вызывающим, когда он предполагает возможные проблемы с совместным использованием.
`IRQF_TIMER` — флаг, маркирующий это прерывание как таймерное.
`IRQF_PERCPU` — прерывание закреплённое монопольно за отдельным CPU.

`IRQF_NOBALANCING` — флаг, запрещающий вовлекать это прерывание в балансировку IRQ.

При успешной установке функция `request_irq()` возвращает нуль. Возврат ненулевого значения указывает на то, что произошла ошибка и указанный обработчик прерывания не был зарегистрирован. Наиболее часто встречающийся код ошибки — это значение `-EBUSY` (ошибки в ядре возвращаются отрицательными значениями!), что указывает на то, что данная линия запроса на прерывание уже занята (или при текущем вызове, или при предыдущем вызове для этой линии не был указан флаг `IRQF_SHARED`).

С регистрацией нового обработчика прерываний всё просто. Но здесь есть одна маленькая (нигде не документированная, мне, по крайней мере, не удалось найти) деталь, которая может вызвать большую досаду при работе. Параметр `name` вызова `request_irq()` — это просто **указатель** на константную строку имени, но эта строка никуда не копируется (как это обычно принято в API пространства пользователя), и указатель указывает на строку всё время, пока загружен модуль. А отсюда следуют далеко идущие последствия. Вот такой вызов в функции инициализации модуля будет замечательно работать:

```
request_irq( irq, handler, 0, "my_interrupt", NULL );
```

И с таким будет всё как вы ожидали:

```
int init_module( void ) {
   char *dev = "my_interrupt";
   ...
   request_irq( irq, handler, 0, dev, NULL );
}
```

Но уже вот такой, очень похожий, код даст вам в `/proc/interrupts` не читаемую ерунду (и это в лучшем случае, если вам сильно повезёт — вы в ядре!):

```
int init_module( void ) {
   char dev[] = "my_interrupt";
   ...
   request_irq( irq, handler, 0, dev, NULL );
}
```

Здесь строка имени размещена и инициализирована в стеке, и после завершения функции инициализации она уже не существует ... но модуль то существует? Перепишем этот фрагмент и всё опять заработает:

```
char dev[] = "my_interrupt";
...
int init_module( void ) {
   ...
   request_irq( irq, handler, 0, dev, NULL );
}
```

Особенно сложно подлежащие толкованию результаты вы получите из-за этой особенности, если в одном модуле собираетесь зарегистрировать несколько обработчиков прерываний:

```
int init_module( void ) {
   int i;
   char *dev = "serial_xx";
   for( i = 0; i < num; i++ ) {
      sprintf( dev, "serial_%02d", i + 1 );
      request_irq( irq, handler, IRQF_SHARED, dev, (void*)( i + 1 ) );
   }
```

И что мы получим в этом случае? Правильно, мы получим `num` идентичных копий имён обработчиков (идентичных последнему заполнению), поскольку все экземпляры обработчиков будут использовать одну копию строки имени:

```
$ cat /proc/interrupts
...
  22:       1652   IO-APIC-fasteoi   ohci_hcd:usb2, serial_04, serial_04, serial_04, serial_04
...
```

И ещё одно очень серьёзное предупреждение относительно удаления обработчика `free_irq()`. В приложениях пространства пользователя мы можем себе позволить изрядную небрежность относительно завершающих действий программы: не выполнять `close()` для открытых дескрипторов файлов, не выполнять

`free()` для динамически выделенных блоков памяти... Эти шалости прощаются пользователю поскольку при завершении программы выполняется так называемый эпилог, в котором система выполнит все не указанные явно действия. Если же в модуле при его завершении (выгрузке) вы не выполните явно `free_irq()`, то почти со 100% вероятностью произойдёт следующее:

- модуль будет выгружен, но вектор прерывания будет установлен на тот адрес, который перед тем занимала зарегистрированная функция `handler()` ...
- по истечению некоторого времени эта область памяти будет переписана ядром под какие-то иные цели...
- и первое же произошедшее после этого аппаратное прерывание по этой линии IRQ приведёт к немедленному краху всей системы.

Обработчик прерываний, верхняя половина

Сам обработчик прерывания часто называют (разработчики аппаратных решений, в других операционных системах, ...) как ISR (Interrupt Service Routine). Это и есть то, что в Linux обозначают как верхняя половина обработчика прерывания.

Прототип функции обработчика прерывания уже показывался выше:

```
typedef irqreturn_t (*irq_handler_t)( int irq, void *dev );
```

где :

- `irq` — линия IRQ, чем меньше номер линии IRQ, тем выше приоритет обработки этого прерывания;

- `dev` — уникальный указатель экземпляра обработчика (именно тот, который передавался последним параметром `request_irq()` при регистрации обработчика).

Это именно та функция, которая будет вызываться в первую очередь при каждом возникновении аппаратного прерывания. Но это вовсе не означает, что при возврате из этой функции работа по обработке текущего прерывания будет завершена (хотя и такой вариант вполне допустим). Из-за этой «неполноты» такой обработчик и получил название «верхняя половина» обработчика прерывания. Дальнейшие действия по обработке могут быть запланированы эти обработчиком на более позднее время, используя несколько различных механизмов, обобщённо называемых «нижняя половина».

Важно то, что код обработчика верхней половины выполняется при **запрещённых** последующих прерываниях **по линии** `irq` (этой же линии) для того **локального** процессора, на котором этот код выполняется. А после возврата из этой функции локальные прерывания будут вновь разрешены.

Возвращается значение (`<linux/irqreturn.h>`):

```
typedef int irqreturn_t;
#define IRQ_NONE        (0)
#define IRQ_HANDLED     (1)
#define IRQ_RETVAL(x)   ((x) != 0)
```

`IRQ_HANDLED` — устройство прерывания распознано как обслуживаемое обработчиком, и прерывание успешно обработано.

`IRQ_NONE` — устройство не является источником прерывания для данного обработчика, прерывание должно быть передано далее другим обработчикам, зарегистрированным на данной линии IRQ.

Типичная схема обработчика при этом будет выглядеть так:

```
static irqreturn_t intr_handler ( int irq, void *dev ) {
   if( ! /* проверка того, что обслуживаемое устройство запросило прерывание*/ )
      return IRQ_NONE;
```

```
    /* код обслуживания устройства */
    return IRQ_HANDLED;
}
```

Пока мы не углубились в дальнейшую обработку, производимую в нижней половине, хотелось бы отметить следующее: в ряде случаев (при крайне простой обработке, но, самое главное, отсутствии возможности очень быстрых наступлений повторных прерываний) оказывается вполне достаточно простого обработчика верхней половины, и нет необходимости мудрить со сложно диагностируемыми механизмами отложенной обработки.

Здесь же напомним **самую большую сложность** в программировании функции обработчика: код функции ни непосредственно, ни косвенно, через сколь угодно длинную цепочку вложенных вызовов, не имеет права вызывать ни единой функции API ядра, которые вызывают переход выполняющегося кода в **блокированное** состояние ожидания (`msleep()` и др.), или даже только **предполагают** возможность такого перехода при некоторых условиях (`kmalloc()` и др.)! Это связано с тем, что функция обработчика верхней половины выполняется **в контексте прерывания**, и после завершения блокированного ожидания будет просто некуда возвратиться для продолжения её выполнения. Результатом такого ошибочного вызова будет, с большой вероятностью, крах всей операционной системы. (Отметим, что таких блокирующих вызовов в составе API ядра достаточно много.)

Управление линиями прерывания

Под управлением линиями прерываний, в этом месте описаний, мы будем понимать запрет-разрешение прерываний по одной или нескольким линиям `irq`. Раньше существовала возможность вообще запретить прерывания (на время, естественно). Но сейчас («заточенный» под SMP) набор API для этих целей выглядит так: либо вы запрещаете прерывания по всем линиям `irq`, но локального процессора, либо на всех процессорах, но только для одной линии `irq`.

Макросы управления линиями прерываний определены в `<linux/irqflags.h>`. Управление запретом и разрешением прерываний на локальном процессоре:

`local_irq_disable()` - запретить прерывания на локальном CPU;

`local_irq_enable()` - разрешить прерывания на локальном CPU;

`int irqs_disabled()` - возвратить ненулевое значение, если запрещены прерывания на локальном CPU, в противном случае возвращается нуль ;

Напротив, управление (запрет и разрешение) одной выбранной линией `irq`, но уже относительно всех процессоров в системе, делают макросы:

`void disable_irq( unsigned int irq ) -`

`void disable_irq_nosync( unsigned int irq ) -` обе эти функции запрещают прерывания с линии `irq` на контроллере (для всех CPU), причём, `disable_irq()` не возвращается до тех пор, пока все обработчики прерываний, которые в данный момент выполняются, не закончат работу;

`void enable_irq( unsigned int irq ) -` разрешаются прерывания с линии `irq` на контроллере (для всех CPU);

`void synchronize_irq( unsigned int irq )` - ожидает пока завершится обработчик прерывания от линии `irq` (если он выполняется), в принципе, хорошая идея — всегда вызывать эту функцию перед выгрузкой модуля использующего эту линию IRQ;

Вызовы функций `disable_irq*()` и `enable_irq()` должны обязательно быть **парными** - каждому вызову функции запрещения линии должен соответствовать вызов функции разрешения. Только после последнего вызова функции `enable_irq()` линия запроса на прерывание будет снова разрешена.

Пример обработчика прерываний

Обычно затруднительно показать работающий код обработчика прерываний, потому что такой код

должен был бы быть связан с реальным аппаратным расширением, и таким образом он будет перегружен специфическими деталями, скрывающими суть происходящего. Но оригинальный пример приведен в [6] откуда мы его и заимствуем (архив `IRQ.tgz`):

lab1_interrupt.c :

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

#define SHARED_IRQ 17

static int irq = SHARED_IRQ, my_dev_id, irq_counter = 0;
module_param( irq, int, S_IRUGO );

static irqreturn_t my_interrupt( int irq, void *dev_id ) {
   irq_counter++;
   printk( KERN_INFO "In the ISR: counter = %d\n", irq_counter );
   return IRQ_NONE;  /* we return IRQ_NONE because we are just observing */
}

static int __init my_init( void ) {
   if ( request_irq( irq, my_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id ) )
      return -1;
   printk( KERN_INFO "Successfully loading ISR handler on IRQ %d\n", irq );
   return 0;
}

static void __exit my_exit( void ) {
   synchronize_irq( irq );
   free_irq( irq, &my_dev_id );
   printk( KERN_INFO "Successfully unloading, irq_counter = %d\n", irq_counter );
}

module_init( my_init );
module_exit( my_exit );
MODULE_AUTHOR( "Jerry Cooperstein" );
MODULE_DESCRIPTION( "LDD:1.0 s_08/lab1_interrupt.c" );
MODULE_LICENSE( "GPL v2" );
```

Логика этого примера в том, что обработчик вешается в цепочку с существующим в системе, но он не нарушает работу ранее работающего обработчика, фактически ничего не выполняет, но подсчитывает число обработанных прерываний. В оригинале предлагается опробовать его с установкой на IRQ сетевой платы, но ещё показательнее — с установкой на IRQ клавиатуры (IRQ 1) или мыши (IRQ 12) на интерфейсе PS/2 (если таковой используется в компьютере):

```
$ cat /proc/interrupts
           CPU0
  0:   20329441          XT-PIC  timer
  1:        423          XT-PIC  i8042
...
$ sudo /sbin/insmod lab1_interrupt.ko irq=1
$ cat /proc/interrupts
           CPU0
  0:   20527017          XT-PIC  timer
  1:        572          XT-PIC  i8042, my_interrupt
...
$ sudo /sbin/rmmod lab1_interrupt
$ dmesg | tail -n5
In the ISR: counter = 33
In the ISR: counter = 34
```

```
In the ISR: counter = 35
In the ISR: counter = 36
Successfully unloading, irq_counter = 36
$ cat /proc/interrupts
           CPU0
  0:   20568216          XT-PIC  timer
  1:        622          XT-PIC  i8042
...
```

Оригинальность такого подхода в том, что на подобном коде можно начать отрабатывать код модуля реального устройства, ещё не имея самого устройства, и имитируя его прерывания одним из штатных источников прерываний компьютера, с тем, чтобы позже всё это переключить на реальную линию IRQ, используемую устройством.

Отложенная обработка, нижняя половина

Идея отложенной обработки прерывания состоит в том, чтобы только самые экстремальные действия выполнить непосредственно в обработчике, активирующемся по возникновению прерывания, («верхняя половина»), а все последующие действия «неспешно» отложить на более позднее время («нижняя половина»), на то время, когда система будет менее загружена. Главная достигаемая здесь цель состоит в том, что отложенную обработку можно производить не в самой функции обработчика прерывания (контексте прерывания, который нужно возбуждать **немедленно**), а к этому моменту времени может быть уже восстановлено разрешение прерываний по обслуживаемой линии (непосредственно в обработчике прерываний последующие прерывания запрещены).

Термин «нижняя половина» обработчика прерываний как раз и сложился для обозначения той совокупности действий, которую можно отнести к отложенной обработке прерываний. **Когда-то** в ядре Linux был один из способов организации отложенной обработки, который так и именовался: обработчик нижней половины, но сейчас он неприменим. А термин так и остался как нарицательный, относящийся к всем разным способам организации отложенной обработки, которые и рассматриваются далее.

Отложенные прерывания (softirq)

Отложенные прерывания определяются статически **во время компиляции ядра**. Поэтому, забегая вперёд, примем на заметку, что технику отложенных прерываний можно реализовать **в чистом виде** только если произвести новую сборку ядра Linux под свои требования.

Отложенные прерывания представлены с помощью структур `softirq_action`, определенных в файле `<linux/interrupt.h>` в следующем виде (ядро 2.6.37):

```
// структура, представляющая одно отложенное прерывание
struct softirq_action {
   void (*action)(struct softirq_action *);
};
```

В ядре 2.6.18 (и везде в литературе) определение (более раннее) другое:

```
struct softirq_action {
   void (*action)(struct softirq_action *);
   void *data;
};
```

Для уточнения картины с `softirq_action` вам будет недостаточно заголовочных файлов (это один из редких таких случаев), и необходимо будет окунуться в рассмотрение исходных кодов **реализации** ядра (файл `<kernel/softirq.c>`: если у вас не установлены исходные тексты ядра, то нужно либо сделать это, либо, что более разумно, обратиться к таким ресурсам как http://lxr.free-electrons.com/ или http://lxr.linux.no/):

```
enum {          /* задействованные номера */
   HI_SOFTIRQ=0,
   TIMER_SOFTIRQ,
   NET_TX_SOFTIRQ,
   NET_RX_SOFTIRQ,
   BLOCK_SOFTIRQ,
   BLOCK_IOPOLL_SOFTIRQ,
   TASKLET_SOFTIRQ,
   SCHED_SOFTIRQ,
   HRTIMER_SOFTIRQ,
   RCU_SOFTIRQ, /* Preferable RCU should always be the last softirq */
   NR_SOFTIRQS  /* число задействованных номеров */
};
static struct softirq_action softirq_vec[NR_SOFTIRQS]
char *softirq_to_name[NR_SOFTIRQS] = {
   "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "BLOCK_IOPOLL",
   "TASKLET", "SCHED", "HRTIMER",<>"RCU"
};
```

В 2.6.18 (то, что кочует из одного литературного источника в другой) аналогичные описания были заметно проще и более статичные (а потому и понятнее):

```
enum {
   HI_SOFTIRQ=0,
   TIMER_SOFTIRQ,
   NET_TX_SOFTIRQ,
   NET_RX_SOFTIRQ,
   BLOCK_SOFTIRQ,
   TASKLET_SOFTIRQ
};
static struct softirq_action softirq_vec[32]
```

Следовательно, имеется возможность создать 32 обработчика `softirq`, и это количество фиксировано. В этой версии ядра (2.6.18) их было 32, из которых задействованных было 6. Эти определения из предыдущей версии помогают лучше понять то, что имеет место в настоящее время.

Динамическая диагностика использования `softirq` в работающей системе может производиться так:

```
$ cat /proc/softirqs
                CPU0       CPU1       CPU2       CPU3
      HI:          0          0          0          0
   TIMER:     764858     806727     615475     617660
  NET_TX:          0          0       2520          0
  NET_RX:        777        700     471255        521
   BLOCK:      48435     112893         12      24613
BLOCK_IOPOLL:       0          0          0          0
 TASKLET:        108         81          5          1
   SCHED:     405389     430031     327195     345751
 HRTIMER:        985       1356       1211       1518
     RCU:     884440     850717     837750     672217
```

В любом случае (независимо от версии), добавить новый уровень обработчика (назовём его `XXX_SOFT_IRQ`) без перекомпиляции ядра мы не сможем. Максимальное число используемых обработчиков `softirq` не может быть динамически изменено. Отложенные прерывания с меньшим номером выполняются раньше отложенных прерываний с большим номером (приоритетность). Обработчик одного отложенного прерывания никогда не вытесняет другой обработчик `softirq`. Единственное событие, которое может вытеснить обработчик `softirq`, — это аппаратное прерывание. Однако на другом процессоре одновременно с обработчиком отложенного прерывания может выполняться другой (**и даже этот же**) обработчик отложенного прерывания. Отложенное прерывание выполняется **в контексте прерывания**, а значит для него недопустимы блокирующие операции.

Если вы решились на пересборку ядра (в чём нет ничего страшного) и создание нового уровня `softirq`, то для этого необходимо:

1. Определить новый индекс (уровень) отложенного прерывания, вписав (файл `<linux/interrupt.h>`) свою константу вида `XXX_SOFT_IRQ` в перечисление, где-то, очевидно, на одну позицию выше `TASKLET_SOFTIRQ` (иначе зачем переопределять новый уровень и не использовать тасклет?).
2. Во время инициализации модуля должен быть зарегистрирован (объявлен) обработчик отложенного прерывания с помощью вызова `open_softirq()`, который принимает три параметра: этот индекс отложенного прерывания, функция-обработчик и значение поля `data` :

```
/* The bottom half */
void xxx_analyze( void *data ) {
  /* Analyze and do ................ */
}
void __init roller_init() {
   /* ... */
   request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
   open_softirq( XXX_SOFT_IRQ, xxx_analyze, NULL );
}
```

3. Функция-обработчик отложенного прерывания (в точности как и рассматриваемого далее тасклета) должна в точности соответствовать правильному прототипу:

```
void xxx_analyze( unsigned long data );
```

4. Зарегистрированное отложенное прерывание, для того, чтобы оно было поставлено в очередь на выполнение, должно быть отмечено (генерировано, возбуждено: rise softirq). Это называется генерацией отложенного прерывания. Обычно обработчик аппаратного прерывания (ISR, верхней половины) перед возвратом возбуждает свои обработчики отложенных прерываний:

```
/* The interrupt handler */
static irqreturn_t xxx_interrupt( int irq, void *dev_id ) {
   /* ... */
   /* Mark softirq as pending */
   raise_softirq( XXX_SOFT_IRQ );
   return IRQ_HANDLED;
}
```

5. Затем, в подходящий (не для вас, для системы) момент времени отложенное прерывание начнёт выполнятся. Обработчик отложенного прерывания выполняется при разрешённых прерываниях процессора (особенность нижней половины). Во время выполнения обработчика отложенного прерывания новые отложенные прерывания **на данном** процессоре запрещаются. Однако на другом процессоре обработчики отложенных прерываний могут выполняться. На самом деле, если вдруг генерируется отложенное прерывание в тот момент, когда ещё выполняется предыдущий его обработчик, то такой же обработчик может быть запущен на другом процессоре одновременно с первым обработчиком. Это означает, что любые совместно используемые данные, которые используются в обработчике отложенного прерывания, и даже глобальные данные, которые используются только внутри самого обработчика, должны соответствующим образом ограждаться примитивами синхронизации.

С одной стороны, всё достаточно просто — можете поверить, а ещё лучше проверить, что набросанной выше схемы вполне достаточно для реализации кода. С другой стороны всё как-то тяжеловесно и усложнено, особенно в сравнении с альтернативными методами, рассматриваемыми далее. Вопрос: зачем? Ответ прост: главная причина использования отложенных прерываний — превосходная **масштабируемость на многие процессоры**. Это становится актуальным на сегодня, когда число ядер ординарного настольного компьютера может составлять 8, и может серьёзно увеличиться за ближайшее время. Но если нет необходимости масштабирования на многие процессоры, то лучшим выбором будет механизм тасклетов.

Тасклеты

(Не ищите перевода этому термину, это жаргон: если task — это отдельная задача в терминах ядра, то tasklet, надо понимать, это так ... «маленькая задачка»).

Предыдущая схема достаточно тяжеловесная, и в большинстве случаем её подменяют тасклеты — механизм на базе тех же softirq с двумя фиксированными индексами HI_SOFTIRQ или TASKLET_SOFTIRQ. Тасклеты это ни что иное, как частный случай реализации softirq. Тасклеты представляются (<linux/interrupt.h>) с помощью структуры:

```
struct tasklet_struct {
   struct tasklet_struct *next; /* указатель на следующий тасклет в списке */
   unsigned long state;         /* текущее состояние тасклета */
   atomic_t count;              /* счетчик ссылок */
   void (*func)(unsigned long); /* функция-обработчик тасклета*/
   unsigned long data;          /* аргумент функции-обработчика тасклета */
};
```

Поле state может принимать только одно из значений: 0, TASKLET_STATE_SCHED, TASLET_STATE_RUN. Значение TASKLET_STATE_SCHED указывает на то, что тасклет запланирован на выполнение, а значение TASLET_STATE_RUN — что тасклет выполняется.

```
enum {
   TASKLET_STATE_SCHED,   /* Tasklet is scheduled for execution */
   TASKLET_STATE_RUN      /* Tasklet is running (SMP only) */
};
```

Поле count используется как счетчик ссылок на тасклет. Если это значение не равно нулю, то тасклет запрещен и не может выполняться; если оно равно нулю, то тасклет разрешен и может выполняться в случае, когда он помечен как ожидающий выполнения.

Схематически код использования тасклета полностью повторяет структуру кода softirq:

- Инициализация тасклета при инициализации модуля:

```
struct xxx_device_struct { /* Device-specific structure */
  /* ... */
  struct tasklet_struct tsklt;
  /* ... */
}
void __init xxx_init() {
   struct xxx_device_struct *dev_struct;
   /* ... */
   request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
   /* Initialize tasklet */
   tasklet_init( &dev_struct->tsklt, xxx_analyze, dev );
}
```

Для статического создания тасклета (и соответственно, обеспечения прямого доступа к нему) могут использоваться один из двух макросов:

```
DECLARE_TASKLET( name, func, data )
DECLARE_TASKLET_DISABLED( name, func, data );
```

Оба макроса статически создают экземпляр структуры struct tasklet_struct с указанным именем (name). Второй макрос создает тасклет, но устанавливает для него значение поля count, равное единице, и, соответственно, этот тасклет будет запрещен для исполнения. Макрос DECLARE_TASKLET(name, func, data) эквивалентен (можно записать и так):

```
struct tasklet_struct namt = { NULL, 0, ATOMIC_INIT(0), func, data ) ;
```

Используется, что совершенно естественно, в точности тот же прототип функции обработчика тасклета, что и в случае отложенных прерываний (в моих примерах просто использована та же функция).

Для того чтобы запланировать тасклет на выполнение (обычно в обработчике прерывания), должна быть вызвана функция `tasklet_schedule()`, которой в качестве аргумента передается указатель на соответствующий экземпляр структуры `struct tasklet_struct`:

```
/* The interrupt handler */
static irqreturn_t xxx_interrupt( int irq, void *dev_id ) {
   struct xxx_device_struct *dev_struct;
   /* ... */
   /* Mark tasklet as pending */
   tasklet_schedule( &dev_struct->tsklt );
   return IRQ_HANDLED;
}
```

После того как тасклет запланирован на выполнение, он выполняется один раз в некоторый момент времени в ближайшем будущем. Для оптимизации тасклет всегда выполняется на том процессоре, который его запланировал на выполнение, что дает надежду на лучшее использование кэша процессора.

Если вместо стандартного тасклета нужно использовать тасклет высокого приоритета (`HI_SOFTIRQ`), то вместо функции `tasklet_schedule()` вызываем функцию планирования `tasklet_hi_schedule()`.

Уже запланированный тасклет может быть запрещен к исполнению (временно) с помощью вызова функции `tasklet_disable()`. Если тасклет в данный момент уже начал выполнение, то функция не возвратит управление, пока тасклет не закончит своё выполнение. Как альтернативу можно использовать функцию `tasklet_disable_nosync()`, которая запрещает указанный тасклет, но возвращается сразу не ожидая, пока тасклет завершит выполнение (это обычно небезопасно, так как в данном случае нельзя гарантировать, что тасклет не закончил выполнение). Вызов функции `tasklet_enable()` разрешает тасклет. Эта функция также должна быть вызвана для того, чтобы можно было выполнить тасклет, созданный с помощью макроса `DECLARE_TASKLET_DISABLED ()`. Из очереди тасклетов, ожидающих выполнения, тасклет может быть удален с помощью функции `tasklet_kill()`.

Так же как и в случае отложенных прерываний (на которых он построен), тасклет не может переходить в блокированное состояние.

Демон ksoftirqd

Обработка отложенных прерываний (`softirq`) и, соответственно, тасклетов осуществляется с помощью набора потоков пространства ядра (по одному потоку на каждый процессор). Потоки пространства ядра помогают обрабатывать отложенные прерывания, когда система перегружена большим количеством отложенных прерываний.

```
$ ps -ALf | head -n12
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root         1     0     1  0    1 08:55 ?        00:00:01 /sbin/init
...
root         4     2     4  0    1 08:55 ?        00:00:00 [ksoftirqd/0]
...
root         7     2     7  0    1 08:55 ?        00:00:00 [ksoftirqd/1]
...
```

Для каждого процессора существует свой поток. Каждый поток имеет имя в виде `ksoftirqd/n`, где `n` — номер процессора. Например, в двухпроцессорной системе будут запущены два потока с именами `ksoftirqd/0` и `ksoftirqd/1`. То, что на каждом процессоре выполняется свой поток, гарантирует, что если в системе есть свободный процессор, то он всегда будет в состоянии выполнять отложенные прерывания. После того как потоки запущены, они выполняют замкнутый цикл.

Здесь же попутно уместно напомнить, что в современном ядре Linux, даже в самых ординарных конфигурациях, может выполняться совсем немалое число автономных потоков ядра:

```
$ ps -ALf | grep ' \[' | wc -l
120
```

Очереди отложенных действий (workqueue)

Очереди отложенных действий (`workqueue`) — это еще один, но совершенно другой, способ реализации отложенных операций. Очереди отложенных действий позволяют откладывать некоторые операции для последующего выполнения **потоком пространства ядра** (эти потоки ядра называют рабочими потоками - worker threads) — отложенные действия всегда выполняются в **контексте процесса**. Поэтому код, выполнение которого отложено с помощью постановки в очередь отложенных действий, получает все преимущества, которыми обладает код, выполняющийся в контексте процесса, главное из которых — это возможность переходить в блокированные состояния. Рабочие потоки, которые выполняются по умолчанию, называются `events/n`, где `n` — номер процессора, например:

```
$ ps -ALf | grep '\['events
root        15     2    15  0    1 17:08 ?        00:00:00 [events/0]
root        16     2    16  0    1 17:08 ?        00:00:00 [events/1]
root        17     2    17  0    1 17:08 ?        00:00:00 [events/2]
root        18     2    18  0    1 17:08 ?        00:00:00 [events/3]
```

Когда какие-либо действия ставятся в очередь, поток ядра возвращается к выполнению и выполняет эти действия. Когда в очереди не остается работы, которую нужно выполнять, поток снова возвращается в состояние ожидания. Каждое действие представлено с помощью `struct work_struct` (определяется в файле `<linux/workqueue.h>` - очень меняется от версии к версии ядра!):

```
typedef void (*work_func_t)( struct work_struct *work );
struct work_struct {
   atomic_long_t data;      /* аргумент функции-обработчика */
   struct list_head entry;  /* связанный список всех действий */
   work_func_t func;        /* функция-обработчик */
...
};
```

Для создания статической структуры действия на этапе компиляции необходимо использовать макрос:

```
DECLARE_WORK( name, void (*func) (void *), void *data );
```

Это выражение создает `struct work_struct` с именем `name`, с функцией-обработчиком `func()` и аргументом функции-обработчика `data`. Динамически отложенное действие создается с помощью указателя на ранее созданную структуру, используя следующий макрос:

```
INIT_WORK( struct work_struct *work, void (*func)(void *), void *data );
```

Функция-обработчика имеет тот же прототип, что и для отложенных прерываний и тасклетов, поэтому в примерах будет использоваться та же функция (`xxx_analyze()`).

Для реализации нижней половины обработчика IRQ на технике `workqueue`, выполнем последовательность действий примерно следующего содержания:

При инициализации модуля создаём отложенное действие:

```
#include <linux/workqueue.h>
struct work_struct *hardwork;
void __init xxx_init() {
   /* ... */
   request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
   hardwork = kmalloc( sizeof(struct work_struct), GFP_KERNEL );
   /* Init the work structure */
   INIT_WORK( hardwork, xxx_analyze, data );
}
```

Или то же самое может быть выполнено статически

```
#include <linux/workqueue.h>
DECLARE_WORK( hardwork, xxx_analyze, data );
void __init xxx_init() {
```

```
    /* ... */
    request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
}
```

Самая интересная работа начинается когда нужно запланировать отложенное действие; при использовании для этого рабочего потока ядра по умолчанию (`events/n`) это делается функциями :

- `schedule_work( struct work_struct *work );` - действие планируется на выполнение немедленно и будет выполнено, как только рабочий поток events, работающий на данном процессоре, перейдет в состояние выполнения.

- `schedule_delayed_work( struct delayed_work *work, unsigned long delay );` - в этом случае запланированное действие не будет выполнено, пока не пройдет хотя бы заданное в параметре `delay` количество импульсов системного таймера.

В обработчике прерывания это выглядит так:

```
static irqreturn_t xxx_interrupt( int irq, void *dev_id ) {
    /* ... */
    schedule_work( hardwork );
    /* или schedule_work( &hardwork ); - для статической инициализации */
    return IRQ_HANDLED;
}
```

Очень часто бывает необходимо ждать пока очередь отложенных действий очистится (отложенные действия завершатся), это обеспечивает функция:

```
void flush_scheduled_work( void );
```

Для отмены незавершённых отложенных действий с задержками используется функция:

```
int cancel_delayed_work( struct work_struct *work );
```

Но мы совсем не обязательно должны рассчитывать на общую очереди (потоки ядра `events`) для выполнения отложенных действий — мы можем создать под эти цели собственные очереди (вместе с обслуживающим потоком). Создание обеспечивается макросами вида:

```
struct workqueue_struct *create_workqueue( const char *name );
struct workqueue_struct *create_singlethread_workqueue( const char *name );
```

Планирование на выполнение в этом случае осуществляют функции:

```
int queue_work( struct workqueue_struct *wq, struct work_struct *work );
int queue_delayed_work( struct workqueue_struct *wq,
                        struct wesrk_struct *work, unsigned long delay);
```

Они аналогичны рассмотренным выше `schedule_*()`, но работают с созданной очередью, указанной 1-м параметром. С вновь созданными потоками предыдущий пример может выглядеть так:

```
struct workqueue_struct *wq;
/* Driver Initialization */
static int __init xxx_init( void ) {
    /* ... */
    request_irq( irq, xxx_interrupt, 0, "xxx", NULL );
    hardwork = kmalloc( sizeof(struct work_struct), GFP_KERNEL );
    /* Init the work structure */
    INIT_WORK( hardwork, xxx_analyze, data );
    wq = create_singlethread_workqueue( "xxxdrv" );
    return 0;
}
static irqreturn_t xxx_interrupt( int irq, void *dev_id ) {
    /* ... */
    queue_work( wq, hardwork );
    return IRQ_HANDLED;
}
```

Аналогично тому, как и для очереди по умолчанию, ожидание завершения действий в заданной очереди может быть выполнено с помощью функции :

```
void flush_workqueue( struct workqueue_struct *wq );
```

Техника очередей отложенных действий показана здесь на примере обработчика прерываний, но она гораздо шире по сферам её применения (в отличие, например, от тасклетов), для других целей.

Сравнение и примеры

Начнём со сравнений. Оставив в стороне рассмотрение softirq, как механизм тяжёлый, и уже достаточно обсуждённый, в том смысле, что его использование оправдано при требовании масштабирования высокоскоростных процессов на большое число обслуживающих процессоров в SMP. Две другие рассмотренные схемы — это тасклеты и очереди отложенных действий. Они представляют две различные схемы реализации отложенных работ в современном Linux, которые переносят работы из верхних половин в нижние половины драйверов. В тасклетах реализуется механизм с низкой латентностью, который является простым и ясным, а очереди работ имеют более гибкий и развитый API, который позволяет обслуживать несколько отложенных действий в порядке очередей. В каждой схеме откладывание (планирование) последующей работы выполняется из контекста прерывания, но только тасклеты выполняют запуск автоматически в стиле «работа до полного завершения», тогда как очереди отложенных действий разрешают функциям-обработчикам переходить в блокированные состояния. В этом состоит главное принципиальное отличие: рабочая функция тасклета не может блокироваться.

Теперь можно перейти к примерам. Уже отмечалось, что экспериментировать с аппаратными прерываниями достаточно сложно. Кроме того, в ходе проводимых занятий неоднократно задавался вопрос: «Можно ли тасклеты использовать автономно, вне процесса обработки прерываний?». Вот так мы и построим иллюстрирующие модули: сама функция инициализации модуля будет активировать отложенную обработку. Ниже показан пример для тасклетов:

mod_tasklet.c :

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

MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_LICENSE( "GPL v2" );

static cycles_t cycles1, cycles2;
static u32 j1, j2;
static int context;

char tasklet_data[] = "tasklet function was called";

void tasklet_function( unsigned long data ) { /* Bottom Half Function */
   context = in_atomic();
   j2 = jiffies;
   cycles2 = get_cycles();
   printk( "%010lld [%05d] : %s in contxt %d\n",
           (long long unsigned)cycles2, j2, (char*)data, context );
   return;
}

DECLARE_TASKLET( my_tasklet, tasklet_function,
                 (unsigned long)&tasklet_data );

int init_module( void ) {
   context = in_atomic();
   j1 = jiffies;
   cycles1 = get_cycles();
   tasklet_schedule( &my_tasklet );   /* Schedule the Bottom Half */
   printk( "%010lld [%05d] : tasklet was scheduled in contxt %d\n",
           (long long unsigned)cycles1, j1, context );
```

```
   return 0;
}

void cleanup_module( void ) {
  tasklet_kill( &my_tasklet );        /* Stop the tasklet before we exit */
  return;
}
```

Вот как выглядит его исполнение:

```
$ uname -a
Linux modules.localdomain 3.14.8-200.fc20.x86_64 #1 SMP Mon Jun 16 21:57:53 UTC 2014 x86_64 x86_64
x86_64 GNU/Linux
$ sudo insmod mod_tasklet.ko
$ lsmod | head -n3
Module                  Size  Used by
mod_tasklet            12689  0
fuse                   86935  2
$ dmesg | tail -n2 | grep " : "
[ 2303.199450] 5995036764088 [2001684] : tasklet was scheduled in contxt 0
[ 2303.199461] 5995036802118 [2001684] : tasklet function was called in contxt 1
$ sudo rmmod mod_tasklet
$ ../time/clock
0000079AB4EAA738
0000079AB4EC1212
0000079AB4EC28F0
2594035590
```

По временным меткам видно, что выполнение функции тасклета происходит позже планирования тасклета на выполнение, но латентность очень низкая (системный счётчик `jiffies` не успевает изменить значение, всё происходит в пределах одного системного тика), отсрочка выполнения составляет порядка 30000 процессорных тактов частоты 2.5 Ghz (показан уже обсуждавшийся тест `clock` из раздела о службе времени, нас интересует только последняя строка его вывода). Специально показана диагностика контекста в функции инициализации модуля (контекст процесса) и в функции тасклета (атомарный контекст).

В следующем примере мы проделаем практически то же самое (близкие эксперименты для возможностей сравнения), но относительно очередей отложенных действий:

mod_workqueue.c :

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

MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_LICENSE( "GPL v2" );

static int works = 2; // число отложенных работ
module_param( works, int, S_IRUGO );

static struct workqueue_struct *my_wq;

typedef struct {
   struct work_struct my_work;
   int    id;
   u32    j;
   cycles_t cycles;
} my_work_t;

static void my_wq_function( struct work_struct *work ) { /* Bottom Half Function */
```

```
    u32 j = jiffies;
    cycles_t cycles = get_cycles();
    my_work_t *wrk = (my_work_t *)work;
    printk( "#%d : %010lld [%05d] => %010lld [%05d] = %06lu : context %d\n",
            wrk->id, (long long unsigned)wrk->cycles, wrk->j,
            (long long unsigned)cycles, j,
            (long unsigned)( cycles - wrk->cycles ), in_atomic()
          );
  kfree( (void *)wrk );
  return;
}

int init_module( void ) {
   int n, ret;
   if( ( my_wq = create_workqueue( "my_queue" ) ) )
      for( n = 0; n < works; n++ ) {
         /* One more additional work */
         my_work_t *work = (my_work_t*)kmalloc( sizeof(my_work_t), GFP_KERNEL );
         if( work ) {
            INIT_WORK( (struct work_struct *)work, my_wq_function );
            work->id = n;
            work->j = jiffies;
            work->cycles = get_cycles();
            ret = queue_work( my_wq, (struct work_struct*)work );
            if( !ret ) return -EPERM;
         }
         else return -ENOMEM;
      }
   else return -EBADRQC;
   return 0;
}

void cleanup_module( void ) {
  flush_workqueue( my_wq );
  destroy_workqueue( my_wq );
  return;
}
```

Вот как исполнение проходит на этот раз на том же самом компьютере:

```
$ sudo insmod mod_workqueue.ko works=5
$ lsmod | head -n3
Module                  Size  Used by
mod_workqueue           1079  0
vfat                    6740  1
$ ps -ef | grep root | grep my_
root     10163     2  0 22:46 ?        00:00:00 [my_queue]
$ dmesg | tail -n5
[11901.358300] #0 : 30877353486032 [11593784] => 30877353494880 [11593784] = 00008848 : context 0
[11901.358309] #1 : 30877353516980 [11593784] => 30877353521696 [11593784] = 00004716 : context 0
[11901.358315] #2 : 30877353534284 [11593784] => 30877353538372 [11593784] = 00004088 : context 0
[11901.358320] #3 : 30877353549788 [11593784] => 30877353553488 [11593784] = 00003700 : context 0
[11901.358326] #4 : 30877353563112 [11593784] => 30877353567272 [11593784] = 00004160 : context 0
$ sudo rmmod mod_workqueue
```

Здесь мы видим, как появился новый обрабатывающий поток ядра, с заданным нами именем. Теперь латентность реакции несколько больше случая тасклетов, что и соответствует утверждениям в литературе. Снова показана диагностика контекста в функциях отложенных работ, и теперь она — контекст процесса.

Обсуждение

При рассмотрении техники обработки прерываний возникает ряд тонких вопросов, на которые меня натолкнули участники проводимых мной тренингов. Одна из таких интересных групп вопросов (потому, что здесь, собственно, два вопроса), выглядит так:

- При регистрации нескольких обработчиков прерываний, разделяющих одну линию IRQ, какой будет порядок срабатывания по времени этих обработчиков (связанных в последовательный список): от позже зарегистрированных к более ранним (что было бы целесообразно), или же наоборот?
- При регистрации нескольких обработчиков прерываний, разделяющих одну линию IRQ, есть ли способы изменения последовательности срабатывания этих нескольких обработчиков?

На второй вопрос я (пока) не знаю ответа, а вот относительно первого рассмотрим ещё вот такой тест:

mod_ser.c :

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

MODULE_LICENSE( "GPL v2" );
#define SHARED_IRQ 1
#define MAX_SHARED 9
#define NAME_SUFFIX "serial_"
#define NAME_LEN   10
static int irq = SHARED_IRQ, num = 2;
module_param( irq, int, 0 );
module_param( num, int, 0 );

static irqreturn_t handler( int irq, void *id ) {
   cycles_t cycles = get_cycles();
   printk( KERN_INFO "%010lld : irq=%d - handler #%d\n", cycles, irq, (int)id );
   return IRQ_NONE;
}

static char dev[ MAX_SHARED ][ NAME_LEN ];

int init_module( void ) {
   int i;
   if( num > MAX_SHARED ) num = MAX_SHARED;
   for( i = 0; i < num; i++ ) {
      sprintf( dev[ i ], "serial_%02d", i + 1 );
      if( request_irq( irq, handler, IRQF_SHARED, dev[ i ], (void*)( i + 1 ) ) ) return -1;
   }
   return 0;
}

void cleanup_module( void ) {
   int i;
   for( i = 0; i < num; i++ ) {
      synchronize_irq( irq );
      free_irq( irq, (void*)( i + 1 ) );
   }
}
```

Здесь на одну (любую) линию IRQ (параметр модуля `irq`) устанавливается `num` (параметр `num`, по умолчанию 2) последовательно обработчиков прерывания, которые фиксируют время своего срабатывания. Используем этот модуль (инсталляция Ubuntu 10.04.3 в виртуальной машине в Virtual Box, ядро 2.6.32):

```
$ uname -r
```

```
2.6.32-33-generic
$ cat /proc/interrupts | grep hci
           CPU0
  5:      39793    XT-PIC-XT        ahci, Intel 82801AA-ICH
 10:      92471    XT-PIC-XT        ehci_hcd:usb1, eth0
 11:       3845    XT-PIC-XT        ohci_hcd:usb2
```

Нас интересует в этом случае линия IRQ 11 (это USB-мышь):

```
$ sudo insmod mod_ser.ko irq=11
$ lsmod | head -n3
Module                  Size  Used by
mod_ser                 1130  0
binfmt_misc             6587  1
$ cat /proc/interrupts | grep hci
  5:      16120    XT-PIC-XT        ahci, Intel 82801AA-ICH
 10:      59800    XT-PIC-XT        ehci_hcd:usb1, eth0
 11:       3820    XT-PIC-XT        ohci_hcd:usb2, serial_01, serial_02
```

И вот фрагмент системного журнала при перемещении мыши:

```
$ dmesg | grep 'irq=11' | head -n6
[ 9499.031303] 15199977878339 : irq=11 - handler #1
[ 9499.031341] 15199977948850 : irq=11 - handler #2
[ 9499.047312] 15200003431449 : irq=11 - handler #1
[ 9499.047351] 15200003502182 : irq=11 - handler #2
[ 9499.072494] 15200043610967 : irq=11 - handler #1
[ 9499.072532] 15200043682638 : irq=11 - handler #2
```

Здесь уже, по меткам счётчика процессорных тактов (`rdtsc`) мы можем предположить, что ранее зарегистрированный обработчик срабатывает раньше, то есть новые обработчики прерывания устанавливаются в хвост очереди разделяемых прерываний. Для подтверждения и сравнения то же действие, но на другой инсталляции (инсталляция Fedora 14 PFR в той же виртуальной машине, ядро 2.6.32 — сравнение покажет нам довольно интересные вещи):

```
$ uname -r
2.6.35.13-92.fc14.i686
$ cat /proc/interrupts | grep hci
  19:       8683   IO-APIC-fasteoi   ehci_hcd:usb1, eth0
  21:       9079   IO-APIC-fasteoi   ahci, Intel 82801AA-ICH
  22:       1634   IO-APIC-fasteoi   ohci_hcd:usb2
```

Уже здесь всё становится сильно интересно: на одном и том же компьютере виртуальные машины (разные дистрибутивы) видят один и тот же аппаратный контроллер прерываний совершенно различно, в этом случае мы станем использовать IRQ линию 22 (это всё та же USB-мышь):

```
$ sudo insmod mod_ser.ko irq=22 num=5
$ cat /proc/interrupts | grep 22:
  22:       1653   IO-APIC-fasteoi   ohci_hcd:usb2, serial_01, serial_02, serial_03, serial_04,
serial_05
$ sudo rmmod mod_ser
 $ dmesg | grep 'irq=22' | tail -n10
[10618.475392] 16971191379365 : irq=22 - handler #1
[10618.475392] 16971192198665 : irq=22 - handler #2
[10618.475392] 16971192685213 : irq=22 - handler #3
[10618.475392] 16971193423264 : irq=22 - handler #4
[10618.475392] 16971193880266 : irq=22 - handler #5
[10669.904309] 17053241497863 : irq=22 - handler #1
[10669.905331] 17053242842991 : irq=22 - handler #2
[10669.905331] 17053243293397 : irq=22 - handler #3
[10669.905331] 17053243600477 : irq=22 - handler #4
[10669.907843] 17053245722371 : irq=22 - handler #5
```

Здесь та же картина: ранее зарегистрированный обработчик и раньше срабатывает на прерывание. На реальной (не виртуальной) системе картина в точности та же.

Задачи

1. Каким образом вы выбираете конкретное значение IRQ для экспериментов с прерываниями в вашей конкретной Linux системе? Покажите на командах Linux.

2. Установите свой дополнительный обработчик прерываний (на любой IRQ, например сетевой карты), численное значение IRQ указывайте параметром загрузки модуля. Модуль должен диагностировать полученные прерывания в системный журнал и подсчитывать их общее число, диагностируя его при выгрузке модуля.

3. То же, что и в предыдущей задаче, но отображайте динамически общее число обслуженных прерываний в собственное путевое имя в `/proc`.

4. То же, что и предыдущее задание, но с отображением в `/sys` (создайте в `/sys` путевое имя аналогичное `/proc` в предыдущей задаче). Сравните решения для `/proc` и для `/sys`.

5. Подсчёт прошедших прерываний (счётчик) оформите в форме **тасклета**. Этот тасклет может только подсчитывать число произошедших прерываний (в минимальной реализации), или, например для формировать готовую строку вывода сразу по возникновению прерыванияя.

6. Подсчёт прошедших прерываний (счётчик) обрабатывайте в **очереди отложенных действий** (так же как в предыдущей задаче).

10. Периферийные устройства в модулях ядра

Обслуживание проприетарных (которые вы создаёте под свои цели) аппаратных расширений (для самых разнообразных целей) невозможно описать в общем виде: здесь вам предстоит работать в непосредственном и плотном контакте с разработчиком «железа», в постоянных консультациях по каким портам ввода-вывода выполнять операции и с какой целью. Поэтому задачи непосредственно **организации обмена** данными минимально затрагиваются в последующем тексте (да их и невозможно рассмотреть в описании обозримого объёма).

Мы рассмотрим только основные принципы учёта и связывания периферийных устройств в системе, те вопросы, которые позволяют непосредственно выйти на порты и адреса, по которым уже далее нужно читать-писать для обеспечения функционирования устройства по его собственной алгоритмике. Другими словами, нас здесь интересует вопрос «как зацепиться за устройство на шине», а последующая организация работы по обмену с этим устройством — это уже на откуп вам, совместно с вашим консультантом, или разработчиком аппаратуры устройства.

Кроме того, работа с оборудованием в **коде модуля** ядра очень часто и сильно завязана с общими принципами, стандартами и тенденциями в работе собственно «железа» (стандарты шин) и инструментами **пространства пользователя** (таких как `sysfs`, `udev`, `libusb`, ...), степень задействования которых последних версиях всё увеличивается. Это те вопросы, которые в публикациях по ядру не рассматриваются (из-за «не принадлежности»), а посвящённые им целевые публикации либо слишком перегружены ненужными деталями (по «железу»), либо никак не увязываются с процессами в ядре (для пользовательских проектов). Созданию минимальной связной картины взаимодействия этих компонент тоже будет специально уделено некоторое внимание.

Поддержка шинных устройств в модуле

Меня часто спрашивают в обсуждениях: «а устройство само сохдаёт имя в `/dev`?», «есть какие-то особенности в выборе `major` номера для USB устройств?», «а как выражаются в коде обменные операции для USB устройства?». Правильные ответы на эти вопросы должны звучать, соответственно так: «не создаёт», «для шинного устройства PCI или USB просто не создаётся `major` номер» и «никак не выражаются». И вот почему...

Дело в том, что **физические** устройства (в нашем представлении) на PCI или USB (платы, адаптеры, встроенные чипы, внешние устройства на USB и др.) в представлении Linux **логическими** устройствами не являются. Linux знает только 3 типа устройств: символьные устройства, блочные устройств и сетевые интерфейсы. А любое устройство PCI или USB может быть и первым (адаптеры синхронной связи E1/T1/J1), и вторым (флеш-диски, внешние HDD на USB), и третьим (все адаптеры Ethernet). На начальном этапе модуль не может знать какое перед ним **логическое** устройство, и на этой **1-й ступени** инициализации модуль должен:

- Соотнести идентификацию **физического** устройства (VID:PID) с поддерживаемым этим модулем набором поддерживаемых устройств;

- Извлечь непосредственно из устройства (например, из конфигурационной области устройства PCI) характерные параметры устройства: линию IRQ, адреса портов ввода вывода для DMA, характерные режимы обмена с концевыми точками (EP) устройства USB, ...

- Произвести **регистрацию** устройства, главным действием которой будет запись 2-х адресов функций обратного вызова для инициализации устройства, для активации устройства и его деактивации (функции `probe` и `remove` в структуре `struct pci_driver` для PCI устройства, вызываемые при **загрузке и выгрузке модуля**, и функции `probe` и `disconnect` в структуре struct usb_driver для USB устройства, вызываемые при горячем **подключении и отключении** USB устройства).

На этом, собственно 1-я ступень работы с устройством завершается. Следующая, **2-я ступень** инициализации производится уже изнутри кода функции probe, которая теперь уже будет вызвана обязательно в нужный момент. На этом этапе модуль:

- Проделывает все необходимые инициализации: подключает устройство к линии IRQ, определяет функцию (функции) обработки прерываний и т.п.

- Но главное действие на этой ступени состоит в том, что модуль (или его автор), зная функциональное предназначение **физического** устройства (из техдокументации устройства), регистрирует соответствующее ему символьное или блочное устройство, или сетевой интерфейс.

- Это происходит в точности так, как мы это рассматривали ранее при рассмотрении драйверов устройств. Вот на этой ступени и происходит создание имени устройства в `/dev`, если это необходимо, и назначение пары номеров `major` и `minor`.

- Главным итогом этой ступени является определение всех операций ввода-вывода, требуемых таким устройством, например: `read()`, `write()`, `ioctl()` для символьного устройства.

На этом фактически заканчивается 2-я ступень инициализации устройства в драйвере. Все остальные действия описываются и реализуются на **3-й стадии**, **внутри обменных функций** (определённых на предыдущей ступени). Только здесь вступает в игру технические спецификации самого устройства, алгоритмы его обмена, записи-чтения портов, организация работы DMA, обменные операции с концевыми точками (EP) USB устройства. Это уже совершенно специфические операции, полностью определяемые спецификациями устройства.

В любом случае, работа над драйвером устройства начинается с детального анализа оборудование и формирования плана реализации последовательности операций.

Анализ оборудования

Первым шагом всякой разработки, работающей непосредственно с образцом оборудования, является уточнение того, как это оборудование видится со стороны системы, и соответствует ли это видение тому, как мы понимаем это оборудование. Целью анализа, обычно производимого предварительно, перед написанием модуля драйвера, является уточнение специфических **численных параметров** тех или иных образцов оборудования, подготовка исходных данных. Общеизвестные команды для этих целей, это, например, `lspci` и `lsusb`, о которых мы будем вспоминать далее подробно и обстоятельно. Тем не менее, пока мы не подошли к их плотному использованию, они стоят хотя бы минимального отдельного упоминания...

Команда `lspci` перечисляет все устройства, распознанные на внутренней шине обмена PCI:

```
$ lspci
00:00.0 Host bridge: Intel Corporation Mobile 945GM/PM/GMS, 943/940GML and 945GT Express Memory Controller Hub (rev 03)
00:02.0 VGA compatible controller: Intel Corporation Mobile 945GM/GMS, 943/940GML Express Integrated Graphics Controller (rev 03)
00:02.1 Display controller: Intel Corporation Mobile 945GM/GMS/GME, 943/940GML Express Integrated Graphics Controller (rev 03)
...
02:06.0 CardBus bridge: Texas Instruments PCIxx12 Cardbus Controller
02:06.2 Mass storage controller: Texas Instruments 5-in-1 Multimedia Card Reader (SD/MMC/MS/MS PRO/xD)
02:06.3 SD Host controller: Texas Instruments PCIxx12 SDA Standard Compliant SD Host Controller
02:06.4 Communication controller: Texas Instruments PCIxx12 GemCore based SmartCard controller
02:0e.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5788 Gigabit Ethernet (rev 03)
08:00.0 Network controller: Intel Corporation PRO/Wireless 3945ABG [Golan] Network Connection (rev 02)
```

По каждому устройству перечисляются его производитель (например, Broadcom Corporation) и модель этого устройства в терминологии этого производителя (например, NetXtreme BCM5788 Gigabit Ethernet). Зачастую, при работе с драйвером, нас будут интересовать не текстуальные описания производителя и модели, а их численные коды, что **для того же набора устройств** выглядит так:

```
$ lspci -n
00:00.0 0600: 8086:27a0 (rev 03)
```

```
00:02.0 0300: 8086:27a2 (rev 03)
00:02.1 0380: 8086:27a6 (rev 03)
...
02:06.0 0607: 104c:8039
02:06.2 0180: 104c:803b
02:06.3 0805: 104c:803c
02:06.4 0780: 104c:803d
02:0e.0 0200: 14e4:169c (rev 03)
08:00.0 0280: 8086:4222 (rev 02)
```

Первый из этих параметров (численный индекс производителя) называют VID (Vendor ID), а второй — PID (Product ID), или иногда DID (Device ID). Мы о них будем ещё неоднократно говорить. Достаточно часто нужна диагностика устройств PCI по иерархии их подключения (о чём мы детально будем говорить вскоре):

```
$ lspci -t
-[0000:00]-+-00.0
           +-02.0
           +-02.1
           +-1b.0
           +-1c.0-[08]----00.0
           +-1d.0
           +-1d.1
           +-1d.2
           +-1d.3
           +-1d.7
           +-1e.0-[02-06]--+-06.0
           |               +-06.2
           |               +-06.3
           |               +-06.4
           |               \-0e.0
           +-1f.0
           +-1f.1
           \-1f.2
```

Возможности утилиты `lspci` весьма обширны, посмотреть их можно выполнив команду в недозволенном синтаксисе, например, так:

```
$ lspci --help
lspci: invalid option -- '-'
Usage: lspci [<switches>]
...
Resolving of device ID's to names:
-n             Show numeric ID's
-nn            Show both textual and numeric ID's (names & numbers)
-q             Query the PCI ID database for unknown ID's via DNS
-qq            As above, but re-query locally cached entries
-Q             Query the PCI ID database for all ID's via DNS
...
```

Здесь есть чрезвычайно замысловатые (и полезные!) опции, как, например, те, которые оставлены в части приведённого листинга. По любому устройству мы можем запросить диагностику высокой (`-v`) или самой высокой (`-vv`) степени детализации:

```
$ lspci -d14e4:169c -v
02:0e.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5788 Gigabit Ethernet (rev 03)
        Subsystem: Hewlett-Packard Company Device 30aa
        Flags: bus master, 66MHz, medium devsel, latency 64, IRQ 16
        Memory at e8110000 (32-bit, non-prefetchable) [size=64K]
        Expansion ROM at <ignored> [disabled]
        Capabilities: <access denied>
        Kernel driver in use: tg3
```

Другой «незаменимой» командой, к которой мы будем неоднократно обращаться при анализе сменных устройств на линиях USB, будет `lsusb`:

```
$ lsusb
Bus 001 Device 002: ID 0424:2503 Standard Microsystems Corp. USB 2.0 Hub
Bus 001 Device 003: ID 1a40:0101 TERMINUS TECHNOLOGY INC. USB-2.0 4-Port HUB
Bus 001 Device 005: ID 046d:080f Logitech, Inc. Webcam C120
Bus 004 Device 002: ID 046d:c517 Logitech, Inc. LX710 Cordless Desktop Laser
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 006: ID 03f0:171d Hewlett-Packard Wireless (Bluetooth + WLAN) Interface [Integrated
Module]
Bus 001 Device 007: ID 08ff:2580 AuthenTec, Inc. AES2501 Fingerprint Sensor
Bus 001 Device 009: ID 16d5:6502 AnyDATA Corporation CDMA/UMTS/GPRS modem
```

Команда `lsusb` во многом похожа по своей функциональности на названную выше `lspci` :

```
$ lsusb -t
/:  Bus 05.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
/:  Bus 04.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
    |__ Port 1: Dev 2, If 0, Class=HID, Driver=usbhid, 1.5M
    |__ Port 1: Dev 2, If 1, Class=HID, Driver=usbhid, 1.5M
/:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci_hcd/8p, 480M
    |__ Port 1: Dev 2, If 0, Class=hub, Driver=hub/3p, 480M
        |__ Port 1: Dev 6, If 0, Class='bInterfaceClass 0xe0 not yet handled', Driver=btusb, 12M
        |__ Port 1: Dev 6, If 1, Class='bInterfaceClass 0xe0 not yet handled', Driver=btusb, 12M
        |__ Port 1: Dev 6, If 2, Class=vend., Driver=, 12M
        |__ Port 1: Dev 6, If 3, Class=app., Driver=, 12M
        |__ Port 2: Dev 7, If 0, Class=vend., Driver=, 12M
    |__ Port 4: Dev 3, If 0, Class=hub, Driver=hub/4p, 480M
        |__ Port 4: Dev 9, If 0, Class=vend., Driver=option, 12M
        |__ Port 4: Dev 9, If 1, Class=vend., Driver=option, 12M
        |__ Port 4: Dev 9, If 2, Class=vend., Driver=option, 12M
        |__ Port 4: Dev 9, If 3, Class=stor., Driver=usb-storage, 12M
    |__ Port 6: Dev 5, If 0, Class='bInterfaceClass 0x0e not yet handled', Driver=uvcvideo, 480M
    |__ Port 6: Dev 5, If 1, Class='bInterfaceClass 0x0e not yet handled', Driver=uvcvideo, 480M
```

Разобраться **в деталях** листингов `lspci` и `lsusb`, показанных выше, не так просто, но это и не входит в наши планы — эти листинги являются основным подспорьем при разработке аппаратных драйверов, и читать их детали в **конкретном окружении** становится естественно и просто...

Но, в отношении анализа всего установленного в системе оборудования (начиная с анализа изготовителя и состава BIOS) существует достаточно много команд «редкого применения», которые часто помнят только заматерелые системные администраторы, и которые не попадают в справочные руководства. Все такие команды, в большинстве, требует прав `root`, кроме того, некоторые из них могут присутствовать в некоторых дистрибутивах Linux, но отсутствовать в других (и тогда их просто нужно доустановить с помощью менеджера программных пакетов). Информация от этих команд в какой-то мере дублирует друг друга, но только частично. Все такие команды результатом своего выполнения производят очень обширный объём вывода, поэтому его бессмысленно анализировать с экрана, а нужно поток вывода перенаправить в текстовый файл, в качестве журнала работы команды, для последующего изучения.

Сбор информации об оборудовании может стать ключевой позицией при работе над драйверами периферийных устройств. Ниже приводится только краткое перечисление (в порядке справки-напоминания) некоторых подобных команд (и несколько начальных строк их вывода, для идентификации того, что это именно та команда) — более детальное обсуждение увело бы нас слишком далеко от наших целей. Вот некоторые такие

команды:

```
$ time sudo lshw > lshw.lst
real    0m5.545s
user    0m5.029s
sys     0m0.193s
$ cat > lshw.lst
notebook.localdomain
    description: Notebook
    product: HP Compaq nc6320 (ES527EA#ACB)
    vendor: Hewlett-Packard
...
```

Примечание: Обратите внимание, что показанная команда выполняется достаточно долго, это не должно вам смущать.

Ещё несколько полезных команд из той же группы:

```
$ lshal
Dumping 162 device(s) from the Global Device List:
-------------------------------------------------
udi = '/org/freedesktop/Hal/devices/computer'
  info.addons = {'hald-addon-acpi'} (string list)
...
$ sudo dmidecode
# dmidecode 2.10
SMBIOS 2.4 present.
23 structures occupying 1029 bytes.
Table at 0x000F38EB.
...
```

Последняя команда, как пример, в том числе, даёт и детальную информацию о банках памяти, и какие модули оперативной памяти куда установлены.

Для разработчиков **блочных** устройств представляет интерес пакет `smartctl` (предустановлен почти в любом дистрибутиве), предоставляющий детальную информацию по дисковому накопителю:

```
$ sudo smartctl -A /dev/sda
smartctl 5.39.1 2010-01-28 r3054 [i386-redhat-linux-gnu] (local build)
Copyright (C) 2002-10 by Bruce Allen, http://smartmontools.sourceforge.net

=== START OF READ SMART DATA SECTION ===
SMART Attributes Data Structure revision number: 16
Vendor Specific SMART Attributes with Thresholds:
ID# ATTRIBUTE_NAME          FLAG     VALUE WORST THRESH TYPE      UPDATED  WHEN_FAILED RAW_VALUE
  1 Raw_Read_Error_Rate     0x000f   100   100   046    Pre-fail  Always       -       49961
  2 Throughput_Performance  0x0005   100   100   030    Pre-fail  Offline      -       15335665
  3 Spin_Up_Time            0x0003   100   100   025    Pre-fail  Always       -       1
  4 Start_Stop_Count        0x0032   098   098   000    Old_age   Always       -       7320
...
```

Ещё одна утилита, широко используемая при отработке блочных устройств (любых: будь то диск, USB накопитель, или RAM диск) — `hdparm` :

```
$ sudo hdparm -i /dev/sda
/dev/sda:
 Model=WDC WD2500AAKX-001CA0, FwRev=15.01H15, SerialNo=WD-WMAYU0425651
 Config={ HardSect NotMFM HdSw>15uSec SpinMotCtl Fixed DTR>5Mbs FmtGapReq }
 RawCHS=16383/16/63, TrkSize=0, SectSize=0, ECCbytes=50
 BuffType=unknown, BuffSize=16384kB, MaxMultSect=16, MultSect=16
 CurCHS=16383/16/63, CurSects=16514064, LBA=yes, LBAsects=488397168
 IORDY=on/off, tPIO={min:120,w/IORDY:120}, tDMA={min:120,rec:120}
 PIO modes:  pio0 pio3 pio4
```

```
DMA modes:  mdma0 mdma1 mdma2
UDMA modes: udma0 udma1 udma2 udma3 udma4 udma5 *udma6
AdvancedPM=no WriteCache=enabled
Drive conforms to: Unspecified:  ATA/ATAPI-1,2,3,4,5,6,7
```

Подсистема udev

Подсистема udev — это подсистема создания именованных устройств, которая пришла на смену devfs (или даже статическому отображению устройств в каталог /dev, доставшемуся раннему Linux от общей философии UNIX систем). Более того, udev создаёт в /dev имена **динамически** (по мере подключения), и только для тех устройств, которые реально присутствуют на данный момент в системе. Подсистема udev является надстройкой пространства пользователя над файловой системой ядра /sys (которую мы достаточно детально разбирали раньше). Задача ядра определять изменения в аппаратной конфигурации системы (например, для устройств горячего подключения и USB), регистрировать эти изменения, и вносить изменения в каталог /sys. Задача подсистемы udev — выполнить дальнейшую интеграцию и настройку такого устройства в системе (отобразить его в каталоге /dev), и предоставить пользователю уже готовое к работе устройство.

Подсистема udev — это обширная надстройка пользовательского уровня, инструмент из области администрирования системы Linux, и **детальное** её рассмотрение увело бы нас очень далеко от намерений нашего рассмотрения. Но не упоминание udev, в контексте настройки устройств, используемых системой, создало бы, для работающего над драйвером, ложную картину в ещё большей мере. Поэтому мы проделаем беглый экскурс в структуру udev, в мере, достаточной программисту для настройки драйверов.

Асинхронные уведомления от ядра о любых изменениях в /sys посылаются ядром через дэйтаграммный сокет, **специально** для этого случая спроектированного вида обмена — netlink:

```
fd = socket( AF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT );
```

Получив такое уведомление от ядра демон udevd (но это может делать и **любой**, ваш собственный, процесс пространства пользователя) имеет возможность проанализировать дерево /sys (на основе полученных из содержимого дэйтаграммного уведомления **параметров**), а далее **динамически** (по событию) создать соответствующее имя в /dev. Всё достаточно просто. Более того, поскольку всё это (после получения уведомления) происходит в пространстве пользователя (не в ядре), то при создании имени в /dev можно применить достаточно развитую систему **правил**, формируемых пользователем (в текстовых файлах), и определяющих характер создаваемого имени устройства.

Подсистема udev настраивает устройства в соответствии с заданными правилами. Правила содержаться в файлах каталога /etc/udev/rules.d/ (также файлы с правилами могут содержаться и в каталоге /etc/udev/). Все файлы правил просматриваются (отрабатываются) в **алфавитном** порядке (лексографическом — вот почему традицией стало для однозначности именовать такие файлы начиная с численного префикса). Вот возможное содержимое (Fedora 15):

```
$ cat /etc/system-release
RFRemix release 15.1 (Lovelock)
$ ls -w80 /etc/udev/rules.d/
10-vboxdrv.rules             70-persistent-net.rules
40-hplip.rules               80-kvm.rules
56-hpmud_support.rules       90-alsa-tools-firmware.rules
60-fprint-autosuspend.rules  90-hal.rules
60-madwimax.rules            91-drm-modeset.rules
60-sysprof.rules             97-bluetooth-serial.rules
70-persistent-cd.rules       99-fuse.rules
```

Содержимое файлов правил может быть достаточно разнообразным, но оно записывается исходя из весьма ограниченного набора синтаксических **правил действий** для **событий** в /sys:

- ✔ Каждое правило записывается отдельной строкой (даже очень длинной), не предусмотрены способы переноса строки;

- ✔ Если выполняются **все условия** (ключ соответствия, match key — они описываются с знаком ==) возникшего события то предпринимаются действия;

- ✔ Действия, описанные в **правиле** (ключ назначения, assignment key — они описываются с знаком =) предписывают выполнение некоторых операций: изменение флагов доступа (`MODE="0600"`), выполнение программы **до** создания имени устройства (`PROGRAM="..."`), запуск программы **после** создания имени устройства (`RUN="..."`) и другие.

- ✔ Все параметры (значения) в условиях и правилах заключаются в кавычки: `ACTION=="remove"`

- ✔ В записи условий и правил допускаются шаблоны: `KERNEL=="controlD[0-9]*"`, `NAME="dri/%k"` , ...

- ✔ Всё касательно синтаксиса правил `udev` исчерпывающе описано на справочной странице:

```
$ man 7 udev
UDEV(7)                              udev                              UDEV(7)
NAME
       udev - dynamic device management
...
```

Вот примеры несколько реальных файлов правил (из разных систем):

```
$ cat 91-drm-modeset.rules
KERNEL=="controlD[0-9]*", NAME="dri/%k", MODE="0600"
$ cat 60-raw.rules
...
# An example would be:
ACTION=="add", KERNEL=="sda", RUN+="/bin/raw /dev/raw/raw1 %N"
...
```

Основной объём потребностей разработчика по работе с `udev` покрывает не очень широко известная команда `udevadm` с огромным множеством параметров и опций:

```
$ udevadm info -q path -n sda
/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda
$ udevadm info -a -p $(udevadm info -q path -n sda)
...
looking at device '/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda':
    KERNEL=="sda"
    SUBSYSTEM=="block"
...
$ udevadm info -h
Usage: udevadm info OPTIONS
  --query=<type>             query device information:
      name                     name of device node
      symlink                  pointing to node
      path                     sys device path
      property                 the device properties
      all                      all values
  --path=<syspath>           sys device path used for query or attribute walk
  --name=<name>              node or symlink name used for query or attribute walk
...
```

Ключевой вопрос: как определить значения параметров, используемых в записи правил для подключения и отключения конкретного устройства, для которого отрабатывается драйвер? Их можно увидеть при выполнении физического подключения-выключения этого устройства **при работающей программе** мониторинга уведомлений:

```
$ udevadm monitor --property --kernel
monitor will print the received events for:
KERNEL - the kernel uevent
```

```
KERNEL[27478.580340] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-4/1-4.4 (usb)
ACTION=add
BUSNUM=001
DEVNAME=/dev/bus/usb/001/035
DEVNUM=035
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-4/1-4.4
DEVTYPE=usb_device
MAJOR=189
MINOR=34
PRODUCT=1307/163/100
SEQNUM=3186
SUBSYSTEM=usb
TYPE=0/0/0

KERNEL[27478.580711] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-4/1-4.4/1-4.4:1.0 (usb)
ACTION=add
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-4/1-4.4/1-4.4:1.0
DEVTYPE=usb_interface
INTERFACE=8/6/80
MODALIAS=usb:v1307p0163d0100dc00dsc00dp00ic08isc06ip50
PRODUCT=1307/163/100
SEQNUM=3187
SUBSYSTEM=usb
TYPE=0/0/0
...
```

Здесь показан только начальный вывод, охватывающий только **два** асинхронных уведомления от ядра (для реальных USB, например, устройств их может следовать значительно больше: 5, 10 и более — фазы установки устройства). Представленные строки и отображают имена и значения параметров, посылаемых в уведомлении, например: `ACTION=add` — происходит **подключение** устройства (могло бы быть отключение); `MAJOR=189`, `MINOR=34` — старший и младший номера устройства в `/dev`, о которых мы уже так много знаем; `PRODUCT=1307/163/100` — индексы VID:PID для подключаемого устройства, о которых упоминалось в предыдущей части. Именно из этих **именованных** параметров и их значений конструируются те текстовые файлы параметров, которые вносятся в каталог `/etc/udev/rules.d/`, и которые **предписывают** выполнить те или иные действия при поступления уведомления от ядра с совпадающими параметрами.

В архивах предлагается пример `udev.tgz` (мы не будем его подробно комментировать), не имеющий прямого отношения к коду модуля, но демонстрирующий, как **любой произвольный** пользовательский процесс может получать асинхронные уведомления о событиях ядра (приложение `mondev` в архиве регистрирует уведомления о горячих подключениях-выключениях устройств, а приложение `mondev` — уведомления об изменениях в состояниях сетевых интерфейсов). Разбор таких приложений очень проясняют то, как драйверы устройств взаимодействуют с ядром.

Идентификация модуля

При установке новых устройств в системе часто нужно бывает идентифицировать то, каким модулем ядра будет поддерживаться устройство с заданной парой индексов VID:PID (это относится и к устройствам на шине PCI, и к USB устройствам). Используем для демонстрации внешний USB WiFi адаптер (WiFi «свисток») модели **Tenda W311M** (показан на рисунке):

```
$ lsusb | grep Wireless
Bus 001 Device 012: ID 148f:5370 Ralink Technology, Corp. RT5370 Wireless Adapter
```

```
$ modprobe -c | grep usb: | grep -i 148f | grep -i 5370
alias usb:v148Fp5370d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
```

Здесь мы (последним полем строки) получили имя модуля, который будет поддерживать устройство 148f:5370. Если для некоторого устройства неизвестен модуль поддержки, то вывод последней строки будет пустой. Естественно, подобным образом может быть получена информация по всем устройствам этого (или любого другого) производителя:

```
$ modprobe -c | grep usb: | grep -i 148f
alias usb:v148Fp1706d*dc*dsc*dp*ic*isc*ip*in* rt2500usb
alias usb:v148Fp2070d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp2570d*dc*dsc*dp*ic*isc*ip*in* rt2500usb
alias usb:v148Fp2573d*dc*dsc*dp*ic*isc*ip*in* rt73usb
alias usb:v148Fp2671d*dc*dsc*dp*ic*isc*ip*in* rt73usb
alias usb:v148Fp2770d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp2870d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3070d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3071d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3072d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3370d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3572d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3573d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp5370d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp5372d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp5572d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp8070d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp9020d*dc*dsc*dp*ic*isc*ip*in* rt2500usb
alias usb:v148Fp9021d*dc*dsc*dp*ic*isc*ip*in* rt73usb
alias usb:v148FpF101d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148FpF301d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
```

Эту же информацию о модуле поддержки можно получить и по-другому:

```
# cat /lib/modules/`uname -r`/modules.alias | grep usb: | grep -i 148f | grep -i 5370
alias usb:v148Fp5370d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
```

И, самое интересное, то как и откуда эта информация попадает в файл `modules.alias`:

```
$ modinfo rt2800usb
filename:       /lib/modules/3.13.6-200.fc20.i686/kernel/drivers/net/wireless/rt2x00/rt2800usb.ko
license:        GPL
firmware:       rt2870.bin
description:    Ralink RT2800 USB Wireless LAN driver.
version:        2.3.0
author:         http://rt2x00.serialmonkey.com
srcversion:     D6F814DAF78F2BEA3DA12CB
alias:          usb:vF201p5370d*dc*dsc*dp*ic*isc*ip*in*
alias:          usb:v177Fp0254d*dc*dsc*dp*ic*isc*ip*in*
alias:          usb:v083ApF511d*dc*dsc*dp*ic*isc*ip*in*
...
alias:          usb:v148Fp5370d*dc*dsc*dp*ic*isc*ip*in*
...
alias:          usb:v07B8p2870d*dc*dsc*dp*ic*isc*ip*in*
depends:        rt2x00lib,rt2800lib,rt2x00usb
intree:         Y
vermagic:       3.13.6-200.fc20.i686 SMP mod_unload 686
signer:         Fedora kernel signing key
sig_key:        72:23:0F:69:91:87:54:91:2A:09:46:5F:53:D5:ED:EE:0C:C0:71:99
sig_hashalgo:   sha256
parm:           nohwcrypt:Disable hardware encryption. (bool)
```

Здесь примечательно то, что это представлен драйвер чипсета Ralink 5370, а не какой-то конкретной модели

изделия, и этот драйвер поддерживает весьма много (несколько сот) самых различных моделей устройств от разных производителей, общее у которых то, что они собраны именно на таком чипсете:

```
$ modinfo rt2800usb | wc -l
338
```

Если ваше устройство идентифицировано модулем, то дальше такой модуль будет загружен (со всеми требуемыми зависимостями), и, например, в обсуждаемом случае появится новый сетевой интерфейс:

```
$ lsmod | grep rt2
rt2800usb              26581  0
rt2x00usb              19262  1 rt2800usb
rt2800lib              77341  1 rt2800usb
rt2x00lib              56555  3 rt2x00usb,rt2800lib,rt2800usb
crc_ccitt              12549  1 rt2800lib
mac80211              510326  5 iwl3945,iwlegacy,rt2x00lib,rt2x00usb,rt2800lib
cfg80211              400375  4 iwl3945,iwlegacy,mac80211,rt2x00lib
$ iwconfig
wlp8s0    IEEE 802.11bg  ESSID:off/any
          Mode:Managed  Access Point: Not-Associated   Tx-Power=off
          Retry  long limit:7   RTS thr:off   Fragment thr:off
          Power Management:off
lo        no wireless extensions.
enp2s14   no wireless extensions.
wlp0s29f7u4  IEEE 802.11bgn  ESSID:off/any
          Mode:Managed  Access Point: Not-Associated   Tx-Power=0 dBm
          Retry  long limit:7   RTS thr:off   Fragment thr:off
          Power Management:on
```

Последний из показанных и есть обсуждаемый сетевой интерфейс.

Ошибки идентификация модуля

Показанный выше процесс идентификации весьма продуктивен, особенно для интеграции новых, недавно появившихся моделей устройств. Но номенклатура доступных устройств постоянно расширяется, и не исключено, что ваше новое устройство получит поддержку неадекватного модуля, что не обеспечит его работоспособности. Образцом сказанного может быть сетевой адаптер Broadcom BCM43228, который активно используется в ноутбуках HP:

```
$ lspci | grep Broad
24:00.0 Network controller: Broadcom Corporation BCM43228 802.11a/b/g/n
$ lspci -n | grep 24:00.0
24:00.0 0280: 14e4:4359
```

После установки системы (Fedora 20, ядро 3.13) для сетевого адаптера устанавливается поддержка модулем `bcma` — общий модуль поддержки адаптеров 43xxx серии:

```
$ modprobe -c | grep -i 14e4 | grep 4359
alias pci:v000014E4d00004359sv*sd*bc*sc*i* bcma
$ lsmod | grep bcma
bcma 46142 0
```

Но отправившись на сайт производителя (Broadcom) мы узнаём, что модуль поддерживает адаптеры 43xxx серии, вплоть до 43227, а для поддержки 43228 нужно использовать **закрытый** модуль `wl`. К счастью, в большинстве случаев модули новых устройств очень быстро попадают в репозитарии дистрибутивов, откуда их можно и устанавливать:

```
$ sudo yum install kmod-wl
...
Установлено:
 kmod-wl.x86_64 0:6.30.223.141-5.fc20.14.....................................
Установлены зависимости:
 broadcom-wl.noarch 0:6.30.223.141-2.fc20.....................................
 kmod-wl-3.13.6-200.fc20.x86_64.x86_64 0:6.30.223.141-5.fc20.14..............
Выполнено!
```

```
$ sudo rmmod bcma
# echo "blacklist bcma" >> /etc/modprobe.d/blacklist.conf
# modprobe wl
```

После перезагрузки у нас имеет место совершенно другая конфигурация поддержки WiFi (работоспособный вариант):

```
$ lsmod | grep wl
wl                   4207671  0
cfg80211              513095  1 wl
lib80211               13968  2 wl,lib80211_crypt_tkip
$ iwconfig
lo        no wireless extensions.
em1       no wireless extensions.
wlo1      IEEE 802.11abg  ESSID:"ZTE"
          Mode:Managed  Frequency:2.462 GHz  Access Point: C8:64:C7:8A:50:16
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Power Management:off
```

Устройства на шине PCI

Архитектура шины PCI (Peripheral Component Interconnect) был разработана в качестве замены предыдущему стандарту ISA/EISA (Industry Standard Architecture) с тремя основными целями: а). получить лучшую производительность при передаче данных между компьютером и его периферией, б). быть независимой от платформы, насколько это возможно, и в). упростить добавление и удаление периферийных устройств в системе. Первоначальный стандарт PCI описывал параллельный обмен 32-битовыми данными на частоте 33MHz или 66MHz, обеспечивая пиковую производительность 266MBps. Следующее расширение, известное как PCI Extended (PCI-X), определяло шину до 64-бит, частоту до 133MHz и производительность в 1GBps. Стандарт PCI Express (PCIe или PCI-E) представляет семейство нового поколения. В отличие от PCI, PCIe использует последовательный протокол передачи данных. PCIe поддерживает максимально 32 последовательных линии (links), каждая из которых (в стандарте версии 1.1) поддерживает поток 250MBps в каждом направлении передачи, таким образом обеспечивая производительность до 8GBps в каждом направлении. Стандарт PCIe 2.0 предусматривает ещё большие скорости передачи.

Примечание: Последовательные каналы передачи, в отличие от того, что предполагалось на ранних периодах развития компьютерных технологий, обеспечивают более высокие скорости и устойчивость обмена, за счёт отсутствия эффекта интерференции сигнала (рассинхронизации). Поэтому, переход к последовательным протоколам обмена стал общей тенденцией стандартизации, примеры чему: PCIe, SATA, USB, FireWire...

Стандарты PCI, помимо сказанных вариантов, имеют ещё варианты, связанные с мобильными применениями: CardBus, Mini PCI, PCI Express Mini Card, Express Card (которые не имеют существенно принципиальных отличий). В настоящее время PCI широко используется на самых разных процессорных платформах: IA-32 / IA-64, Alpha, PowerPC, SPARC64 ...

Для разработчика драйверной поддержки PCI устройств всё это разнообразие стандартов не накладывает особых различий (может изменяться размер конфигурационной области, о чём будет далее). Поэтому, мы, в контексте нашего обсуждения, больше не будем делать различий PCI шинам.

Самой актуальной для автора драйвера является поддержка PCI автоопределения интерфейса плат: PCI устройства настраивается автоматически во время загрузки (это делается программами BSP — board support programm, поддержки аппаратной платформы, в случае x86 компьютера универсального назначения — эту функцию несут программы BIOS). Затем драйвер устройства получает доступ к информации о конфигурации устройства, и производит инициализацию. Это происходит без необходимости совершать какое-либо тестирование периода выполнения (как, например, в стандарте PnP для устройств ISA).

Каждое периферийное устройство PCI адресуется по подключению такими физическими параметрами, как: номер шины, номер устройства и номер функции. Linux дополнительно вводит и поддерживает такое

логическое понятие как домен PCI. Каждый домен PCI может содержать до 256 шин. Каждая шина содержит до 32 устройств, каждое устройство может быть многофункциональным и поддерживать до 8 функций. В конечном итоге, каждая функция может быть однозначно идентифицирована на аппаратном уровне 16-ти разрядным ключом. Однако, драйверам устройств в Linux, не требуется иметь дело с этими двоичными ключами, потому что они используют для работы с устройствами специальную структуру данных `pci_dev`.

Примечание: Часто то, что мы житейски и физически (плата PCI) понимаем как устройство, в этой системе терминологически правильно называется: функция, устройство же может содержать до 8-ми эквивалентных (по своим возможностям) функций (хорошим примером являются 2, 4, или 8 независимых интерфейсов E1/T1/J1 на PCI платах основных мировых производителей: Digium, Sangoma и других).

Адресацию PCI устройств в своей Linux системе смотрим:

```
$ lspci
00:00.0 Host bridge: Intel Corporation Mobile 945GM/PM/GMS, 943/940GML and 945GT Express Memory Controller Hub (rev 03)
00:02.0 VGA compatible controller: Intel Corporation Mobile 945GM/GMS, 943/940GML Express Integrated Graphics Controller (rev 03)
00:02.1 Display controller: Intel Corporation Mobile 945GM/GMS/GME, 943/940GML Express Integrated Graphics Controller (rev 03)
00:1b.0 Audio device: Intel Corporation 82801G (ICH7 Family) High Definition Audio Controller (rev 01)
00:1c.0 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 1 (rev 01)
00:1c.2 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 3 (rev 01)
00:1c.3 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 4 (rev 01)
00:1d.0 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #1 (rev 01)
00:1d.1 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #2 (rev 01)
00:1d.2 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #3 (rev 01)
00:1d.3 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #4 (rev 01)
00:1d.7 USB Controller: Intel Corporation 82801G (ICH7 Family) USB2 EHCI Controller (rev 01)
00:1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev e1)
00:1f.0 ISA bridge: Intel Corporation 82801GBM (ICH7-M) LPC Interface Bridge (rev 01)
00:1f.2 IDE interface: Intel Corporation 82801GBM/GHM (ICH7 Family) SATA IDE Controller (rev 01)
02:06.0 CardBus bridge: Texas Instruments PCIxx12 Cardbus Controller
02:06.1 FireWire (IEEE 1394): Texas Instruments PCIxx12 OHCI Compliant IEEE 1394 Host Controller
02:06.2 Mass storage controller: Texas Instruments 5-in-1 Multimedia Card Reader (SD/MMC/MS/MS PRO/xD)
02:06.3 SD Host controller: Texas Instruments PCIxx12 SDA Standard Compliant SD Host Controller
02:06.4 Communication controller: Texas Instruments PCIxx12 GemCore based SmartCard controller
02:0e.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5788 Gigabit Ethernet (rev 03)
08:00.0 Network controller: Intel Corporation PRO/Wireless 3945ABG [Golan] Network Connection (rev 02)
```

Особенно полезным может оказаться использование уточняющих (расширяющих) опций команды lspci, например, так:

```
$ lspci -vk
...
00:1b.0 Audio device: Intel Corporation 82801G (ICH7 Family) High Definition Audio Controller (rev 01)
        Subsystem: Hewlett-Packard Company Device 30aa
        Flags: bus master, fast devsel, latency 0, IRQ 21
        Memory at e8580000 (64-bit, non-prefetchable) [size=16K]
        Capabilities: <access denied>
        Kernel driver in use: HDA Intel
        Kernel modules: snd-hda-intel
...
00:1f.2 IDE interface: Intel Corporation 82801GBM/GHM (ICH7 Family) SATA IDE Controller (rev 01) (prog-if 80 [Master])
        Subsystem: Hewlett-Packard Company Device 30aa
        Flags: bus master, 66MHz, medium devsel, latency 0, IRQ 17
        I/O ports at 01f0 [size=8]
```

```
        I/O ports at 03f4 [size=1]
        I/O ports at 0170 [size=8]
        I/O ports at 0374 [size=1]
        I/O ports at 60a0 [size=16]
        Capabilities: <access denied>
        Kernel driver in use: ata_piix

02:06.0 CardBus bridge: Texas Instruments PCIxx12 Cardbus Controller
        Subsystem: Hewlett-Packard Company Device 30aa
        Flags: bus master, medium devsel, latency 168, IRQ 18
        Memory at e8100000 (32-bit, non-prefetchable) [size=4K]
        Bus: primary=02, secondary=03, subordinate=06, sec-latency=176
        Memory window 0: 80000000-83fff000 (prefetchable)
        Memory window 1: 88000000-8bfff000
        I/O window 0: 00003000-000030ff
        I/O window 1: 00003400-000034ff
        16-bit legacy interface ports at 0001
        Kernel driver in use: yenta_cardbus
        Kernel modules: yenta_socket
...
```

Здесь мы информацию по тем же устройствам получаем в развёрнутом виде (а поэтому её часто избыточно много), здесь представлены и технические параметры устройств (порты ввода-вывода, линии IRQ), и имена поддерживающих устройства модулей ядра (драйверов).

Различие между понятиями устройства и функции PCI хорошо заметно на примере (выше) многофункционального устройства производителя Texas Instruments с номером устройства 6 на шине 2, которое представляет из себя объединение пяти функций: 0...4, например, функция 02:06.3 представляет из себя оборудование чтения SD-карт, и поддерживается отдельным модулем ядра, создающим соответствующие имена устройств вида:

```
$ ls /dev/mm*
/dev/mmcblk0  /dev/mmcblk0p1
```

Другое представление той же адресной информации (тот же хост, та же конфигурация) можем получить так:

```
$ tree /sys/bus/pci/devices/
/sys/bus/pci/devices/
├── 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
├── 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:02.0
├── 0000:00:02.1 -> ../../../devices/pci0000:00/0000:00:02.1
├── 0000:00:1b.0 -> ../../../devices/pci0000:00/0000:00:1b.0
├── 0000:00:1c.0 -> ../../../devices/pci0000:00/0000:00:1c.0
├── 0000:00:1c.2 -> ../../../devices/pci0000:00/0000:00:1c.2
├── 0000:00:1c.3 -> ../../../devices/pci0000:00/0000:00:1c.3
├── 0000:00:1d.0 -> ../../../devices/pci0000:00/0000:00:1d.0
├── 0000:00:1d.1 -> ../../../devices/pci0000:00/0000:00:1d.1
├── 0000:00:1d.2 -> ../../../devices/pci0000:00/0000:00:1d.2
├── 0000:00:1d.3 -> ../../../devices/pci0000:00/0000:00:1d.3
├── 0000:00:1d.7 -> ../../../devices/pci0000:00/0000:00:1d.7
├── 0000:00:1e.0 -> ../../../devices/pci0000:00/0000:00:1e.0
├── 0000:00:1f.0 -> ../../../devices/pci0000:00/0000:00:1f.0
├── 0000:00:1f.2 -> ../../../devices/pci0000:00/0000:00:1f.2
├── 0000:02:06.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:06.0
├── 0000:02:06.1 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:06.1
├── 0000:02:06.2 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:06.2
├── 0000:02:06.3 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:06.3
├── 0000:02:06.4 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:06.4
├── 0000:02:0e.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:0e.0
└── 0000:08:00.0 -> ../../../devices/pci0000:00/0000:00:1c.0/0000:08:00.0
```

Здесь отчётливо видно (слева) поля, например для контроллера VGA это: 0000:00:02.0 - выделены домен (16 бит), шина (8 бит), устройство (5 бит) и функция (3 бита). Поэтому, когда мы говорим о конкретном устройстве поддерживаемом модулем (далее), мы часто имеем в виду полный набор: номера домена + номер шины + номер устройства + номер функции.

С другой стороны[17], каждое устройство по типу идентифицируется двумя индексами: индекс производителя (Vendor ID) и индекс типа устройства (Device ID). Эта пара однозначно идентифицирует тип устройства. Использование 2-х основных идентификаторов устройств PCI (Vendor ID : Device ID) глобально регламентировано, и их актуальный перечень поддерживается в файле `pci.ids`, последнюю по времени копию которого можно найти в нескольких местах интернет, например по URL: http://pciids.sourceforge.net/. Эти два параметра являются уникальным (среди всех устройств в мире) ключом поиска устройств, установленных на шине PCI. Идентификация устройства парой Vendor ID : Device ID это константа, неизменно закреплённая за устройством. Адресная же идентификация устройства (шина : устройство : функция) величина изменяющаяся в зависимости от того: в какой конфигурации компьютера (в каком экземпляре компьютера) используется устройство, в какой PCI-слот установлено устройство, и даже от того, какие другие устройства, помимо интересующего нас, устанавливаются или извлекаются из компьютера. Однозначно связать идентификацию VID:DID с адресной идентификацией устройства — является одной из первейших задач модуля-драйвера.

Поиск устройств в программном коде модуля, установленных на шине PCI делается **циклическим перебором** (перечислением, enumeration) всех установленных устройств по определённым критериям поиска. Для поиска (перебора устройств, установленных на шине PCI) в программном коде модуля в цикле используется итератор:

```
struct pci_dev *pci_get_device( unsigned int vendor, unsigned int device, struct pci_dev *from );
```

Здесь `from` — это `NULL` при начале поиска (или возобновлении поиска с начала), или указатель устройства, найденного на предыдущем шаге поиска. Если в качестве Vendor ID и/или Device ID указана константа с символьным именем `PCI_ANY_ID=-1`, то предполагается перебор всех доступных устройств с таким идентификатором. Если искомое устройство не найдено (или больше таких устройств не находится в цикле), то очередной вызов возвратит `NULL`. Если возвращаемое значение не `NULL`, то возвращается указатель структуры описывающей устройство, и счётчик использования для устройства инкрементируется. Когда устройство удаляется (модуль выгружается) для декремента этого счётчика использования необходимо вызвать:

```
void pci_dev_put( struct pci_dev *dev );
```

Примечание: Эта процедура весьма напоминает использование POSIX API в пространстве пользователя для работы с именами, входящими в данный каталог: все имена перебираются последовательно, пока результат очередного перебора на станет `NULL`.

После нахождения устройства, но прежде начала его использования необходимо разрешить использование устройства вызовом: `pci_enable_device( struct pci_dev *dev )`, часто это выполняется в функции инициализации устройства: поле `probe` структуры `struct pci_driver` (см. далее), но может выполняться и автономно в коде драйвера.

Каждое найденное устройство имеет своё пространство конфигурации, значения которого заполнены программами BIOS (или PnP OS, или программами BSP) — важно, что на момент загрузки модуля эта конфигурационное пространство всегда заполнено, и может только читаться (не записываться). Пространство конфигурации PCI устройства состоит из 256 байт для каждой функции устройства (для устройств PCI Express расширено до 4 Кб конфигурационного пространства для каждой функции) и стандартизированную схему регистров конфигурации. Четыре начальных байта конфигурационного пространства должны содержать уникальный ID функции (байты 0-1 — Vendor ID, байты 2-3 — Device ID), по которому драйвер идентифицирует своё устройство. Вот для сравнения начальные строки вывода команды для того же хоста (видно, через двоеточие, пары: Vendor ID — Device ID):

```
$ lspci -n
00:00.0 0600: 8086:27a0 (rev 03)
00:02.0 0300: 8086:27a2 (rev 03)
00:02.1 0380: 8086:27a6 (rev 03)
00:1b.0 0403: 8086:27d8 (rev 01)
```

17 Нужно чётко различать **адресацию** и **идентификацию** PCI устройства. При перестановке устройства в другой разъём PCI его адресация изменится, но идентификация является константным параметром данного устройства.

```
00:1c.0 0604: 8086:27d0 (rev 01)
00:1c.2 0604: 8086:27d4 (rev 01)
...
```

Первые 64 байт конфигурационной области стандартизованы, остальные зависят от устройства. Самыми актуальными для нас являются (кроме ID описанного выше) поля по смещению:

```
0x10 – Base Address 0
0x14 – Base Address 1
0x18 – Base Address 2
0x1C – Base Address 3
0x20 – Base Address 4
0x24 – Base Address 5
0x3C – IRQ Line
0x3D – IRQ Pin
```

Вся регистрация устройства PCI и связывание его параметров с кодом модуля должны происходить **исключительно** через значения, считанные из конфигурационного пространства устройства. Обработку конфигурационной информации показывает модуль (архив `pci.tgz`) `lab2_pci.ko` (заимствовано из [6]):

lab2_pci.c :

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

static int __init my_init( void ) {
   u16 dval;
   char byte;
   int j = 0;
   struct pci_dev *pdev = NULL;
   printk( KERN_INFO "LOADING THE PCI_DEVICE_FINDER\n" );
   /* either of the following looping constructs will work */
   for_each_pci_dev( pdev ) {
      /*    while ( ( pdev = pci_get_device
                     ( PCI_ANY_ID, PCI_ANY_ID, pdev ) ) ) { */
      printk( KERN_INFO "\nFOUND PCI DEVICE # j = %d, ", j++ );
      printk( KERN_INFO "READING CONFIGURATION REGISTER:\n" );
      printk( KERN_INFO "Bus,Device,Function=%s", pci_name( pdev ) );
      pci_read_config_word( pdev, PCI_VENDOR_ID, &dval );
      printk( KERN_INFO " PCI_VENDOR_ID=%x", dval );
      pci_read_config_word( pdev, PCI_DEVICE_ID, &dval );
      printk( KERN_INFO " PCI_DEVICE_ID=%x", dval );
      pci_read_config_byte( pdev, PCI_REVISION_ID, &byte );
      printk( KERN_INFO " PCI_REVISION_ID=%d", byte );
      pci_read_config_byte( pdev, PCI_INTERRUPT_LINE, &byte );
      printk( KERN_INFO " PCI_INTERRUPT_LINE=%d", byte );
      pci_read_config_byte( pdev, PCI_LATENCY_TIMER, &byte );
      printk( KERN_INFO " PCI_LATENCY_TIMER=%d", byte );
      pci_read_config_word( pdev, PCI_COMMAND, &dval );
      printk( KERN_INFO " PCI_COMMAND=%d\n", dval );
      /* decrement the reference count and release */
      pci_dev_put( pdev );
   }
   return 0;
}

static void __exit my_exit( void ) {
   printk( KERN_INFO "UNLOADING THE PCI DEVICE FINDER\n" );
}
```

```
module_init( my_init );
module_exit( my_exit );

MODULE_AUTHOR( "Jerry Cooperstein" );
MODULE_DESCRIPTION( "LDD:1.0 s_22/lab2_pci.c" );
MODULE_LICENSE( "GPL v2" );
```

Рассмотрение кода этого примера позволяет сформулировать ряд полезных утверждений:

- поскольку операция перечисления устройств PCI производится часто, то для её записи сконструирован специальный макрос `for_each_pci_dev()`, следом за ним в коде, комментарием, показано его раскрытие в виде явного цикла;
- конфигурационные параметры никогда не читаются напрямую, по их смещениям — для этого существуют (определены в `<linux/pci.h>`) инлайново определённых вызовах вида `pci_read_config_byte()`, `pci_read_config_word()` и подобных ним;
- конкретный вид считываемого конфигурационного параметра задаётся символьными константами вида `PCI_*` (определены там же), указываемыми как 2-й параметр макроса;
- результат (значение конфигурационного параметра) возвращается в виде побочного эффекта в 3-й параметр макроса.

И теперь мы можем рассмотреть небольшой начальный фрагмент результата выполнения написанного выше модуля (весь результат может быть очень объёмным):

```
$ sudo insmod lab2_pci.ko
$ lsmod | grep lab
lab2_pci                822  0
$ dmesg | tail -n221 | head -n30
LOADING THE PCI_DEVICE_FINDER

FOUND PCI DEVICE # j = 0,
READING CONFIGURATION REGISTER:
Bus,Device,Function=0000:00:00.0
 PCI_VENDOR_ID=8086
 PCI_DEVICE_ID=27a0
 PCI_REVISION_ID=3
 PCI_INTERRUPT_LINE=0
 PCI_LATENCY_TIMER=0
 PCI_COMMAND=6

FOUND PCI DEVICE # j = 1,
 READING CONFIGURATION REGISTER:
 Bus,Device,Function=0000:00:02.0
 PCI_VENDOR_ID=8086
 PCI_DEVICE_ID=27a2
 PCI_REVISION_ID=3
 PCI_INTERRUPT_LINE=10
 PCI_LATENCY_TIMER=0
 PCI_COMMAND=7
...
$ sudo rmmod lab2_pci
$ lsmod | grep lab2
$
```

К этому моменту рассмотрения мы разобрались, хотелось бы надеяться, с тем как перечисляются PCI устройства в системе, и как извлекаются их параметры из области конфигурации. Теперь нас должен интересовать вопрос: как это использовать в коде своего собственного модуля. Общий скелет любого модуля, реализующего драйвер PCI устройства, всегда практически однотипен...

Для использования некоторой группы устройств PCI, код модуля определяет массив (таблицу) описания

устройств, обслуживаемых этим модулем. Каждому новому устройству в этом списке соответствует новый элемент. Последний элемент массива всегда нулевой, это и есть признак завершения списка устройств. Строки такого массива заполняются макросом PCI_DEVICE :

```
static struct pci_device_id i810_ids[] = {
   { PCI_DEVICE( PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG1) },
   { PCI_DEVICE( PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG3) },
   { PCI_DEVICE( PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810E_IG ) },
   { PCI_DEVICE( PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82815_CGC) },
   { PCI_DEVICE( PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82845G_IG) },
   { 0, },
};
```

Очень часто такой массив будет содержать два элемента: элемент, описывающий единичное устройство-функцию, поддерживаемую модулем, и завершающий нулевой терминатор.

Созданная структура struct pci_device_id должна быть экспортирована в пользовательское пространство, чтобы позволить системам горячего подключения и загрузки модулей (sysfs, udev и т.д.) знать, с какими устройствами работает данный модуль. Эту задачу решает макрос MODULE_DEVICE_TABLE :

```
MODULE_DEVICE_TABLE( pci, i810_ids );
```

Кроме доступа к области конфигурационных параметров, программный код должен получить доступ к областям ввода-вывода и регионов памяти, ассоциированных с PCI устройством. Таких областей ввода-вывода может быть до 6-ти (см. формат области конфигурационных параметров выше), они индексируются значением от 0 до 5. Параметры этих регионов получаются функциями:

```
unsigned long pci_resource_start( struct pci_dev *dev, int bar );
unsigned long pci_resource_end( struct pci_dev *dev, int bar );
unsigned long pci_resource_len( struct pci_dev *dev, int bar );
unsigned long pci_resource_flags( struct pci_dev *dev, int bar );
```

- где bar во всех вызовах — это индекс региона: 0 ... 5.

Первые 2 вызова возвращают начальный и конечный адрес региона ввода-вывода (pci_resource_end() возвращает последний используемый регионом адрес, а не первый адрес, следующий после этого региона.), следующий вызов — его размер, и последний — флаги. Полученные таким образом адреса областей ввода/вывода от устройства — это адреса на шине обмена (адреса шины, для некоторых архитектур - x86 из числа таких - они совпадают с физическими адресами памяти). Для использования в коде модуля они должны быть отображены в виртуальные адреса (логические), в которые отображаются страницы RAM посредством устройства управления памятью (MMU). Кроме того, в отличие от обычной памяти, часто эти области ввода/вывода не должны кэшироваться процессором и доступ не может быть оптимизирован. Доступ к памяти таких областей должен быть отмечен как «без упреждающей выборки». Всё, что относится к отображению памяти будет рассмотрено отдельно далее, в следующем разделе. Флаги PCI региона (pci_resource_flags()) определены в <linux/ioport.h>, вот некоторые из них:

IORESOURCE_IO, IORESOURCE_MEM — только один из этих флагов может быть установлен, указывает, относятся ли адреса к пространству ввода-вывода, или к пространству памяти (в архитектурах, отображающих ввод-вывод на память).

IORESOURCE_PREFETCH — определяет, допустима ли для региона упреждающая выборка.

IORESOURCE_READONLY — определяет, является ли регион памяти защищённым от записи.

Назначение любого из 6-ти регионов ввода-вывода и их число — совершенно специфично для каждого типа устройства. Например, согласно спецификации на сетевой адаптер RTL8139C, его MAC адрес занимает первые 6 байт в пространстве портов 0-го региона ввода/вывода, отведённого адаптеру.

Основной структурой, которую должны создать все драйверы PCI для того, чтобы быть правильно зарегистрированными в ядре, является структура (<linux/pci.h>) :

```
struct pci_driver {
   struct list_head node;
   char *name;
   const struct pci_device_id *id_table; /* must be non-NULL for probe to be called */
   int  (*probe) (struct pci_dev *dev, const struct pci_device_id *id); /* New device inserted */
```

```
    void (*remove) (struct pci_dev *dev);/* Device removed (NULL if not a hot-plug driver) */
    int  (*suspend) (struct pci_dev *dev, pm_message_t state); /* Device suspended */
    int  (*suspend_late) (struct pci_dev *dev, pm_message_t state);
    int  (*resume_early) (struct pci_dev *dev);
    int  (*resume) (struct pci_dev *dev); /* Device woken up */
    void (*shutdown) (struct pci_dev *dev);
    struct pci_error_handlers *err_handler;
    struct device_driver driver;
    struct pci_dynids dynids;
};
```

Где:

- `name` — имя драйвера, оно должно быть уникальным среди всех PCI драйверов в ядре, обычно устанавливается таким же, как и имя модуля драйвера, когда драйвер загружен в ядре, это имя появляется в `/sys/bus/pci/drivers/`;
- `id_table` — только что описанный массив записей `pci_device_id`;
- `probe` — функция обратного вызова инициализации устройства; в функции `probe` драйвера PCI, прежде чем драйвер сможет получить доступ к любому ресурсу устройства (область ввода/вывода или прерывание) данного PCI устройства, драйвер должен, как минимум, вызвать функцию :

```
int pci_enable_device( struct pci_dev *dev );
```

- `remove` — функция обратного вызова при удалении устройства;
- `suspend` — функция менеджера энергосохранения, вызываемая когда устройство уходит в пассивное состояние (засыпает);
- `resume` — функция менеджера энергосохранения, вызываемая когда устройство пробуждается;
- ... и другие функции обратного вызова.

Обычно для создания правильную структуру `struct pci_driver` достаточно бывает определить, как минимум, поля:

```
static struct pci_driver own_driver = {
   .name = "mod_skel",
   .id_table = i810_ids,
   .probe = probe,
   .remove = remove,
};
```

Теперь устройство может быть зарегистрировано в ядре:

```
int pci_register_driver( struct pci_driver *dev );
```

Этот вызов возвращает 0 если регистрация устройства прошла успешно. При завершении (выгрузке) модуля выполняется обратная операция:

```
void pci_unregister_driver( struct pci_driver *dev );
```

К этой точке рассмотрения у нас есть вся информация для того, чтобы начинать запись специфических операций ввода-вывода для нашего устройства. Некоторых дополнительных замечаний заслуживает регистрация обработчика прерываний от устройства, но это уже будет, скорее, повторение того материала, который мы рассматривали ранее...

Подключение к линии прерывания

Установка обработчиков прерываний и их написание рассматривалось выше. Здесь мы останавливаемся только на той детали этого процесса, что при установке обработчика прерывания для устройства — необходимо указывать используемую им линию IRQ :

```
typedef irqreturn_t (*irq_handler_t)( int, void* );
int request_irq( unsigned int irq, irq_handler_t handler, ... );
```

В устройствах шины ISA в поле первого параметра указывалось фиксированное значение (номер линии IRQ), устанавливаемое механически на плате устройства (переключателями, джамперами, ...), или записываемое конфигурационными программами в EPROM устройства. В устройствах PnP ISA — предпринимались попытки

проб и тестирования различных линий IRQ на принадлежность данному устройству. В нынешних PCI устройствах это значение извлекается из области конфигурационных параметров устройства (смещение 0x3C), но делается это не непосредственно, а посредством API ядра из структуры struct pci_dev. И тогда весь процесс регистрации, который очень часто записывается в теле функции probe, о которой говорилось выше, записывается, например, так:

```
struct pci_dev *pdev = NULL;
pdev = pci_get_device( MY_PCI_VENDOR_ID, MY_PCI_DEVICE_ID, NULL );
char irq;
pci_read_config_byte( pdev, PCI_INTERRUPT_LINE, &irq );
request_irq( irq, ... );
```

Последний оператор и устанавливает обработчик прерываний для этого устройства PCI. Вся дальнейшая работа с прерываниями обеспечивается уже самим установленным обработчиком прерывания, как это детально обсуждалось раньше.

Отображение памяти

Показанные ранее адреса из адресных регионов устройства PCI, возвращаемые вызовами PCI API:

```
unsigned long pci_resource_start( struct pci_dev *dev, int bar );
unsigned long pci_resource_end( struct pci_dev *dev, int bar );
```

Результаты функция — это адреса шины, которые (в зависимости от архитектуры) необходимо преобразовать в виртуальные (логические) адреса, с которыми оперирует код адресных пространств и ядра и пользователя:

```
#include <asm/io.h>
unsigned long virt_to_bus( volatile void *address );
void *bus_to_virt( unsigned long address );
unsigned long virt_to_phys( volatile void *address );
void *phys_to_virt( unsigned long address );
```

Примечание: для x86 архитектуры физический адрес (phys) и адрес шины (bus) — это одно и то же, но это не означает, что это так же происходит и для других архитектур.

Большинство PCI устройств отображают свои управляющие регистры на адреса памяти и высокопроизводительные приложения предпочитают иметь прямой доступ к таким регистрам, вместо того, чтобы постоянно вызывать `ioctl()` для выполнения этой работы. Отображение устройства означает связывание диапазона адресов пользовательского пространства с памятью устройства. Всякий раз, когда программа читает или записывает в заданном диапазоне адресов, она на самом деле обращается к устройству. Существенным ограничением отображения памяти (`mmap()`) является то, что ядро может управлять виртуальными адресами только на уровне таблиц страниц, таким образом, отображаемая область должна быть кратной размеру страницы RAM (PAGE_SIZE) и должна находиться в физической памяти начиная с адреса, который кратен PAGE_SIZE. Если рассмотреть адрес памяти (виртуальный или физический) он делится на номер страницы и смещение внутри этой страницы; например, если используются страницы по 4096 байт, 12 младших значащих бит являются смещением, а остальные, старшие биты, указывают номер страницы. Если отказаться от смещения и сдвинуть оставшуюся часть адреса вправо, результат называют номером страничного блока (page frame number, PFN). Сдвиг битов для конвертации между номером страничного блока и адресами является довольно распространённой операцией, существующий макрос PAGE_SHIFT сообщает на сколько битов в текущей архитектуре должно быть выполнено смещение адреса для выполнения преобразования в PFN.

DMA

Работа PCI устройства может быть предусмотрена как по прямому чтению адресов ввода/вывода, так и (что гораздо чаще) пользуясь механизмом DMA (Direct Memory Access). Только простые и низко скоростные устройства используют программный ввод-вывод. Передача данных по DMA организуется на аппаратном уровне, и выполняется (например, когда программа запрашивает данные через такую функцию, например, как `read()`) в таком порядке:

– когда процесс вызывает `read()`, метод драйвера выделяет буфер DMA (или указывает адрес в ранее выделенном буфере) и выдаёт команду оборудованию передавать свои данные в этот буфер (указывая в этой команде адрес начала буфера и объём передачи);

– инициировавший операцию **процессор** после этого или блокируется по дотупу к шине памяти (или может выполнять некоторые иные действия, в пределах разгрузки конвейера команд процессора, поскольку шина доступа к памяти блокирована);

– периферийное устройство аппаратно захватывает шину обмена и записывает данные последовательно в буфер DMA с указанного адреса, после этого вызывает **прерывание**, когда весь заказанный объём передан;

– обработчик прерывания подтверждает прерывание и переводит процесс в активное состояние, процесс теперь получает входные данные, имеет возможность читать данные.

С операцией `write()` симметричная история: процессор загружает в устройство начальный адрес и размер передаваемой области, инициирует DMA и уходит с шины памяти. Периферийное устройство когда оно само, полностью автономно сосчитает подготовленный для него обменный блок памяти — уведомит (разбудит) процессор по прерыванию.

Организация обмена по DMA это основной способ взаимодействия со всеми высокопроизводительными устройствами. С другой стороны, обмен по DMA полностью зависим от деталей аппаратной реализации, поэтому в общем виде может быть рассмотрен только достаточно поверхностно. Уже из схематичного описания выше понятно, что одно из ключевых действий, которые должен выполнить код модуля — это предоставить устройству буфер для выполнения операций DMA (предоставить буфер — предполагает указание двух его параметров: начального адреса и размера). Буфера DMA могут выделяться только в строго определённых областях памяти:

- эта память должна распределяться в физически непрерывной области памяти, поэтому выделение посредством `vmalloc()` неприменимо, память под буфера должна выделяться `kmalloc()` или `__get_free_pages()`;

- для многих архитектур выделение памяти должно быть специфицировано с флагом `GFP_DMA`, для x86 PCI устройств это будет выделение ниже адреса `MAX_DMA_ADDRESS`=16MB;

- память должна выделяться начиная с **границы страницы** физической памяти, и в объёме **целых** страниц физической памяти;

Для **распределения памяти** под буфера DMA предоставляются несколько альтернативных групп API (в зависимости от того, что мы хотим получить), их реализации полностью архитектурно зависимы, но вызовы создают уровень абстракций:

1. Coherent DMA mapping:

```
void *dma_alloc_coherent( struct device *dev, size_t size, dma_addr_t *dma_handle, gfp_t flag );
void dma_free_coherent( struct device *dev, size_t size, void *vaddr, dma_addr_t dma_handle );
```

- здесь не требуется распределять предварительно буфер DMA, этот способ применяется для устойчивых распределений многократно (повторно) используемых буферов.

2. Streaming DMA mapping:

```
dma_addr_t dma_map_single( struct device *dev, void *ptr, size_t size,
                           enum dma_data_direction direction );
void dma_unmap_single( struct device *dev, dma_addr_t dma_handle, size_t size,
                       enum dma_data_direction direction );
```

- где `direction` это направление передачи данных: `PCI_DMA_TODEVICE, PCI_DMA_FROMDEVICE, PCI_DMA_BIDIRECTIONAL, PCI_DMA_NONE`.

Этот способ применяется для выделения под однократные операции.

3. DMA pool:

```
#include <linux/dmapool.h>
struct dma_pool *dma_pool_create( const char *name, struct device *dev,
                 size_t size, size_t align, size_t allocation );
void dma_pool_destroy( struct dma_pool *pool );
void *dma_pool_alloc( struct dma_pool *pool, gfp_t mem_flags, dma_addr_t *handle );
void dma_pool_free( struct dma_pool *pool, void *vaddr, dma_addr_t handle );
```

Часто необходимо частое выделение **малых** областей для DMA обмена, но `dma_alloc_coherent()` допускает минимальное выделение только в одну физическую страницу (`PAGE_SIZE` — 4, 8, 64... Kb в зависимости от архитектуры). В таком случае оптимальным становится `dma_pool_alloc()` (как видно из прототипов, пул `struct dma_pool` должен быть сначала создан `dma_pool_create()`, и только затем из него производятся выделения `dma_pool_alloc()`).

4. Старый (перешедший из ядра 2.4) API, PCI-специфический интерфейс — два (две пары вызовов) метода, аналогичных, соответственно п.1 и п.2. Утверждается, что новый, описанный выше интерфейс, независим от вида аппаратных шин, в перспективе на новые развития; этот же (старый) API разрабатывался исключительно в ориентации на PCI шину:

```
void *pci_alloc_consistent( struct device *dev, size_t size, dma_addr_t *dma_handle );
void pci_free_consistent( struct device *dev, size_t size, void *vaddr, dma_addr_t dma_handle );
dma_addr_t pci_map_single( struct device *dev, void *ptr, size_t size, int direction );
void pci_unmap_single( struct device *dev, dma_addr_t dma_handle, size_t size, int direction );
```

Выделив любым подходящим способом блок памяти для обмена по DMA, драйвер выполняет последовательность операций (обычно это проделывается в цикле, в чём и состоит работа драйвера):

- адрес начала блока записывается в соответствующий регистр одной из 6-ти областей ввода-вывода PCI устройства, как обсуждалось выше — конкретные адреса таких регистров здесь и далее определяются исключительно спецификацией устройства...
- ещё в один специфический регистр заносится длина блока для обмена...
- наконец, в регистр команды заносится значение (чаще это выделенный бит) команды начала операции по DMA...
- когда внешнее PCI устройство сочтёт, что оно готово приступить к выполнению этой операции, оно аппаратно захватывает шину PCI, и под собственным управлением записывает (считывает) указанный блок данных...
- по завершению выполнения операции устройство освобождает шину PCI под управление процессора, и извещает систему прерыванием по выделенной устройству линии IRQ о завершении операции.

Из сказанного выше легко понять, что принципиальной операцией при организации DMA-обмена в модуле является только создание буфера DMA, всё остальное должно исполнять периферийное устройство. Примеры различного выделения буферов DMA показаны в архиве `dma.tgz` (идея тестов заимствована из [6], результаты выполнения показаны там же в файле `dma.hist`). Вот как это происходит при использовании нового API:

lab1_dma.c :

```
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/slab.h>
#include <linux/dma-mapping.h>
#include <linux/dmapool.h>

#include "out.c"
#define pool_size 1024
#define pool_align 8

// int direction = PCI_DMA_TODEVICE ;
// int direction = PCI_DMA_FROMDEVICE ;
static int direction = PCI_DMA_BIDIRECTIONAL;
//int direction = PCI_DMA_NONE;
```

```
static int __init my_init( void ) {
   char *kbuf;
   dma_addr_t handle;
   size_t size = ( 10 * PAGE_SIZE );
   struct dma_pool *mypool;
   /* dma_alloc_coherent method */
   kbuf = dma_alloc_coherent( NULL, size, &handle, GFP_KERNEL );
   output( kbuf, handle, size, "This is the dma_alloc_coherent() string" );
   dma_free_coherent( NULL, size, kbuf, handle );
   /* dma_map/unmap_single */
   kbuf = kmalloc( size, GFP_KERNEL );
   handle = dma_map_single( NULL, kbuf, size, direction );
   output( kbuf, handle, size, "This is the dma_map_single() string" );
   dma_unmap_single( NULL, handle, size, direction );
   kfree( kbuf );
   /* dma_pool method */
   mypool = dma_pool_create( "mypool", NULL, pool_size, pool_align, 0 );
   kbuf = dma_pool_alloc( mypool, GFP_KERNEL, &handle );
   output( kbuf, handle, size, "This is the dma_pool_alloc() string" );
   dma_pool_free( mypool, kbuf, handle );
   dma_pool_destroy( mypool );
   return -1;
}
```

Тот же код, но использующий специфичный для PCI API:

lab1_dma_PCI_API.c :

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

#include "out.c"

// int direction = PCI_DMA_TODEVICE ;
// int direction = PCI_DMA_FROMDEVICE ;
static int direction = PCI_DMA_BIDIRECTIONAL;
//int direction = PCI_DMA_NONE;

static int __init my_init( void ) {
   char *kbuf;
   dma_addr_t handle;
   size_t size = ( 10 * PAGE_SIZE );
   /* pci_alloc_consistent method */
   kbuf = pci_alloc_consistent( NULL, size, &handle );
   output( kbuf, handle, size, "This is the pci_alloc_consistent() string" );
   pci_free_consistent( NULL, size, kbuf, handle );
   /* pci_map/unmap_single */
   kbuf = kmalloc( size, GFP_KERNEL );
   handle = pci_map_single( NULL, kbuf, size, direction );
   output( kbuf, handle, size, "This is the pci_map_single() string" );
   pci_unmap_single( NULL, handle, size, direction );
   kfree( kbuf );
   /* let it fail all the time! */
   return -1;
}
```

Каждый из методов (в одном и другом тесте) последовательно создаёт буфер для DMA операций, записывает туда строку именующую метод создания, вызывает диагностику и удаляет этот буфер. Вот общая

часть двух модулей, в частности, содержащая функцию диагностики:

out.c :

```
static int __init my_init( void );
module_init( my_init );

MODULE_AUTHOR( "Jerry Cooperstein" );
MODULE_AUTHOR( "Oleg Tsiliuric" );
MODULE_DESCRIPTION( "LDD:1.0 s_23/lab1_dma.c" );
MODULE_LICENSE( "GPL v2" );

#define MARK "=> "
 static void output( char *kbuf, dma_addr_t handle, size_t size, char *string ) {
   unsigned long diff;
   diff = (unsigned long)kbuf - handle;
   printk( KERN_INFO MARK "kbuf=%12p, handle=%12p, size = %d\n",
           kbuf, (void*)(unsigned long)handle, (int)size );
   printk( KERN_INFO MARK "(kbuf-handle)= %12p, %12lu, PAGE_OFFSET=%12lu, compare=%lu\n",
           (void*)diff, diff, PAGE_OFFSET, diff - PAGE_OFFSET );
   strcpy( kbuf, string );
   printk( KERN_INFO MARK "string written was, %s\n", kbuf );
}
```

Вот как выглядит выполнение этих примеров:

```
$ sudo insmod lab1_dma.ko
insmod: error inserting 'lab1_dma.ko': -1 Operation not permitted
$ dmesg | tail -n200 | grep '=>'
=> kbuf=    c0c10000, handle=      c10000, size = 40960
=> (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
=> string written was, This is the dma_alloc_coherent() string
=> kbuf=    d4370000, handle=    14370000, size = 40960
=> (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
=> string written was, This is the dma_map_single() string
=> kbuf=    c0c02000, handle=      c02000, size = 40960
=> (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
=> string written was, This is the dma_pool_alloc() string
$ sudo insmod lab1_dma_PCI_API.ko
insmod: error inserting 'lab1_dma.ko': -1 Operation not permitted
$ dmesg | tail -n50 | grep '=>'
=> kbuf=    c0c10000, handle=      c10000, size = 40960
=> (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
=> string written was, This is the pci_alloc_consistent() string
=> kbuf=    d4370000, handle=    14370000, size = 40960
=> (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
=> string written was, This is the pci_map_single() string
```

Эти примеры интересны не столько своими результатами, сколько тем, что фрагменты этого кода могут быть использованы в качестве стартовых шаблонов для написания реальных DMA обменов.

Устройства USB

Стандарт USB описывает протокол ведущий-ведомый (master-slave), где ведущим **всегда** является USB хост, а ведомым — периферийное устройство. Контроллер хоста, в свою очередь, является одним из устройств на PCI шине, как это обсуждалось ранее, например:

```
$ lspci | grep USB
00:1d.0 USB Controller: Intel Corporation N10/ICH 7 Family USB UHCI Controller #1 (rev 01)
00:1d.1 USB Controller: Intel Corporation N10/ICH 7 Family USB UHCI Controller #2 (rev 01)
00:1d.2 USB Controller: Intel Corporation N10/ICH 7 Family USB UHCI Controller #3 (rev 01)
00:1d.3 USB Controller: Intel Corporation N10/ICH 7 Family USB UHCI Controller #4 (rev 01)
00:1d.7 USB Controller: Intel Corporation N10/ICH 7 Family USB2 EHCI Controller (rev 01)
```

USB терминология охватывает три версии стандарта — дифференциация по скорости обмена: оригинальный стандарт 1.0 (называемый низко-скоростным) специфицирующий скорость до 1.5MBps, стандарт 1.1 (называемый полно-скоростным) специфицирующий 12MBps, стандарт 2.0 (называемый высоко-скоростным), поддерживающий до 480MBps, последний стандарт является на сегодня наиболее массовым. Также появляются уже устройства, работающие согласно ещё более высокоскоростного стандарта 3.0. Более высокие стандарты совместимы сверху вниз, и поддерживают устройства предыдущих стандартов. Обмен во всех стандартах происходит по дифференциальной последовательной линии (контакты D+ и D-). Стандарт оговаривает 4-х контактные оконечные разъёмы, которые, кроме дифференциальной линии, содержат 2 линии питания оконечного устройства[18].

В качестве хоста в системе могут присутствовать контроллеры **разных** стандартов (**не совместимых** на нижнем уровне интерфейса работы с хостом):

- UHCI (Universal Host Controller Interface): спецификация инициализированная Intel;
- OHCI (Open Host Controller Interface): спецификация созданная компаниями Compaq и Microsoft;
- EHCI (Enhanced Host Controller Interface): спецификация для поддержки стандарта USB 2.0;
- USB OTG: спецификация, популярная во встраиваемых и мобильных устройствах, в частности, для поддержки dual-role (DRD) устройств, которые могут выступать либо как хост, либо как устройство, в зависимости от ситуации.

К счастью для разработчика, низкоуровневый слой поддержки USB в ядре Linux в значительной мере нивелирует различия спецификаций контроллера для уровня API, используемого при написании модулей поддержки устройств. Поддержка разных **типов** контроллеров, и другие опции USB низкого уровня, должны быть разрешены и скомпилированы в ядре (обычно это так и имеет место), проверить какие опции доступны можно:

```
$ cat /boot/config-`uname -r` | grep _USB_
...
CONFIG_USB=y
CONFIG_USB_XHCI_HCD=y
CONFIG_USB_EHCI_HCD=y
CONFIG_USB_OHCI_HCD=y
CONFIG_USB_UHCI_HCD=y
CONFIG_USB_WHCI_HCD=m
...
```

Схема **идентификации** устройств парой VD:PID (Vendor ID : Product ID — производитель : изделие), оказавшаяся плодотворной ранее для устройств PCI, была перенесена и для устройств USB (пара значений после 'ID' в листинге):

```
$ lsusb
Bus 001 Device 002: ID 0424:2503 Standard Microsystems Corp. USB 2.0 Hub
Bus 001 Device 003: ID 1a40:0101 TERMINUS TECHNOLOGY INC. USB-2.0 4-Port HUB
Bus 001 Device 005: ID 046d:080f Logitech, Inc. Webcam C120
Bus 004 Device 002: ID 046d:c517 Logitech, Inc. LX710 Cordless Desktop Laser
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 006: ID 03f0:171d Hewlett-Packard Wireless (Bluetooth + WLAN) Interface [Integrated
```

18 Стандарт USB OTG, кроме этого, предусматривает 5-й контакт для идентификации по признаку хост-устройство.

```
Module]
Bus 001 Device 007: ID 08ff:2580 AuthenTec, Inc. AES2501 Fingerprint Sensor
Bus 001 Device 009: ID 16d5:6502 AnyDATA Corporation CDMA/UMTS/GPRS modem
```

На один контроллер USB (разъем) могут быть последовательно подключены до 127 устройств. Но непосредственно подключать одно устройство к другому нельзя, поскольку питание таких устройств осуществляется по той же шине. Поэтому для подключения дополнительных устройств используются специальные хабы, обеспечивающие снабжение устройств энергией. В результате USB устройства образуют дерево, каждая не терминальная вершина которого является хабом. Каждое устройство USB характеризуется и своей адресацией (размещением на шине), которая имеет формат: <шина>:<устройство> (2 первых числа в каждой строке). Устройство с номером 1 на каждой шине (из числа 127-ми возможных) — это **всегда** есть очередной корневой разветвитель USB (хаб) для этой шины. Адресная информация может существенно меняться в зависимости от того, в какой разъём USB включается устройство, тогда как идентификация (VID:PID) остаётся закреплённой за устройством.

Список идентификаторов USB (VID:PID) централизовано в сети поддерживается в файле с именем `usb.ids`, в некоторых дистрибутивах он может присутствовать в системе, в других нет, но, в любом случае, лучше воспользоваться самой свежей копией этого файла, например: http://www.linux-usb.org/usb.ids (образец такого файла по состоянию на 05 мая 2014г., последний на дату написание, прилагается среди примеров). Как видим, присутствуют многие прямые аналогии с идентификацией устройств PCI, но ... как утверждается: «дьявол скрывается в деталях»:

```
$ cat usb.ids  | wc -l
18810
```

А детали состоят в том, что число общеизвестных USB устройств исчисляется уже **десятками тысяч**, а с учётом «серых» производителей в несколько раз больше, и ежедневно это число увеличивается на сотни. В таких условиях **одним** и тем же модулем ядра, обычно, реализуется поддержка нескольких **десятков** или даже **сотен** типов функционально подобных устройств. Выяснить каким модулем поддерживается интересующее нас USB устройство (по VID:PID) можно такой формой команды:

```
$ modprobe -c | grep -i 16d5 | grep -i 6502
alias usb:v16D5p6502d*dc*dsc*dp*ic*isc*ip* option
```

Которая (команда) черпает свою информацию из файла `modules.alias`:

```
$ cat /lib/modules/`uname -r`/modules.alias | grep -i 16d5 | grep -i 6502
alias usb:v16D5p6502d*dc*dsc*dp*ic*isc*ip* option
```

В качестве тестового устройствах в некоторых примерах будет использоваться EUDO модем ADU-510A, как представитель класса мобильных WAN модемов, которые вызывают в последнее время наибольшее количество **вопросов**. Как видим, показанное устройство (16d5:6502) поддерживается модулем `option`. Вообще, как показывает экспериментальный опыт (на достаточно широком наборе таких устройств), из-за огромного разнообразия моделей именно WAN беспроводных модемов, поддержка конкретного экземпляра (потому что это вызывает много вопросов) может осуществляться одним из **3-х** модулей ядра (о которых мы можно запросить дополнительную информацию):

1. модуль **`usb_storage`** — поддержка более старых моделей, устройство представляется как 3 последовательных линии с именами: `/dev/ttyACM0`, `/dev/ttyACM1`, `/dev/ttyACM2` :

```
$ modinfo usb_storage
filename:       /lib/modules/3.13.6-200.fc20.i686/kernel/drivers/usb/storage/usb-storage.ko
license:        GPL
description:    USB Mass Storage driver for Linux
...
parm:           option_zero_cd:ZeroCD mode (1=Force Modem (default), 2=Allow CD-Rom (uint)
parm:           swi_tru_install:TRU-Install mode (1=Full Logic (def), 2=Force CD-Rom, 3=Force
Modem) (uint)
parm:           delay_use:seconds to delay before using a new device (uint)
parm:           quirks:supplemental list of device IDs and their quirks (string)
$ modprobe -c | grep -w usb_storage | wc -l
371
```

2. модуль **option** — поддержка наиболее распространённых моделей, устройство здесь представляется как несколько (чаще всего 3, но бывает и 1, и до 5-ти) последовательных линии с именами: /dev/ttyUSB0, /dev/ttyUSB1, /dev/ttyUSB2, ... :

```
$ modinfo option
filename: /lib/modules/2.6.42.12-1.fc15.i686.PAE/kernel/drivers/usb/serial/option.ko
license: GPL
version: v0.7.2
description: USB Driver for GSM modems
...
$ modprobe -c | grep -w option | wc -l
651
```

3. модуль **qcaux** — поддержка новых многостандартных (CDMA/GPRS/EDGE/HSUPA) моделей (например, Pantech/Verizon UMW190), здесь устройство в /dev представляется как смесь имён разного образца, например: /dev/ttyUSB0, /dev/ttyUSB1, /dev/ttyACM0 :

```
$ modinfo qcaux
filename:       /lib/modules/3.13.6-200.fc20.i686/kernel/drivers/usb/serial/qcaux.ko
license:        GPL
...
$ modprobe -c | grep -w qcaux | wc -l
15
```

Но вернёмся к иерархии подключения USB устройств, независимо от типов самих устройств. Иерархическое дерево устройств **по подключению** смотрим так (показана только часть):

```
$ lsusb -tv
/:  Bus 05.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
/:  Bus 04.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
    |__ Port 1: Dev 2, If 0, Class=HID, Driver=usbhid, 1.5M
    |__ Port 1: Dev 2, If 1, Class=HID, Driver=usbhid, 1.5M
...
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci_hcd/8p, 480M
...
    |__ Port 4: Dev 3, If 0, Class=hub, Driver=hub/4p, 480M
        |__ Port 4: Dev 9, If 0, Class=vend., Driver=option, 12M
        |__ Port 4: Dev 9, If 1, Class=vend., Driver=option, 12M
        |__ Port 4: Dev 9, If 2, Class=vend., Driver=option, 12M
        |__ Port 4: Dev 9, If 3, Class=stor., Driver=usb-storage, 12M
```

Детальнейшую (но очень объёмную, обращаем внимание на права root) информацию по любому конкретному устройству (крайне нужную разработчику драйвера) изучаем так:

```
# lsusb -vv -d 16d5:6502
Bus 001 Device 009: ID 16d5:6502 AnyDATA Corporation CDMA/UMTS/GPRS modem
Device Descriptor:
...
  idVendor           0x16d5 AnyDATA Corporation
  idProduct          0x6502 CDMA/UMTS/GPRS modem
...
# lsusb -vv -d 16d5:6502 | wc -l
159
```

Ещё одним важнейшим источником информации при работе с USB устройством для разработчика должен быть **системный журнал**, а именно сообщения, помещаемые туда sysfs и udev при подключении устройства к USB разъёму:

```
$ dmesg
...
[22292.918081] usb 1-4: new high-speed USB device number 10 using ehci_hcd
[22293.032635] usb 1-4: New USB device found, idVendor=1a40, idProduct=0101
[22293.032645] usb 1-4: New USB device strings: Mfr=0, Product=1, SerialNumber=0
[22293.032651] usb 1-4: Product: USB 2.0 Hub [MTT]
```

```
[22293.033465] hub 1-4:1.0: USB hub found
[22293.033609] hub 1-4:1.0: 4 ports detected
[22293.296159] usb 1-4.4: new full-speed USB device number 11 using ehci_hcd
[22293.372138] usb 1-4.4: New USB device found, idVendor=16d5, idProduct=6502
[22293.372148] usb 1-4.4: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[22293.372154] usb 1-4.4: Product: AnyDATA CDMA Products
[22293.372159] usb 1-4.4: Manufacturer: AnyDATA Corporation
[22293.373474] option 1-4.4:1.0: GSM modem (1-port) converter detected
[22293.373719] usb 1-4.4: GSM modem (1-port) converter now attached to ttyUSB0
[22293.374337] option 1-4.4:1.1: GSM modem (1-port) converter detected
[22293.374566] usb 1-4.4: GSM modem (1-port) converter now attached to ttyUSB1
[22293.375111] option 1-4.4:1.2: GSM modem (1-port) converter detected
[22293.375346] usb 1-4.4: GSM modem (1-port) converter now attached to ttyUSB2
...
```

Ещё более детальную информацию можно получить из рассмотрения протокола асинхронных сообщений ядра при подключении (выводимых командой `udevadm`, как уже это обсуждалось при рассмотрении `udev`).

Для одного из показанных ранее в выводе `lsusb` выше устройств (WEB-камеры), для которого мы будем проводить следующий тест, запись в файле `usb.ids` выглядит так:

```
#       List of USB ID's
# Date:    2011-04-14 20:34:04
046d  Logitech, Inc.
...
        080f  Webcam C120
...
```

Это устройство и будет использовано в примерах модулей ниже, а вы можете использовать любое доступное устройство, скорректировав соответственно VID:PID устройства в коде. Пример подключения (и отключения) и регистрации USB устройства показывает демонстрационный модуль (архив `usb.tgz`) `lab1_usb.ko` (заимствован из [6] при замене, естественно, в коде VID:PID USB устройства на используемые в эксперименте, и сделаны достаточно существенные изменениях в исходном коде). Но прежде чем рассматривать пример, отметим, что регистрация USB устройства в точности напоминает регистрацию PCI устройства. Основой для связывания является определяемая разработчиком большая структура структура (все описания в `<linux/usb.h>`)

```
struct usb_driver {
   const char *name;
   int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);
   void (*disconnect) (struct usb_interface *intf);
   int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf);
   ...
}
```

Но, в отличии от PCI функции обратного вызова `probe()` и `disconnect()` вызываются не при **загрузке** и выгрузке модуля, а при физическом **подключении** и **отключении** USB устройства. Код примера будет выглядеть так:

lab1_usb.c :

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

struct my_usb_info {   // своя структура данных, неизвестная ядру
   int connect_count;
};

#define USB_INFO KERN_INFO "MY: "

static int my_usb_probe( struct usb_interface *intf, const struct usb_device_id *id ) {
   struct my_usb_info *usb_info;
```

```
   struct usb_device *dev = interface_to_usbdev( intf );
   static int my_counter = 0;
   printk( USB_INFO "connect\n" );
   printk( USB_INFO "devnum=%d, speed=%d\n", dev->devnum, (int)dev->speed );
   printk( USB_INFO "idVendor=0x%hX, idProduct=0x%hX, bcdDevice=0x%hX\n",
           dev->descriptor.idVendor,
           dev->descriptor.idProduct, dev->descriptor.bcdDevice );
   printk( USB_INFO "class=0x%hX, subclass=0x%hX\n",
           dev->descriptor.bDeviceClass, dev->descriptor.bDeviceSubClass );
   printk( USB_INFO "protocol=0x%hX, packetsize=%hu\n",
           dev->descriptor.bDeviceProtocol,
           dev->descriptor.bMaxPacketSize0 );
   printk( USB_INFO "manufacturer=0x%hX, product=0x%hX, serial=%hu\n",
           dev->descriptor.iManufacturer, dev->descriptor.iProduct,
           dev->descriptor.iSerialNumber);
   usb_info = kmalloc( sizeof( struct my_usb_info ), GFP_KERNEL );
   usb_info->connect_count = my_counter++;
   usb_set_intfdata( intf, usb_info );
   printk( USB_INFO "connect_count=%d\n\n", usb_info->connect_count );
   return 0;
}

static void my_usb_disconnect( struct usb_interface *intf ) {
   struct my_usb_info *usb_info;
   usb_info = usb_get_intfdata(intf);
   printk( USB_INFO "disconnect\n" );
   kfree( usb_info );
}

static struct usb_device_id my_usb_table[] = {
   { USB_DEVICE( 0x046d, 0x080f ) }, // Logitech, Inc. - Webcam C120
   { }                               // Null terminator (required)
};

MODULE_DEVICE_TABLE( usb, my_usb_table );

static struct usb_driver my_usb_driver = {
   .name = "usb-my",
   .probe = my_usb_probe,
   .disconnect = my_usb_disconnect,
   .id_table = my_usb_table,
};

static int __init my_init_module( void ) {
   int err;
   printk( USB_INFO "Hello USB\n" );
   err = usb_register( &my_usb_driver );
   return err;
}

static void my_cleanup_module( void ) {
   printk( USB_INFO "Goodbye USB\n" );
   usb_deregister( &my_usb_driver );
}

module_init( my_init_module );
module_exit( my_cleanup_module );
```

Сложность наблюдения подобного модуля (для любого вашего устройства) состоит в том, что необходимо из системы удалить модуль, **ранее** поддерживающий данное устройство, и обрабатывающий его горячие подключения. Сделать это можно отследив сообщения такого модуля при подключениях вашего USB-устройства, в случае рассматриваемой WEB-камеры это потребовало:

```
$ dmesg
...
usb 1-4: new high speed USB device using ehci_hcd and address 19
usb 1-4: New USB device found, idVendor=046d, idProduct=080f
usb 1-4: New USB device strings: Mfr=0, Product=0, SerialNumber=2
usb 1-4: SerialNumber: 1DC23270
usb 1-4: configuration #1 chosen from 1 choice
uvcvideo: Found UVC 1.00 device <unnamed> (046d:080f)
input: UVC Camera (046d:080f) as /devices/pci0000:00/0000:00:1d.7/usb1/1-4/1-4:1.0/input/input17
$ lsmod | grep uvcvideo
uvcvideo               47532  0
videodev               28423  1 uvcvideo
v4l1_compat            11370  2 uvcvideo,videodev
$ sudo rmmod uvcvideo
$ lsmod | grep uvcvideo
```

Только проделав это мы будем видеть при последовательных подключениях нашего устройства реакцию собственного тестового модуля:

```
$ sudo insmod lab1_usb.ko.
$ lsmod | grep lab
lab1_usb                1546  0
$ dmesg | tail -n 10
MY: Hello USB
...
```

... размыкаем кабель USB-камеры :

```
$ dmesg | tail -n 3
...
usb 1-4: USB disconnect, address 19
MY: disconnect
```

... и снова подключаем кабель USB-камеры :

```
$ dmesg | tail -n 20
...
usb 1-4: new high speed USB device using ehci_hcd and address 20
usb 1-4: New USB device found, idVendor=046d, idProduct=080f
usb 1-4: New USB device strings: Mfr=0, Product=0, SerialNumber=2
usb 1-4: SerialNumber: 1DC23270
usb 1-4: configuration #1 chosen from 1 choice
...
MY: connect
MY: devnum=20, speed=3
MY: idVendor=0x46D, idProduct=0x80F, bcdDevice=0x9
MY: class=0xEF, subclass=0x2
MY: protocol=0x1, packetsize=64
MY: manufacturer=0x0, product=0x0, serial=2
MY: connect_count=1
...
$ sudo rmmod lab1_usb
$ dmesg | tail -n 2
MY: Goodbye USB
usbcore: deregistering interface driver usb-my
```

То, что показано выше, относилось к идентификации USB устройства и увязыванию его в код модуля. Но никак не затрагивало обмен данными с устройством. Модель описания устройства USB, в общем случае, включает в себя одну или более **конфигураций** для каждого устройства, но активной в любой момент времени

может быть только одна из них. Конфигурации имеют один или более **интерфейсов**, каждый из которых может содержать различные параметры/настройки. Такие интерфейсы могут соответствовать стандарту USB, а могут быть специфичными лишь для определенного производителя/устройства. Интерфейсы имеют одну или более **конечных точек** (endpoints), каждая из которых поддерживает **только один** тип и направление передачи данных (например, «bulk out» или «interrupt in»). Полная конфигурация может иметь до шестнадцати конечных точек в каждом направлении. Передача данных по USB осуществляется пакетами. Для каждой концевой точки хранится запись о максимальном размере пакета. Хост всегда является мастером в обмене независимо от направления, он устанавливает флаг направления обмена: `out` — если хост отсылает данные устройству, `in` — если хост отправляет запрос на приём данных из устройства. Устройство никогда не инициирует передачу данных к хосту.

Любая обменная операция производится только с конечной (концевой) точкой устройства (endpoint, EP). Только EP может выступать как источник или приёмник данных. Устройство может иметь до 32 EP: 16 на приём и 16 на передачу. Обращение к выбранному EP происходит по его адресу (номеру). Стандарты USB поддерживает 4 **типа** передачи данных:

- **bulk**, для пакетной передачи больших объёмов некритичной по времени информации, размер пакетов 8, 16, 32, 64 для USB 1.1 и 512 для USB 2.0, используется алгоритм подтверждения и повторной передачи (в случае возникновения ошибок), поэтому этот тип является достоверным, поддерживаются оба направления — in и out;

- **control**, используется для передачи конфигурационной и управляющей информации, используются алгоритмы подтверждения и повторной передачи, направления — in (status) и out (setup, control);

- **interrupt**, для получения малых порций критичной по времени информации от устройства, размер пакета от 1 до 64 байт для USB 1.1 и до 1024 байт для USB 2.0, этот тип предполагает, что устройство будет опрашиваться (со стороны хоста) с заданным интервалом, направление только in;

- **isochronous**, для передачи реал-тайм потоков на фиксированных скоростях передачи без управления потоком (без подтверждений), область применения: аудио-потоки, видео-потоки, ... размер пакета до 1023 байт для USB 1.1 и до 1024 байт для USB 2.0, предусмотрен контроль ошибок на приёмной стороне по CRC16, направления — in и out.

Из представленных описаний уже понятно то множество «степеней свободы», которое отдано на откуп фантазии разработчика USB устройства. Все требуемые алгоритмы USB реализованы в ядре Linux, а программисту драйверов предоставляется удобный и простой интерфейс в виде набора функций, макросов, структур (которые ищем в `<linux/usb.h>`), таких, например, как (перечисление для справки): `usb_register_dev()`, `interface_to_usbdev()`, `usb_control_msg()`, `usb_interrupt_msg()`, `usb_bulk_msg()`, `usb_set_interface()`, `usb_reset_endpoint()`,... С помощью этого API достаточно несложно реализовать обмен для всякого конкретного случая.

Собственно, реализация обмена с USB устройством, **после** его связывания с модулем ядра, является уже существенно зависимой от специфики самого устройства и протоколов его обмена. Это уже не есть предметом непосредственно общей функциональности модуля ядра, а поэтому обмен и не стоит рассматривать в общем виде.

Многофункциональные устройства

В последние несколько (5-7) лет среди многих производителей USB устройств стала популярна идея многофункциональных USB устройств: путём записи по определённой концевой точке (EP) заданной **производителем** байтовой (символьной) управляющей последовательности VID:PID могут изменяться, и устройство приобретает совершенно другую функциональность. Первоначально это приобрело широкую популярность для беспроводных USB модемов WAN самых разных коммуникационных стандартов:

- 2G и 2.5G: GPRS (General Packet Radio Service) и EDGE (Enhanced Data rates for GSM Evolution) — для сетей GSM;

- 3G: EVDO (или EV-DO — Evolution-Data Only) — для сетей CDMA;

- 4G: LTE (Long Term Evolution) и WiMAX (802.16-2005, 802.16e) — для автономных сетей передачи данных;

Позже эту технику производители распространили и на другие типы устройств. В простейшем случае, логика состоит в том, что при первоначальном включении устройство выглядит как флэш-диск, на котором записаны драйвера, инсталляционные и тестовые программы (по крайней мере, для операционной системы Windows). После инсталляции, при повторных подключения устройства к USB шине, **модуль**-драйвер или какое-то **приложение** пользовательского пространства должно заслать управляющую последовательность в устройство, чем **переключить** его в режим модема (сменив при этом VID:PID). В более общем случае а). это могут быть устройства различной функциональности, не только названной и б). может быть и более 2-х функциональностей, которые заложены в устройство. При написании модулей поддержки нужно быть готовым к такой архитектуре.

Проследить процесс смены VID:PID многофункционального устройства можно запустив монитор `udevadm` (обсуждался ранее) перед физическим подключением устройства к USB и наблюдая генерируемые при этом сообщения `sysfs`. В показанных примерах мы будем экспериментировать с 3G модемом ADU-510A от AnyDATA (показан на рисунке, довольно популярная в Linux модель), а вы можете проделать то же с любым оказавшимся под рукой USB модемом. Последовательность сообщений об изменениях состояний весьма объёмная, но нет другого способа отследить происходящие процессы, как её анализ. Ниже показаны только некоторые, ключевые для понимания, сообщения ядра, из числа 50-ти (!) сообщений, передаваемых через `netlink` сокет в ходе отработки такого горячего подключения:

```
$ udevadm monitor --kernel --property
monitor will print the received events for:
KERNEL - the kernel uevent

KERNEL[9534.471672] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4 (usb)
ACTION=add
BUSNUM=001
DEVNAME=/dev/bus/usb/001/010
DEVNUM=010
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4
DEVTYPE=usb_device
MAJOR=189
MINOR=9
PRODUCT=5c6/1000/0
SEQNUM=2404
SUBSYSTEM=usb
TYPE=0/0/0
...
KERNEL[9534.478431] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/host8
(scsi)
ACTION=add
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/host8
DEVTYPE=scsi_host
SEQNUM=2406
SUBSYSTEM=scsi
...
KERNEL[9535.490388] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-
3.4:1.0/host8/target8:0:0/8:0:0:0/block/sr1 (block)
ACTION=add
DEVNAME=/dev/sr1
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-
3.4:1.0/host8/target8:0:0/8:0:0:0/block/sr1
DEVTYPE=disk
MAJOR=11
```

```
MINOR=1
SEQNUM=2411
SUBSYSTEM=block
...
KERNEL[9536.048197] remove   /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-
3.4:1.0/host8/target8:0:0/8:0:0:0/block/sr1 (block)
ACTION=remove
DEVNAME=/dev/sr1
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-
3.4:1.0/host8/target8:0:0/8:0:0:0/block/sr1
DEVTYPE=disk
MAJOR=11
MINOR=1
SEQNUM=2420
SUBSYSTEM=block
...
KERNEL[9537.336850] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4 (usb)
ACTION=add
BUSNUM=001
DEVNAME=/dev/bus/usb/001/011
DEVNUM=011
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4
DEVTYPE=usb_device
MAJOR=189
MINOR=10
PRODUCT=16d5/6502/0
SEQNUM=2427
SUBSYSTEM=usb
TYPE=0/0/0
...
KERNEL[9537.338995] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-
3.4:1.0/ttyUSB0/tty/ttyUSB0 (tty)
ACTION=add
DEVNAME=/dev/ttyUSB0
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/ttyUSB0/tty/ttyUSB0
MAJOR=188
MINOR=0
SEQNUM=2430
SUBSYSTEM=tty
...
```

Первым сообщением (SEQNUM=2404) фиксируется обнаружение на USB шине устройства 5c6:1000 (что как мы увидим далее, никак не соответствует VID:PID модема). Позже (SEQNUM=2406) это устройство квалифицируется как SCSI (MAJOR=11 , MINOR=1), для него устанавливается (SEQNUM=2411) модуль-драйвер блочного устройства `/dev/sr1`. Но дальше (SEQNUM=2420) следует размонтирование и удаление устройства (MAJOR=11 , MINOR=1), обнаружение (SEQNUM=2427) нового устройства 16d5:6502 (и это уже и есть 3G модем), и делается вся дальнейшая инициализация такого устройства.

В результате такого установочного процесса мы получаем новое USB устройство:

```
$ lsusb | grep AnyDATA
Bus 001 Device 011: ID 16d5:6502 AnyDATA Corporation CDMA/UMTS/GPRS modem
```

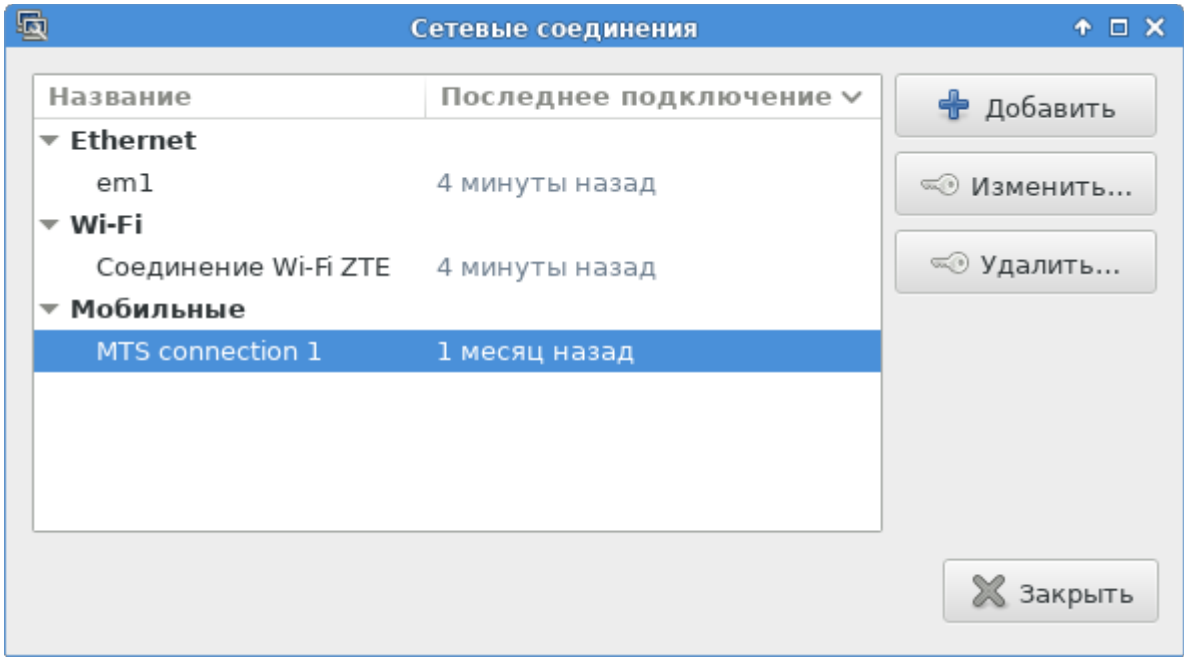
А в каталоге устройств — 3 дополнительных устройства, имитирующих поведение **последовательных** линий передачи, из которых `/dev/ttyUSB0` является линией AT-совместимого модема, для которой мы можем устанавливать традиционное PPP-соединение любым из известных способов, например, с помощью любого диалера (весьма популярным является `wvdial`, но можно это сделать и классическим способом ручной настройки PPP):

```
$ ls -l /dev/ttyUSB*
crw-rw---- 1 root dialout 188, 0 май 13 12:57 /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 1 май 13 12:57 /dev/ttyUSB1
```

```
crw-rw---- 1 root dialout 188, 2 май 13 12:57 /dev/ttyUSB2
```

Мы это сделаем с помощью известного инструмента (программы, GUI аплета) `NetworkManager` :

```
$ which NetworkManager
/usr/sbin/NetworkManager
```



Убеждаемся, что всё вышесказанное действительно имеет место (сетевой интерфейс `ppp0`):

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: em1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group
default qlen 1000
    link/ether a0:1d:48:f4:93:5c brd ff:ff:ff:ff:ff:ff
3: wlo1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DORMANT group
default qlen 1000
    link/ether 34:23:87:d6:85:0d brd ff:ff:ff:ff:ff:ff
4: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode
DEFAULT group default qlen 3
    link/ppp
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    1024   0        0 em1
80.255.73.34    0.0.0.0         255.255.255.255 UH    0      0        0 ppp0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 em1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 wlo1
```

Остаётся невыясненным единственный вопрос: **кто** и с помощью **какого механизма** переключил устройство 5c6:1000 на устройство 16d5:6502 между SEQNUM=2411 и SEQNUM=2427? Это может делать, в принципе, непосредственно программный код вашего модуля после его установки, и такой подход имеет смысл проанализировать при подготовке проекта. Но в данном случае это выполнил **прикладной** пакет `usb_modeswitch`, разработанный специально для таких случаев, потому что такие потребности многократно повторяются. Рассмотрим конфигурационный каталог этого пакета:

```
$ ls -w80 /etc/usb_modeswitch.d
bash-4.2$ ls -w80 /etc/usb_modeswitch.d
03f0:002a                  0af0:7a05        12d1:14c3           19d2:1216
0408:ea17                  0af0:8006        12d1:14c4           19d2:1224
...
```

```
05c6:0010                   0af0:d157         12d1:1f16           1bbb:f017
05c6:1000:sVe=GT            0af0:d255         12d1:1f17           1bbb:f052
05c6:1000:sVe=Option        0af0:d257         12d1:1f18           1c9e:1001
05c6:1000:uMa=AnyDATA       0af0:d357         12d1:1f19           1c9e:6061
05c6:1000:uMa=CELOT         0b3c:c700         12d1:380b           1c9e:9200
05c6:1000:uMa=DGT           0b3c:f000         1307:1169           1c9e:9800
05c6:1000:uMa=Option        0cf3:20ff         1410:5010           1c9e:98ff
...
```

Здесь несколько сот (в зависимости от версии) **многофункциональных** USB устройств, требующих переключения режима:

```
$ ls -l /etc/usb_modeswitch.d | wc -l
314
```

Если подключается USB устройство с VID:PID совпадающим с **именем** одного из файлов (обслуживаемых устройств), то срабатывают правила (обмена командами USB), записанные в содержимом этого файла (например, посылка символьной строки `MessageContent`)

```
$ cat /etc/usb_modeswitch.d/05c6:1000:uMa=AnyDATA
# AnyDATA devices, Bless UC165
TargetVendor=   0x16d5
TargetProduct=  0x6502
MessageContent="5553424312345678000000000000061b000000020000000000000000000000"
```

А обеспечивает это переключение (в соответствии с правилами) программа с именем именно `usb_modeswitch` которую запускает, естественно, `udevd` в соответствии с его правилами файла `40-usb_modeswitch.rules`, который ищите в каталогах `/etc/udev/rules.d` или `/usr/lib/udev/rules.d` — получив известный VID:PID демон запускает программу `usb_modeswitch`. Но программу `usb_modeswitch` можно запустить и в консольном режиме, в диалоге, для изучения поведения **многофункционального** устройства. Правила `40-usb_modeswitch.rules` представляют собой практически повторение однотипных строк для каждой из обслуживаемых моделей оборудования:

```
ACTION=="add" SUBSYSTEM=="usb", SYSFS{idProduct}=="1446", SYSFS{idVendor}=="12d1",
RUN+="/usr/sbin/usb_modeswitch"
```

Программа `usb_modeswitch` является программой **пользовательского** пространства и, как и следовало ожидать, использует библиотеки проекта `libusb`: обмен с USB устройствами из пользовательского пространства, об этом инструменте будет коротко сказано далее.

Синтаксис записи самих правил `usb_modeswitch` (`/etc/usb_modeswitch.d`) в меру замысловат, но зато хорошо описан, и его детальное рассмотрение выходит за рамки нашего рассмотрения. Принципиально важно то, что **для любого** многофункционального устройства (стороннего или под свой проект) вы сможете записать правила его переключений, используя пакет `usb_modeswitch`.

Операции I/O пространства пользователя

Ряд функций управления реальным устройством может быть вообще выполнен в пространстве пользователя, не прибегая к технике написания модулей, и не входя в адресное пространство ядра. Таким случаем, например, в котором работа в пространстве пользователя может иметь большой смысл, является тот случай, когда вы начинаете иметь дело с новым и необычным оборудованием. Работая таким образом вы можете научиться управлять вашим оборудованием без риска подвешивания системы в целом. После того, как вы сделали это, выделение этого программного обеспечения в модуль ядра должно быть безболезненной операцией.

Прежде всего, может создаться впечатление, что обеспечивая обмен из пространства пользователя можно создать только простейшие учебные приложения. Но это мнение опровергается наличием известнейших публичных проектов, использующих ввод-вывод только из пространства пользователя, некоторые примеры чему:

– Употребляемая (на сегодня безальтернативно) во всех UNIX операционных системах графическая

подсистема X11. В своём **базовом варианте** (не расширенном проприетарными видеодрайверами пространства ядра) этот пакет не нуждается в модулях ядра: работа с портами и видеопамятью адаптеров успешно осуществляется из пространства пользователя, а аппаратные прерывания, которые могут генерироваться оборудованием (обратный ход кадра) система X11 не использует. Именно эта архитектура базовой системы X11 создала возможность такого лёгкого её переноса в самые экзотические системы UNIX (как например MINIX 3), и такое широкое её применение в мире UNIX.

– Проект `libusb` — предоставление API для поддержки всего спектра возможных USB устройств из пространства пользователя. Здесь смысл и возможности реализации основаны на других основаниях: низкоуровневый обмен с USB устройством (требующий обработки прерываний и других возможностей ядра) обеспечивает базовый слой поддержки USB встроенный в ядро. Сам проект `libusb` является уже **надстройкой** пользовательского пространства, обращающийся к базовым возможностям посредством системных вызовов. Показателем признания качества `libusb` может быть тот факт, что этот API используют, в свою очередь, такие крупнейшие проекты, как: CUPS (**C**ommon **U**nix **P**rinting **S**ystem — основная на сегодня подсистема печати в Linux), SANE (обслуживание сканеров), `fprint` (биометрическая система идентификации отпечатков пальцев), `libgphoto2` (обслуживание фотографических камер) и это ещё далеко не всё...

Инструментарий

Посмотрите порты, реально присутствующие в системе, которые могут использоваться для обмена:

```
$ cat /proc/ioports
0000-0cf7 : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0062-0062 : EC data
  0064-0064 : keyboard
  0066-0066 : EC cmd
  0070-0071 : rtc0
...
  0378-037a : parport0
...
  03f8-03ff : serial
...
```

Реализацию ввода/вывода из адресного пространства **пользователя** можно реализовать несколькими способами. Первый из них — это использовать функции ввода-вывода. Для обмена с портами предоставляются (`<sys/io.h>`) набор функций:

```
unsigned char inb( unsigned short int port );
unsigned short int inw( unsigned short int port );
unsigned int inl( unsigned short int port );
void outb( unsigned char value, unsigned short int port );
void outw( unsigned short int value, unsigned short int port );
void outl( unsigned int value, unsigned short int port );
```

Сразу же акцентируем внимание (о таких вещах говорилось ранее), что при внешней полной идентичности с аналогичными функциями ядра, эти вызовы представляют собой совершенно другие реализации. Функции ядра размещены в коде ядра, а пользовательские функции реализуются инлайновыми ассемблерными вставками GCC (обсуждалось ранее):

```
static __inline unsigned char inb( unsigned short int __port ) {
   unsigned char _v;
   __asm__ __volatile__ ("inb %w1,%0":"=a" (_v):"Nd" (__port));
   return _v;
}
```

Из-за такого (инлайн) способа определений, при компиляции GCC должны быть обязательно включены опции оптимизации (как минимум `-O` или `-O2`).

Но операции обращения к портам недопустимы для не привилегированных процессов (выполняющихся в кольце защиты 3, а также не запущенные от имени `root`), позже мы обсудим как это выглядит. Для разрешения обменных операций возможно (`<sys/io.h>`):

1. Отобразить диапазон портов ввода-вывода в пространство задачи и разрешить их использовать:

```
int ioperm( unsigned long int from, unsigned long int num, int turn_on );
```

Вызов: `ioperm()` устанавливает биты привилегий для доступа к области портов ввода/вывода, параметры здесь:

- `from` - начальный номер порта области;
- `num` - число портов в области;
- `turn_on` — разрешить (1) или запретить (0) привилегированные операции.

Таким образом можно изменить привилегии только для первых 0x3ff портов ввода/вывода, если нужно получить тот же результат для всех 65536 портов, нужно воспользоваться системным вызовом `iopl()`.

2. Изменить уровень привилегированности пользовательского процесса **в отношении ввода-вывода** (это вовсе не означает перевод пользовательского приложения в другое кольцо защиты):

```
int iopl( int level );
```

Вызов `iopl()` меняет уровень **привилегий ввода-вывода**: всем процессам, уровень кольца защиты которых **ниже или равен** (более привилегированные) `level`, будут разрешены привилегированные операции. (В дополнение к неограниченному доступу к портам ввода-вывода работа на высоком уровне привилегий также позволяет процессу отключать прерывания. Скорее всего, это приведёт к сбою системы, поэтому это небезопасно). Эти права наследуются через `fork()` и `execve()`.

Естественно, что для получения привилегий ввода-вывода любым из способов процесс должен обладать правами `root`. После получения привилегий процесс может выполнять перечисленные ранее операции обмена: `out*()` и `in*()`.

Другой способ основан на том, что Linux отображает пространство аппаратных портов в файл:

```
$ ls -l /dev/port
crw-r----- 1 root kmem 1, 4 нояб. 23 14:31 /dev/port
```

Поэтому возможность состоит в том, чтобы:

- открыть `fd = open()` указанный файл (естественно, с правами root);
- сдвинуть указатель позиции в файле на требуемый порт: `lseek( fd, port, SEEK_SET );`
- считать (`read( fd, &data, 1 )`) или записать (`write( fd, &data, 1 )`) данные из порта или в порт;

Смысл проделываемого в этом случае состоит в том, что мы поручаем ядру Linux выполнить для нас операции ввода-вывода. Этот способ, конечно, будет в работе намного медленнее, но он не требует ни получения дополнительно привилегий уровня защиты процессора (режима супервизора), что небезопасно, ни оптимизации при компиляции (что иногда нежелательно).

Помимо возможности ввода/вывода, для таких программ, как правило, нужно предотвратить выгрузку страниц такой программы на диск:

```
#include <sys/mman.h>
int mlock( const void *addr, size_t len );
int munlock( const void *addr, size_t len );
int mlockall( int flags );
```

```
int munlockall( void );
```

В наиболее нужном в этом качестве вызове `mlockall()`, параметр `flags` может быть :

`MCL_CURRENT` - локировать все страницы, которые на текущий момент отображены в адресное пространство процесса;

`MCL_FUTURE` - локировать все страницы, которые будут отображаться в будущем в адресное пространство процесса;

Демонстрируем всё сказанное примером кода (архив `user_io.tgz`):

ioports.c :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/io.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>

#define PARPORT_BASE 0x378

void do_io( unsigned long addr ) {
   unsigned char zero = 0, readout = 0;
   printf( "\twriting: 0x%02x to 0x%lx\n", zero, addr );
   outb( zero, addr );
   usleep( 1000 );
   readout = inb( addr + 1 );
   printf( "\treading: 0x%02x from 0x%lx\n", readout, addr + 1 );
}

void do_read_devport( unsigned long addr ) {
   unsigned char zero = 0, readout = 0;
   int fd;
   printf( "/dev/port :\n" );
   if( ( fd = open( "/dev/port", O_RDWR ) ) < 0 ) {
      perror( "reading /dev/port method failed" ); return;
   }
   if( addr != lseek( fd, addr, SEEK_SET ) ) {
      perror( "lseek failed" ); close( fd ); return;
   }
   printf( "\twriting: 0x%02x to 0x%lx\n", zero, addr );
   write( fd, &zero, 1 );
   usleep( 1000 );
   read( fd, &readout, 1 );
   printf( "\treading: 0x%02x from 0x%lx\n", readout, addr + 1 );
   close( fd );
   return;
}

void do_ioperm( unsigned long addr ) {
   printf( "ioperm :\n" );
   if( ioperm( addr, 2, 1 ) ) {
      perror( "ioperm failed" ); return;
   }
   do_io( addr );
   if( ioperm( addr, 2, 0 ) ) perror( "ioperm failed" );
   return;
}
```

```
int iopl_level = 3;
void do_iopl( unsigned long addr ) {
   printf( "iopl :\n" );
   if( iopl( iopl_level ) ) {
      perror( "iopl failed" ); return;
   }
   do_io( addr );
   if( iopl( 0 ) ) perror( "ioperm failed" );
}

int main( int argc, char *argv[] ) {
   unsigned long addr = PARPORT_BASE;
   if( argc > 1 )
      if( !sscanf( argv[ 1 ],"%lx", &addr ) ) {
         printf( "illegal address: %s\n", argv[ 1 ] );
         return EXIT_FAILURE;
      };
   if( argc > 2 ) iopl_level = atoi( argv[ 2 ] );
   do_read_devport( addr );
   do_ioperm( addr );
   do_iopl( addr );
   return errno;
}
```

Результаты его выполнения:

```
$ sudo ./ioports
/dev/port :
        writing: 0x00 to 0x378
        reading: 0x78 from 0x379
ioperm :
        writing: 0x00 to 0x378
        reading: 0x78 from 0x379
iopl :
        writing: 0x00 to 0x378
        reading: 0x78 from 0x379
```

Некоторые особенности

Если попытаться использовать обменные операции, не получив привилегий (`ioperm()` или `iopl()`), то программа немедленно прервётся на этом операторе. Странным образом различаются при этом выполнение с правами root и без них (в архив приложен пример `ioperm.c`, выполняющий обмен, но без привилегий):

```
$ ./ioperm
writing: 0x00 to 0x378
Ошибка сегментирования (core dumped)
$ echo $?
139
$ sudo ./ioperm
writing: 0x00 to 0x378
$ echo $?
139
```

Код завершения программы (139) довольно странный, но в любом случае, он настолько серьёзный, что попытка выполнения фиксируется в системном журнале:

```
$ dmesg | tail -n1
[12310.795229] ioperm[4485] general protection ip:80484e5 sp:bf8351b0 error:0 in
ioperm[8048000+1000]
```

Функции обмена (`in*() / out*()`) вообще никаким образом не возвращают и не устанавливают код

ошибочности своего выполнения. Например, при обращении к не существующим аппаратным портам:

```
$ sudo ./ioports 200
/dev/port :
        writing: 0x00 to 0x200
        reading: 0xff from 0x201
ioperm :
        writing: 0x00 to 0x200
        reading: 0xff from 0x201
iopl :
        writing: 0x00 to 0x200
        reading: 0xff from 0x201
```

Из не существующих портов читается `0xff`, это не сильно надёжный признак, и присутствие портов, наверное, нужно контролировать отдельно, например, по содержимому `/proc/ioports`, как это показывалось выше.

Замечание относительно `iopl()` и уровней привилегий: если функции указать параметр больше 3, то она возвратит ошибку:

```
$ sudo ./ioports 378 4
...
iopl failed: Invalid argument
$ echo $?
22
```

А поскольку, в Linux из 4-х колец защиты процессора x86 (0, 1, 2, 3) использованы только 2 (0 — для ядра Linux, 3 — для пользовательского пространства), то для вызова `iopl()` имеет смысл только:

- `iopl( 3 )` - разрешить пользовательскому процессу привилегированные операции;

- `iopl( 0 )` - возвратить запрет на выполнение пользовательским процессом привилегированных операции;

Ещё один вопрос: а нельзя ли (например при отработке прототипов, о чём говорилось в начале) получить доступ из пользовательского пространства к DMA и прерываниям? Нет, никаких средств для этого в пользовательском API нет. Но для отработки прототипов можно создать простейший модуль для обработки аппаратного прерывания, который по получению прерывания должен будет всего лишь отослать **сигнал UNIX** процессу пользовательского пространства (такая возможность будет обсуждена далее). Так может решаться вопрос асинхронных событий аппаратуры.

Проект `libusb`

Проект `libusb` — это настолько мощный и широко использующийся в различных проектах инструментарий, и настолько давно прижившийся в Linux проект, что он заслуживает отдельного упоминания. Библиотека `libusb`, как уже было сказано выше, позволяет осуществлять практически все операции ввода-вывода с USB устройствами из **пользовательских** процессов. Это позволяет перенести часть (большую или меньшую) функциональности проектов, работающих с USB устройствами из модульного кода в пользовательский. Так сделано во многих известных проектах, и такой подход поощряется.

Примечание: С середины 2012 г. существует развитие этого проекта, которое может фигурировать под именем `libusbx`.

Ещё одной значимой чертой проекта есть то, что он развивается как платформенно независимый проект, и позволяет реализовывать переносимый код для операционных систем Linux, Solaris, Windows (хоть это и не является предметом нашего текущего рассмотрения).

Ещё одним достоинством, возникшим в ходе реализации проекта и участия в нём многих

заинтересованных авторов, явилось то, что для `libusb` были созданы языковые обёртки, позволяющие использовать API работы с USB устройствами из кода на различных языках программирования. На сегодня это, как минимум: C и C++, Java, C#, Go, Ada, Python, Perl, Ruby, Lua, Common Lisp, Ocaml, Haskell. Например, используя Python расширение PyUSB (одно из нескольких доступных) мы могли бы писать непосредственно из Python кода что-то типа следующего:

```
import usb.core
import usb.util as util
...
VID = 0x04d9
PID = 0x8010
dev = usb.core.find( idVendor=VID, idProduct=PID )
if dev.is_kernel_driver_active( 0 ):
   dev.detach_kernel_driver(0)
dev.set_configuration() # do reset and set active conf. Must be done before claim.
util.claim_interface( dev, None )
try:
   dev.read( 0x81, 64 )
except usb.core.USBError:
   pass
...
```

Файловая система FUSE

Файловая система FUSE (Filesystem in Userspace) — ещё один проект, позволяющий вам строить свои самые разнообразные файловые системы, не прибегая к программированию в ядре. Со стороны ядра поддержка FUSE осуществляется модулем проекта:

```
$ lsmod | grep fuse
fuse                   91410  3
```

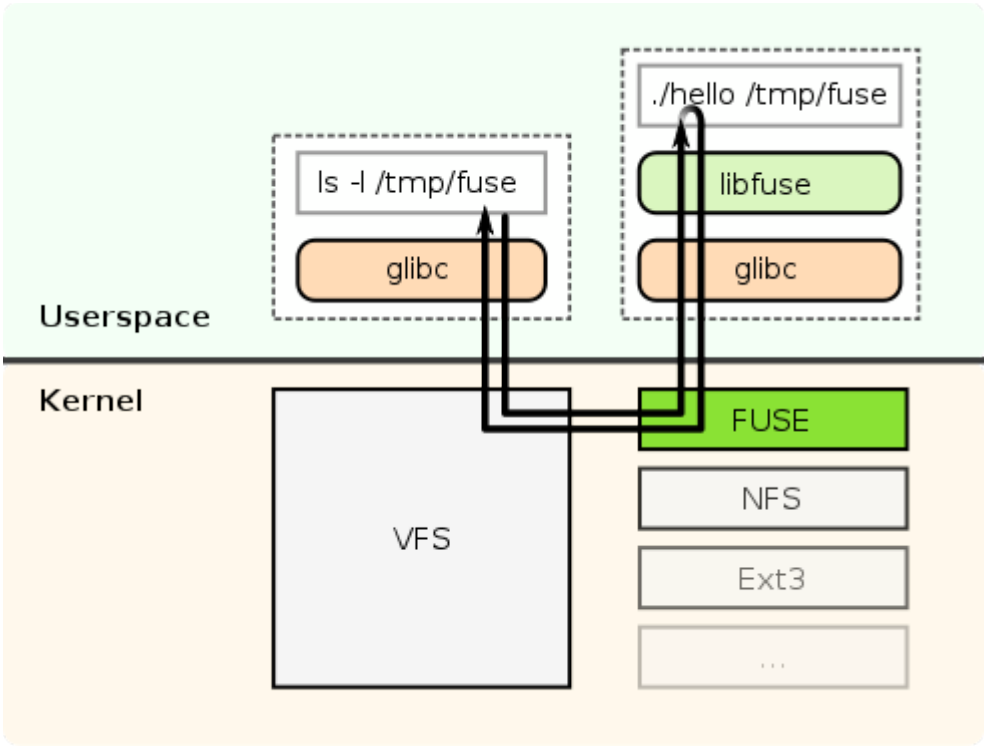
Вся остальная реализация файловой целевой системы производится в пространстве пользователя, используя API библиотеки `libfuse.so`:

```
$ ls -l /lib64/*fuse*
lrwxrwxrwx. 1 root root     16 мар 11 09:21 /lib64/libfuse.so -> libfuse.so.2.9.3
lrwxrwxrwx. 1 root root     16 янв 17  2014 /lib64/libfuse.so.2 -> libfuse.so.2.9.3
-rwxr-xr-x. 1 root root 256640 авг  3  2013 /lib64/libfuse.so.2.9.3
```

Начиная с ядра 2.6.14, файловая система FUSE включена как стандартный компонент дерева исходных кодов ядра Linux.

Огромным преимуществом FUSE является то, что для модтирования файловых систем FUSE не требуются привилегии суперпользователя `root`.

Взаимодействие компонент FUSE при работе пользовательской файловой системы (`hello`) показано на рисунке.



Весьма упрощённо, для создания собственной файловой системы вам необходимо в **приложении** пользовательского пространства:

- Определить таблицу файловых операций, указывающих реализующие функции:

```
static struct fuse_operations oper = {
    .getattr    = ... ,
    .readdir    = ... ,
    .open       = ... ,
    .read       = ... ,
};
```

- Вызвать в функции `main()` для связи с модулем `fuse`:

```
return fuse_main( argc, argv, &oper, NULL );
```

Это полностью напоминает то, что мы делали в модулях ядра, но не требует написания ни единого оператора в режиме ядра, с соответствующими устойчивостью, скоростью разработки и отладки. Более детальное рассмотрение FUSE выходит далеко за рамки нашего предмета, но в архив примеров включен `fuse.tgz`, который может быть вполне достаточным стартовым уровнем для написания собственных файловых систем. Детальнее FUSE можно изучить на странице проекта: Filesystem in Userspace.

В последние годы очень многие разработчики, в частности разнообразных мультимедийных устройств (цифровых фотоаппаратов, диктофонов, ...), представили многие **десятки** различных файловых систем на базе FUSE.

Задачи

1. Сделайте реакцией на подключение любого из имеющихся у вас USB устройств запуск выбранного приложения или осуществление других определённых вами действий.

2. Создайте образец собственной файловой системы в пространстве пользователя, используя FUSE.

11. Внутренние API ядра

«Очень трудно видеть и понимать неизбежное в хаосе вероятного»
Андрей Ваджра (псевдоним украинского политолога и публициста).

В отличие от предыдущего раздела, где мы обсуждали интерфейсы модуля «торчащие в наружу», сейчас мы сосредоточимся исключительно на тех механизмах API, которые никак не видимы и не ощущаются пользователем, но нужны они исключительно разработчику модуля в качестве строительных конструкций для реализации своих замыслов. Большинство механизмов и понятий этой части описания уже знакомо по API пользовательского пространства, они имеют там свои прямые аналогии. Но существуют и некоторые принципиальные расхождения.

Механизмы управление памятью

В общем виде управление памятью в ядре Linux, в защищённой аппаратной архитектуре, это самая сложная часть функций операционной системы. Ещё более громоздкой её делает то, что эта модель управления памятью должна отображаться на разных платформах в конкретные для платформы механизмы отображения логических (виртуальных) регионов памяти в физические. К счастью для разработчиков **модулей** ядра, большая часть механизмов управления памятью скрыта из нашего поля зрения и не имеет практического применения. Основная потребность разработчика модулей ядра состоит в выделении динамически по требованию блоков памяти заданного размера (иногда это очень небольшие блоки в десятки байт при построении динамических структур данных, а иногда большие области, исчисляемые размерами в мегабайты, например, при выделении циклических буферов данных в драйверах). В дальнейшем управление памятью будет рассматриваться только в смысле таких потребностей, вопросы выделения больших регионов памяти, из которых «нарезаются» такие запросы в модуле, и то, как это реализуется в ядре, рассматриваться не будут.

Однако, из **общей структуры** управления памятью в Linux нужно только назвать несколько фактов, неверное понимание которых радикально искажает картину происходящего:

1. Адресное пространство ядра и адресное пространство **текущего** процесса (на который указывает макрос-указатель `current`) разделяют единое «плоское» адресуемое пространство виртуальных адресов, для 32-бит архитектуры это пространство в 4Gb. Переключение контекста (состояния сегментных регистров) при переключении из пространства пользователя в пространство ядра в Linux **не производится**.

2. Исходя из этого, общий объём адресов должен разделяться в фиксированном соотношении между диапазоном для пространства пользователя, и диапазоном для пространство ядра. Для конкретности, на 32-бит платформах это соотношение обычно 3:1, и общий диапазон адресов от `0x00000000` до `0xffffffff` разделяется граничным адресом `0xc0000000`: ниже него 3Gb адресов принадлежат пространству пользователя, выше него 1Gb адресов относится к пространству ядра. Это легко видеть на примерах: все имена из `/proc/kallsyms`, все адреса функций в вашем коде модуля, все адреса динамически выделяемых по `kmalloc()` данных — все будут находится выше границы `0xc0000000` (это хорошо видно на примерах из архива примеров `hidden.tgz`). С другой стороны, все адреса в пользовательских приложениях будут ниже этой границы.

3. В принципе, и пользовательское пространство и пространство ядра могли бы располагать каждое своим изолированным адресным пространством — полным максимально возможным диапазоном адресов (для 32-бит платформ — по 4Gb для процессов и для ядра). Но при этом возникла бы необходимость перезагрузки сегментных регистров при переключении в режим ядра — при выполнении системного вызова. Разработчики ядра Linux сочли это излишне накладным из соображений производительности[19].

19 Утверждается [2], что ядра после 2.6, с дополнительным патчем могут быть сгенерированы с поддержкой режима 4Gb/4Gb, при этом достигается средняя, но приемлемая производительность.

4. Сегменты пространства пользователя, таким образом, имеют фиксированный ограниченный предел сегмента, для 32-бит это `0xc0000000`. Обработчики системных вызовов производят проверку принадлежности пространству пользователя параметров-указателей на **не превышение** этой границы. Код модуля мог бы, в принципе, непосредственно использовать указатели в пользовательском коде, если бы не возможность физического отсутствия требуемой страницы в памяти (виртуализация). Поэтому используются операции `copy_from_user()` и `copy_to_user()` для взаимодействия с данными пользователя.

5. Соотношение 3:1 и, соответственно, граница разделения `0xc0000000` — могут быть изменены, при новой **генерации** ядра Linux. Это иногда делается для специальных применений, но крайне редко.

6. Поскольку в область ядра должны отображаться ещё некоторые области, например, аппаратные области видеопамяти, и некоторые такие области расширения ROM не допускают операций записи, то все они должны исключаться из общего адресного диапазона ядра, поэтому он будет ещё немногим менее 1Gb. На типовой x86 архитектуре объём непосредственно адресуемых логических адресов составляет 892Mb.

7. А вот отображение **одинаковых** логических адресов пространства пользователя из различных процессов в отличающиеся физические адреса — это происходит не за счёт различий **сегментных регистров**, а за счёт различий на уровне страничного отображения памяти, осуществляемого MMU.

Динамическое выделение участка

В ядре Linux существует несколько альтернативных механизмов динамического выделения участка памяти (распределение статически описанных непосредственно в коде областей данных мы не будем затрагивать, хотя это тоже вариант решения поставленной задачи). Каждый из таких механизмов имеет свои особенности, и, естественно, свои преимущества и недостатки перед своими альтернативными собратьями.

Примечание: Отметьте, что (практически) все механизмы динамического выделения памяти в пространстве пользователя (`malloc()`, `calloc()`, etc.) являются **библиотечными** вызовами, которые ретранслируются (транзитом через соответствующие системные вызовы[20]) в рассматриваемые здесь механизмы. Исключение составляет один `alloca()`, который распределяет память непосредственно из стека выполняемой функции (что имеет свою опасность в использовании). Таким образом, рассматриваемые вопросы имеют прямой практический интерес и для прикладного программирования (пространства пользователя).

Механизмы динамического управления памятью в коде модулей (ядра) имеют два главных направления использования:

1. Однократное распределение буферов данных (иногда достаточно и объёмных и сложно структурированных), которое выполняется, как правило, при начальной инициализации модуля (в сетевых драйверах часто при активизации интерфейса командой ifconfig);

2. Многократное динамическое создание-уничтожение временных структур, организованных в некоторые списочные структуры;

Первоначально мы рассматриваем механизмы первой названной группы (которые, собственно, и являются механизмами динамического управления памятью), но к концу раздела отклонимся и рассмотрим использование циклических двусвязных списков, ввиду их максимально широкого использования в ядре Linux (и призывов разработчиков ядра использовать только эти, или подобные им, там же описанные, структуры).

Динамическое выделение участка памяти размером `size` байт производится вызовом:

```
#include <linux/slab.h>
void *kmalloc( size_t size, int flags );
```

20 В зависимости от размера запрашиваемой области, и от реализации используемой библиотеки `libc`, `malloc()` может транслироваться в системный вызов `brk()`, или `mmap()` если размер запрашиваемой области велик. Убедится в этом можно написав простой цикл выделения памяти с различными запрашиваемыми размерами, и запустив тестовую программу под strace (спасибо читателям, которые обратили внимание на необходимость такого дополнения).

Выделенная таким вызовом область памяти является **непрерывной в физической** памяти.

Впервые встреченный нами параметр `flags` очень часто фигурирует в коде ядра (в отличие от пользовательского кода), и определяет то, какими характеристиками должен обладать запрошенный участок памяти. Возможных вариантов значений этого флага — великое множество, они определены в отдельном файле `<gfp.h>`, например: `__GFP_WAIT, __GFP_HIGH, __GFP_MOVABLE, ...` Рассмотрим только немногие из них, наиболее используемые, и те, которые нам активно потребуются в дальнейшем изложении:

- `GFP_KERNEL (__GFP__WAIT | __GFP__IO | __GFP__FS)` - выделение производится от имени процесса, который выполняет системный запрос в пространстве ядра — такой запрос может быть временно переводиться в пассивное состояние (блокирован).

- `GFP_ATOMIC (__GFP_HIGH)` - выделения памяти в обработчиках прерываний, тасклетах, таймерах ядра и другом коде, выполняющемся вне контекста процесса — такой не может быть блокирован (нет процесса, который активировать после блокирования). Но это означает, что в случаях, когда память могла бы быть выделена после некоторого блокирования, в данном случае будет сразу возвращаться ошибка.

Все эти флаги могут быть совместно (по «или») определены с большим числом других, например таким как:

- `GFP_DMA` - выделение памяти должно произойти в DMA-совместимой зоне памяти.

Выделенный в результате блок может быть больше размером (что никогда не создаёт проблем пользователю), и ни при каких обстоятельствах не может быть меньше. В зависимости от размера страницы архитектуры, минимальный размер возвращаемого блока может быть 32 или 64 байта, максимальный размер зависит от архитектуры, но если рассчитывать на переносимость, то, утверждается в литературе, это не должно быть больше 128 Кб; но даже уже при размерах больших 1-й страницы (несколько килобайт, для x86 — 4 Кб), есть лучше способы, чем получение памяти `kmalloc()`.

После использования всякого блока памяти он должен быть освобождён. Это касается вообще любого способа выделения блока памяти, которые ещё будут рассматриваться. Важно, чтобы освобождение памяти выполнялось вызовом, соответствующим **тому способу**, которым она выделялась. Для `kmalloc()` это:

```
void kfree( const void *ptr );
```

Повторное освобождение, или освобождение не размещённого блока приводит к тяжёлым последствиям, но `kfree( NULL )` проверяется и является совершенно допустимым.

Примечание: Требование освобождения блока памяти после использования — в ядре становится заметно актуальнее, чем в программировании пользовательских процессов: после завершения пользовательского процесса, некорректно распоряжающегося памятью, вместе с завершением процесса системе будут возвращены и все ресурсы, выделенные процессу, в том числе и область для динамического выделения памяти. Память, выделенная модулю ядра и не возвращённая явно им при выгрузке явно, никогда больше **не возвратится** под управление системы.

Альтернативным `kmalloc()` способом выделения блока памяти, но **не обязательно в непрерывной области** в физической памяти, является вызов:

```
#include <linux/vmalloc.h>
void *vmalloc( unsigned long size );
void vfree( void *addr );
```

Распределение `vmalloc()` менее производительно, чем `kmalloc()`, но может стать предпочтительнее при выделении больших блоков памяти, когда `kmalloc()` вообще не сможет выделить блок требуемого размера и завершится аварийно. Отображение страниц физической памяти в непрерывную логическую область, возвращаемую `vmalloc()`, обеспечивает MMU (аппаратная реализация управления таблицами страниц), и для пользователя разрывность физических адресов обычно незаметна и не составляет проблемы (за исключением случаев аппаратного взаимодействия с памятью, самым явным из которых является обмен по DMA).

Ещё одним (итого три) принципиально иным способом выделения памяти будут те вызовы API ялра, которые выделяют память в размере целого числа физических страниц, управляемых MMU: `__get_free_pages()` и подобные (они все имеют в своих именах суффикс `*page*`). Такие механизмы будут детально рассмотрены ниже.

Вопрос сравнения возможностей по выделению памяти различными способами актуален, но весьма запутан (по литературным источникам), так как радикально зависит от используемой архитектруры процессора, физических ресурсов оборудования (объём реальной RAM, число процессоров SMP, ...), версии ядра Linux и других факторов. Этот вопрос настолько важен и заслуживает обстоятельного тестирования, что такие оценки были проделаны для нескольких конфигураций, но в виду объёмности сам тест (архив `mtest.tgz`) и его обсуждение снесены в отдельную главу в приложениях в конце текста, а здесь приведём только сводную таблицу результатов:

Архитектура	**Максимальный выделенный блок* (байт)**		
	`kmalloc()`	`__get_free_pages()`	`vmalloc()`
Celeron (Coppermine) - 534 MHz RAM 255600 kB kernel 2.6.18.i686	131072	4194304	134217728
Genuine Intel(R), core 2 - 1.66GHz kernel 2.6.32.i686 RAM 2053828 kB	4194304	4194304	33554432
Intel(R) Core(TM)2 Quad - 2.33GHz kernel 2.6.35.x86_64 RAM 4047192 kB	4194304	4194304	2147483648

* - приведен размер не максимально возможного для размещения блока в системе, а размер максимального блока в конкретном описываемом тесте: блок вдвое большего размера выделить уже не удалось.

Из таблицы следует, по крайней мере, что в основе каждого из сравниваемых методов выделения памяти лежит свой отдельный механизм (особенно это актуально в отношении `kmalloc()` и `__get_free_pages()`), отличающийся от всех других.

Ещё одно сравнение (описано полностью там же, в отдельном приложении) — сравнение по затратам процессорных актов на одно выполнение запроса на выделение:

Размер блока (байт)	**Затраты (число процессорных тактов** , 1.6Ghz)**		
	`kmalloc()`	`__get_free_pages()`	`vmalloc()`
5*	143	890	152552
1000*	146	438	210210
4096	181	877	59626
65536	1157	940	84129
262144	2151	2382	52026
262000*	8674	4730	55612

* - не кратно `PAGE_SIZE`

** - оценки времени, связанные с диспетчеризацией процессов в системе, могут отличаться в 2-3 раза в ту или иную сторону, и могут быть только грубыми ориентирами порядка величины.

Распределители памяти

Реально распределение памяти по запросам `kmalloc()` может поддерживаться различными механизмами более низкого уровня, называемыми распределителями. Совершенно не обязательно это будет выделение

непосредственно из общей неразмеченной физической памяти, как может показаться — чаще это производится из пулов фиксированного размера, заранее размеченных специальным образом. Механизм распределителя памяти просто скрывает то, что скрыто «за фасадом» `kmalloc()`, те рутинные детали, которые стоят за выделением памяти. Кроме того, при развитии системы алгоритмы распределителя памяти могут быть заменены, но работа `kmalloc()`, на видимом потребителю уровне, останется неизменной.

Первоначальные менеджеры памяти использовали стратегию распределения, базирующуюся на heap («куча» - единое пространство для динамического выделения памяти). В этом методе большой блок памяти (heap) используется для обеспечения памятью для любых целей. Когда пользователям требуется блок памяти, они запрашивают блок памяти требуемого размера. Менеджер heap проверяет доступную память и возвращает блок. Для поиска блока менеджер использует алгоритмы либо first-fit (первый встречающийся блок, превышающий запрошенный размер), либо best-fit (блок, вмещающий запрошенный размер с наименьшим превышением). Когда блок памяти больше не нужен, он возвращаются в heap. Основная проблема этой стратегии распределения — фрагментация, и деградация системы с течением длительного времени непрерывной эксплуатации (что особо актуально для серверов). Проблемой вторичного порядка малости является высокая затратность времени для управления свободным пространством heap.

Подход, применявшийся в Linux для выделения больших регионов (называемый buddy memory allocation), выделяет по запросу блок, размером кратным степени 2, и превышающий фактический запрошенный размер (по существу, используется подход best-fit). При освобождении блока предпринимается попытка объединить в освобождаемый свободный блок все свободные соседние блоки (слить). Такой подход позволяет снизить фрагментирование и повышает эффективность управления свободным пространством. Но он может существенно увеличить непродуктивное расходование памяти.

Алгоритм распределителя, использующийся `kmalloc()` как основной механизм в версиях ядра 2.6 для текущего выделения небольших блоков памяти — это сляб алокатор (slab allocation). Слябовый распределитель впервые предложен Джефом Бонвиком (Jeff Bonwick), реализован и описан в SunOS (в середине 90-х годов). Идея такого распределителя состоит в том, что последовательные запросы на выделение памяти под объекты **равного** размера удовлетворяются из области одного кэша (сляба), а запросы на объекты другого размера (пусть отличающиеся от первого случая самым незначительным образом) - удовлетворяются из совершенно другого такого же кэша.

Примечание: Сам термин сляб переводится близко к «облицовочная плитка», и принцип очень похож: любую вынутую из плоскости плитку можно заменить другой такой же, но это только потому, что их размеры в точности совпадают.

Использование алокатора SLAB (по умолчанию) может быть отменено при новой сборке ядра (параметр `CONFIG_SLAB`). Это имеет смысл и используется для небольших и встроенных систем. При таком решении может быть включен алокатор, который называют SLOB. При таком способе участки выделяемой памяти выстраиваются в единый линейный связный список. Такой способ распределения памяти может экономить до 512KB памяти памяти (в сравнении с SLAB). Естественно, этот способ страдает названными уже недостатками, главный из которых — фрагментация.

Начиная с версии ядра 2.6.22 начинает использоваться распределитель SLUB, разработанный Кристофом Лэйметром (Christoph Lameter) из компании SGI, но это только отличающаяся **реализация** всё той же идеи распределителя SLAB. В отличие от SLOB, ориентированного на малые конфигурации, SLUB ориентирован, напротив, на системы с большими и огромными (huge) объёмами RAM. Идея состоит в том, чтобы уменьшить непроизводительные расходы на управляющие структуры слабов при их больших объёмах. Для этого управление организуется не на основе единичных страниц памяти, а на основе объединения таких страниц в группы, и управлении на базе групп страниц.

Детально смотрите какой распределитель используется по конфигурационным параметрам, с которыми собиралось ядро, например так:

```
$ cat /boot/config-2.6.32.9-70.fc12.i686.PAE | grep CONFIG_SLOB
# CONFIG_SLOB is not set
$ cat /boot/config-2.6.32.9-70.fc12.i686.PAE | grep CONFIG_SLAB
# CONFIG_SLAB is not set
CONFIG_SLABINFO=y
$ cat /boot/config-2.6.32.9-70.fc12.i686.PAE | grep CONFIG_SLUB
```

```
CONFIG_SLUB_DEBUG=y
CONFIG_SLUB=y
# CONFIG_SLUB_DEBUG_ON is not set
# CONFIG_SLUB_STATS is not set
```

Дальше детально мы будем рассматривать только слябовый распределитель SLAB.

Слябовый распределитель[21]

Текущее состояние слябового распределителя можем рассмотреть в файловой системе `/proc` (что даёт достаточно много для понимания самого принципа слябового распределения):

```
$ cat /proc/slabinfo
slabinfo - version: 2.1
# name     <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount>
<sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
...
kmalloc-8192          28     32   8192    4    8 : tunables    0    0    0 : slabdata      8      8      0
kmalloc-4096         589    648   4096    8    8 : tunables    0    0    0 : slabdata     81     81      0
kmalloc-2048         609    672   2048   16    8 : tunables    0    0    0 : slabdata     42     42      0
kmalloc-1024         489    512   1024   16    4 : tunables    0    0    0 : slabdata     32     32      0
kmalloc-512         3548   3648    512   16    2 : tunables    0    0    0 : slabdata    228    228      0
kmalloc-256          524    656    256   16    1 : tunables    0    0    0 : slabdata     41     41      0
kmalloc-128        13802  14304    128   32    1 : tunables    0    0    0 : slabdata    447    447      0
kmalloc-64         12460  13120     64   64    1 : tunables    0    0    0 : slabdata    205    205      0
kmalloc-32         12239  12800     32  128    1 : tunables    0    0    0 : slabdata    100    100      0
kmalloc-16         25638  25856     16  256    1 : tunables    0    0    0 : slabdata    101    101      0
kmalloc-8          11662  11776      8  512    1 : tunables    0    0    0 : slabdata     23     23      0
...
```

Сам принцип прост: сам сляб должен быть создан (зарегистрирован) вызовом `kmem_cache_create()`, а потом из него можно «черпать» элементы фиксированного размера (под который и был создан сляб) вызовами `kmem_cache_alloc()` (это и есть тот вызов, в который, в конечном итоге, с наибольшей вероятностью ретранслируется ваш `kmalloc()`). Все сопутствующие описания ищите в `<linux/slab.h>`. Так это выглядит на качественном уровне. А вот при переходе к деталям начинается цирк, который состоит в том, что прототип функции `kmem_cache_create()` меняется от версии к версии.

В версии 2.6.18 и **практически во всей литературе** этот вызов описан так:

```
kmem_cache_t *kmem_cache_create( const char *name, size_t size,
                  size_t offset, unsigned long flags,
                  void (*ctor)( void*, kmem_cache_t*, unsigned long flags ),
                  void (*dtor)( void*, kmem_cache_t*, unsigned long flags ) );
```

`name` — строка имени кэша;

`size` — размер элементов кэша (единый и общий для всех элементов);

`offset` — смещение первого элемента от начала кэша (для обеспечения соответствующего выравнивания по границам страниц, достаточно указать 0, что означает выравнивание по умолчанию);

`flags` — опциональные параметры (может быть 0);

`ctor, dtor` — **конструктор** и **деструктор**, соответственно, вызываются при размещении-освобождении каждого элемента, но с некоторыми ограничениями ... например, деструктор будет вызываться (финализация), но не гарантируется, что это будет поисходить сразу непосредственно после удаления объекта.

К версии 2.6.24 [5, 6] он становится другим (деструктор исчезает из описания):

```
struct kmem_cache *kmem_cache_create( const char *name, size_t size,
                        size_t offset, unsigned long flags,
```

21 В литературе и электронных публикациях мне встречались самые разнообразные русскоязычные наименования для такого распределителя, а именно, как: «слабовый», «слябовый», «слэбовый»... Термин нужно как-то именовать, и ни одна из транскрипций не лучше других, но... Более устоявшимся, а кроме того, использующимся, помимо IT, в совершенно иной области — металлургии, является произношение «слябовый», поэтому давайте его использовать.

```
                        void (*ctor)( void*, kmem_cache_t*, unsigned long flags ) );
```

Наконец, в 2.6.32 и до 3.14 можем наблюдать следующую фазу изменений (меняется прототип конструктора):

```
struct kmem_cache *kmem_cache_create( const char *name, size_t size,
                        size_t offset, unsigned long flags,
                        void (*ctor)( void* ) );
```

Это значит, что то, что компилировалось для одного ядра, перестанет компилироваться для следующего. Вообще то, это достаточно обычная практика для ядра, но к этому нужно быть готовым, а при использовании таких достаточно глубинных механизмов, руководствоваться не навыками, а изучением заголовочных файлов текущего ядра.

Из флагов создания, поскольку они также находятся в постоянном изменении, и большая часть из них относится к отладочным опциям, стоит назвать:

`SLAB_HWCACHE_ALIGN` — расположение каждого элемента в слябе должно выравниваться по строкам процессорного кэша, это может существенно поднять производительность, но непродуктивно расходуется память;

`SLAB_POISON` — начально заполняет сляб предопределённым значением (A5A5A5A5) для обнаружения выборки неинициализированных значений;

Если не нужны какие-то особые изыски, то нулевое значение будет вполне уместно для параметра `flags`.

Как для любой операции выделения, ей сопутствует обратная операция по уничтожению сляба:

```
int kmem_cache_destroy( kmem_cache_t *cache );
```

Операция уничтожения может быть успешна (здесь достаточно редкий случай, когда функция уничтожения возвращает значение результата), только если уже **все** объекты, полученные из кэша, были возвращены в него. Таким образом, модуль должен проверить статус, возвращённый `kmem_cache_destroy()`; ошибка указывает на какой-то вид утечки памяти в модуле (так как некоторые объекты не были возвращены).

После того, как кэш объектов создан, вы можете выделять объекты из него, вызывая:

```
void *kmem_cache_alloc( kmem_cache_t *cache, int flags );
```

Здесь `flags` - те же, что передаются `kmalloc()`.

Полученный объект должен быть возвращён когда в нём отпадёт необходимость :

```
void kmem_cache_free( kmem_cache_t *cache, const void *obj );
```

Несмотря на изменчивость API сляб алокатора, вы можете охватить даже диапазон версий ядра, пользуясь директивами условной трансляции препроцессора; модуль использующий такой алокатор может выглядеть подобно следующему (архив `slab.tgz`):

slab.c :

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

MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_VERSION( "5.2" );

static int size = 7;  // для наглядности - простые числа
module_param( size, int, 0 );
static int number = 31;
module_param( number, int, 0 );

static void* *line = NULL;
```

```
static int sco = 0;
static
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,31)
void co( void* p ) {
#else
void co( void* p, kmem_cache_t* c, unsigned long f ) {
#endif
   *(int*)p = (int)p;
   sco++;
}
#define SLABNAME "my_cache"
struct kmem_cache *cache = NULL;

static int __init init( void ) {
   int i;
   if( size < sizeof( void* ) ) {
      printk( KERN_ERR "invalid argument\n" );
      return -EINVAL;
   }
   line = kmalloc( sizeof(void*) * number, GFP_KERNEL );
   if( !line ) {
      printk( KERN_ERR "kmalloc error\n" );
      goto mout;
   }
   for( i = 0; i < number; i++ )
      line[ i ] = NULL;
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,32)
   cache = kmem_cache_create( SLABNAME, size, 0, SLAB_HWCACHE_ALIGN, co, NULL );
#else
   cache = kmem_cache_create( SLABNAME, size, 0, SLAB_HWCACHE_ALIGN, co );
#endif
   if( !cache ) {
      printk( KERN_ERR "kmem_cache_create error\n" );
      goto cout;
   }
   for( i = 0; i < number; i++ )
      if( NULL == ( line[ i ] = kmem_cache_alloc( cache, GFP_KERNEL ) ) ) {
         printk( KERN_ERR "kmem_cache_alloc error\n" );
         goto oout;
      }
   printk( KERN_INFO "allocate %d objects into slab: %s\n", number, SLABNAME );
   printk( KERN_INFO "object size %d bytes, full size %ld bytes\n", size, (long)size * number );
   printk( KERN_INFO "constructor called %d times\n", sco );
   return 0;
oout:
   for( i = 0; i < number; i++ )
      kmem_cache_free( cache, line[ i ] );
cout:
   kmem_cache_destroy( cache );
mout:
   kfree( line );
   return -ENOMEM;
}
module_init( init );

static void __exit exit( void ) {
   int i;
   for( i = 0; i < number; i++ )
```

```
      kmem_cache_free( cache, line[ i ] );
   kmem_cache_destroy( cache );
   kfree( line );
}
module_exit( exit );
```

Вот как выглядит выполнение этого размещения (картина весьма поучительная, поэтому остановимся на ней подробнее) в достаточно свежей версии ядра (32-разрядного):

```
$ uname -r
3.13.6-200.fc20.i686
$ sudo insmod ./slab.ko
$ dmesg | tail -n3
[11648.079727] allocate 31 objects into slab: my_cache
[11648.079735] object size 7 bytes, full size 217 bytes
[11648.079737] constructor called 257 times
$ sudo cat /proc/slabinfo | grep my_
my_cache     256    256     16  256    1 : tunables    0    0    0 : slabdata      1      1      0
```

Итого: объекты размером 7 байт благополучно разместились в новом слябе с именем `my_cache`, отображаемом в `/proc/slabinfo`, организованным с размером элементов 16 байт (эффект выравнивания?), конструктор при размещении 31 таких объектов вызывался 257 раз. Обратим внимание на чрезвычайно важное обстоятельство: при создании сляба никаким образом не указывается реальный или **максимальный** объём памяти, находящейся под управлением этого сляба: это динамическая структура, «добирающая» столько **страниц** памяти, сколько нужно для поддержания размещения требуемого числа элементов данных (с учётом их размера). Увеличенное число вызовов конструктора можно отнести: а). на необходимость переразмещения существующих элементов при последующих запросах, б). эффекты SMP (2 ядра) и перераспределения данных между процессорами. Проверим тот же тест на однопроцессорном Celeron и более старой версии ядра:

```
$ uname -r
2.6.18-92.el5
$ sudo /sbin/insmod ./slab.ko
$ /sbin/lsmod | grep slab
slab                    7052  0
$ dmesg | tail -n3
allocate 31 objects into slab: my_cache
object size 7 bytes, full size 217 bytes
constructor called 339 times
$ cat /proc/slabinfo | grep my_
my_cache      31    339      8  339    1 : tunables  120   60    8 : slabdata      1      1      0
$ sudo /sbin/rmmod slab
```

Число вызовов конструктора не уменьшилось, а даже возросло, а вот размер объектов, под который создан сляб, изменился с 16 на 8.

Примечание: Если рассмотреть 3 первых поля вывода `/proc/slabinfo`, то и в первом и во втором случае видно, что под сляб размечено некоторое фиксированное количество фиксированных объекто-мест (339 в последнем примере), которые укладываются в некоторый начальный объём сляба меньше или порядка 1-й страницы физической памяти.

И для полноты картины для 64-бит реализации (размер объектов 11 байт, 7 байт слишком короткий объект для 64-битной реализации **теста**):

```
$ uname -r
3.14.4-200.fc20.x86_64
$ sudo insmod ./slab.ko size=11
$ dmesg | tail -n3
[11835.972001] allocate 31 objects into slab: my_cache
[11835.972007] object size 11 bytes, full size 341 bytes
[11835.972009] constructor called 129 times
$ sudo cat /proc/slabinfo | grep my_
my_cache     128    128     32  128    1 : tunables    0    0    0 : slabdata      1      1      0
$ sudo rmmod slab
```

Тот же тест при больших размерах объектов и их числе:

```
$ sudo insmod ./slab.ko size=1111 number=300
$ dmesg | tail -n3
allocate 300 objects into slab: my_cache
object size 1111 bytes, full size 333300 bytes
constructor called 330 times
$ sudo rmmod slab
$ sudo insmod ./slab.ko size=1111 number=3000
$ dmesg | tail -n3
allocate 3000 objects into slab: my_cache
object size 1111 bytes, full size 3333000 bytes
constructor called 3225 times
$ sudo rmmod slab
```

Примечание: Последний рассматриваемый пример любопытен в своём поведении. Вообще то «завалить» операционную систему Linux — ничего не стоит, когда вы пишете модули ядра. В противовес тому, что за несколько лет плотной (почти ежедневной) работы с микроядерной операционной системой QNX мне так и не удалось её «завалить» ни разу (хотя попытки и предпринимались). Это, попутно, к цитировавшемуся ранее эпиграфом высказыванию Линуса Торвальдса относительно его оценок микроядерности. Но сейчас мы не о том... Если погонять показанный тест с весьма большим размером блока и числом блоков для размещения (заметно больше показанных выше значений), то можно наблюдать прелюбопытную ситуацию: нет, система не виснет, но распределитель памяти настолько активно отбирает память у системы, что постепенно угасают все графические приложения, потом и вся подсистема X11 ... но остаются в живых чёрные текстовые консоли, в которых даже живут мыши. Интереснейший получается эффект[22].

Ещё одна вариация на тему распределителя памяти, в том числе и сляб-алокатора — механизм пула памяти:

```
#include <linux/mempool.h>
mempool_t *mempool_create( int min_nr,
                           mempool_alloc_t *alloc_fn, mempool_free_t *free_fn,
                           void *pool_data );
```

Пул памяти сам по себе вообще не является алокатором, а всего лишь является **интерфейсом** к алокатору (**к любому**, к тому же кэшу, например). Само наименование «пул» (имеющее схожий смысл в разных контекстах и разных операционных системах) предполагает, что такой механизм будет всегда поддерживать «в горячем резерве» некоторое количество объектов для распределения. Аргумент вызова `min_nr` является тем минимальным числом выделенных объектов, которые пул должен всегда поддерживать в наличии. Фактическое выделение и освобождение объектов по запросам обслуживают `alloc_fn()` и `free_fn()`, которые предлагается написать пользователю, и которые имеют такие прототипы:

```
typedef void* (*mempool_alloc_t)( int gfp_mask, void *pool_data );
typedef void (*mempool_free_t)( void *element, void *pool_data );
```

Последний параметр `mempool_create()` - `pool_data` (блок данных) передаётся последним параметром в вызовы `alloc_fn()` и `free_fn()`.

Но обычно просто дают обработчику-распределителю ядра выполнить за нас задачу — объявлено (`<linux/mempool.h>`) несколько групп API для разных распределителей памяти. Так, например, существуют две функции, например, (`mempool_alloc_slab()` и `mempool_free_slab()`), ориентированный на рассмотренный уже сляб алокатор, которые выполняют соответствующие согласования между прототипами выделения пула памяти и `kmem_cache_alloc()` и `kmem_cache_free()`. Таким образом, код, который инициализирует пул памяти, который будет использовать сляб алокатор для управления памятью, часто выглядит следующим образом:

```
// создание нового сляба
```

22 Что напомнило высказывание классика отечественного юмора М. Жванецкого: «А вы не пробовали слабительное со снотворным? Удивительный получается эффект!».

```
kmem_cache_t *cache = kmem_cache_create( ... );
// создание пула, который будет распределять память из этого сляба
mempool_t *pool = mempool_create( MY_POOL_MINIMUM, mempool_alloc_slab, mempool_free_slab, cache );
```

После того, как пул был создан, объекты могут быть выделены и освобождены с помощью:

```
void *mempool_alloc_slab( gfp_t gfp_mask, void *pool_data );
void mempool_free_slab( void *element, void *pool_data );
```

После создания пула памяти функция выделения будет вызвана достаточное число раз для создания пула предопределённых объектов. После этого вызовы `mempool_alloc_slab()` пытаются получить новые объекты от функции выделения - возвращается один из предопределённых объектов (если таковые сохранились). Когда объект освобождён `mempool_free_slab()`, он сохраняется в пуле если количество предопределённых объектов в настоящее время ниже минимального, в противном случае он будет возвращён в систему.

Примечание: Такие же группы API есть для использования в качестве распределителя памяти для пула `kmalloc()` (`mempool_kmalloc()`) и страничного распределителя памяти (`mempool_alloc_pages()`).

Размер пула памяти может быть динамически изменён:

```
int mempool_resize( mempool_t *pool, int new_min_nr, int gfp_mask );
```

- в случае успеха этот вызов изменяет размеры пула так, чтобы иметь по крайней мере `new_min_nr` объектов.

Когда пул памяти больше не нужен он возвращается системе:

```
void mempool_destroy( mempool_t *pool );
```

В архиве примеров присутствует пример модуля (файл `pool.c`), демонстрирующий реализацию пула памяти как над кэшем (`mempool_create_slab_pool()`), так и над распределителем (`mempool_kmalloc()`), позволяющий наблюдать создание пула, размещение в нём элементов и тестирование корректности размещённых элементов. Тест позволяет наглядно поэкспериментировать с техникой создания пулов. Но его код мы не станем рассматривать.

Страничное выделение

Когда нужны блоки больше одной машинной страницы и кратные целому числустраниц:

```
#include <linux/gfp.h>
struct_page * alloc_pages( gfp_t gfp_mask, unsigned int order )
```

Такой вызов выделяет `2**order` смежных страниц (**непрерывный** участок) физической памяти. Полученный физический адрес требуется конвертировать в логический для использования:

```
void *page_address( struct_page * page )
```

Если не требуется физический адрес, то сразу получить логический позволяют:

`unsigned long __get_free_page( gfp_t gfp_mask );` - выделяет одну страницу;

`unsigned long get_zeroed_page( gfp_t gfp_mask );` - выделяет одну страницу и заполняет её нулями;

`unsigned long __get_free_pages( gfp_t gfp_mask, unsigned int order );` - выделяет несколько (`2**order`) последовательных страниц непрерывной областью;

Принципиальное отличие выделенного таким способом участка памяти от выделенного `kmalloc()` (при равных размерах запрошенных участков для сравнения) состоит в том, что участок, выделенный механизмом страничного выделения и будет всегда **выровнен на границу страницы**.

В любом случае, выделенную страничную область после использования необходимо возвратить по логическому или физическому адресу (способом в точности симметричным тому, которым выделялся участок!):

```
void __free_pages( unsigned long addr, unsigned long order );
void free_page( unsigned long addr );
```

```
void free_pages( unsigned long addr, unsigned long order );
```

При попытке освободить другое число страниц чем то, что выделялось, карта памяти становится повреждённой и система позднее будет **разрушена**.

Выделение больших буферов

Для выделения экстремально больших буферов, иногда описывают и рекомендуют технику выделения памяти непосредственно при загрузке системы (ядра). Но эта техника доступна только модулям, загружаемым с ядром (при начальной загрузке), далее они не подлежат выгрузке. Техника, приемлемая для команды разработчиков ядра, но сомнительная в своей ценности для сторонних разработчиков модулей ядра. Тем не менее, вскользь упомним и её. Для её реализации есть такие вызовы:

```
#include <linux/bootmem.h>
void *alloc_bootmem( unsigned long size );
void *alloc_bootmem_low( unsigned long size );
void *alloc_bootmem_pages( unsigned long size);
void *alloc_bootmem_low_pages( unsigned long size );
```

Эти функции выделяют либо целое число страниц (если имя функции заканчивается на `_pages`), или не выровненные странично области памяти.

Освобождение памяти, выделенной при загрузке, производится даже в ядре крайне редко: сам модуль выгружен быть не может, а почти наверняка получить освобождённую память позже, при необходимости, он будете уже не в состоянии. Однако, существует интерфейс для освобождения и этой памяти:

```
void free_bootmem( unsigned long addr, unsigned long size );
```

Динамические структуры и управление памятью

Статический и динамический способ размещения структур данных имеют свои положительные и отрицательные стороны, главными из которых принято считать: а). статическая память: надёжность, живучесть и меньшая подверженность ошибкам; б). динамическая память: гибкость использования. Использование динамических структур всегда требует того или иного механизма управления памятью: создание и уничтожение терминальных элементов динамически увязываемых структур.

Циклический двусвязный список

Чтобы уменьшить количество дублирующегося кода, разработчики ядра создали (с ядра 2.6) стандартную реализацию кругового, двойного связного списка; всем другим нуждающимся в манипулировании списками (даже простейшими линейными односвязными, к примеру) рекомендуется разработчиками использовать это средство. Именно поэтому они заслуживают отдельного рассмотрения.

= = = = = = = = = =

здесь может быть рисунок (если я когда нарисую): односвязный список head заканчивающийся NULL, и эквивалентный циклический двухсвязный список head.

= = = = = = = = = =

Примечание: При работе с интерфейсом связного списка всегда следует иметь в виду, что функции списка выполняют без блокировки. Если есть вероятность того, что драйвер может попытаться выполнить на одном списке конкурентные операции, вашей обязанностью является реализация схемы блокировки. Результаты некорректного параллельного доступа (повреждённые структуры списка, потеря данных, паники ядра) как правило, трудно диагностировать.

Чтобы использовать механизм списка, ваш драйвер должен подключить файл `<linux/list.h>`. Этот файл определяет простую структуру типа `list_head`:

```
struct list_head {
   struct list_head *next, *prev;
};
```

Для использования в вашем коде средства списка Linux, необходимо лишь вставлять `list_head` внутри собственных структур, входящих в список, например:

```
struct todo_struct {
   struct list_head list;
   int priority;
   /* ... добавить другие зависимые от задачи поля */
};
```

Эта условная структура будет дальше использоваться для иллюстрации работы макросов, обслуживающих связные списки.

Заголовки списков должны быть проинициализированы перед использованием с помощью макроса `INIT_LIST_HEAD`. Заголовок списка может быть объявлен и проинициализирован так (динамически):

```
struct list_head todo_list;
INIT_LIST_HEAD( &todo_list );
```

Альтернативно, списки могут быть созданы и проинициализированы статически при компиляции:

```
LIST_HEAD( todo_list );
```

Некоторые функции для работы со списками определены в `<linux/list.h>`. Как мы видим, API работы с циклическим списком позволяет выразить любые операции с элементами списка, не вовлекая в операции манипулирование с внутренними полями связи списка; это очень ценно для сохранения целостности списков:

```
list_add( struct list_head *new, struct list_head *head );
```

Этот макрос добавляет новую запись `new` сразу же **после головы** списка (head), как правило, в начало списка. Таким образом, она может быть использована для создания стеков. Однако, следует отметить, что голова не должна быть номинальной головой списка; если вы передадите структуру `list_head`, которая окажется где-то в середине списка, новая запись пойдёт сразу после неё. Так как списки Linux являются круговыми, голова списка обычно не отличается от любой другой записи.

```
list_add_tail( struct list_head *new, struct list_head *head );
```

Этот макрос добавляет элемент `new` **перед головой** данного списка (head), в **конец** списка, другими словами, `list_add_tail()` может, таким образом, быть использована для создания очередей первый вошёл - первый вышел.

```
list_del( struct list_head *entry );
list_del_init( struct list_head *entry );
```

Этот макрос **удаляет** указанную данная запись из списка (обратим внимание, что указывать сам список в параметрах не нужно). Если эта запись может быть когда-либо вставленной в другой список, вы должны использовать `list_del_init()`, которая инициализирует заново указатели связного списка.

```
list_move( struct list_head *entry, struct list_head *head );
list_move_tail( struct list_head *entry, struct list_head *head );
```

Этот макрос удаляет указанную запись из своего текущего положения и **перемещает** (запись) в начало списка. Чтобы переместить запись в конец списка используется `list_move_tail()`.

```
list_empty( struct list_head *head );
```

Возвращает ненулевое значение, если данный список пуст.

```
list_splice( struct list_head *list, struct list_head *head );
```

Объединение двух списков с **вставкой** нового списка `list` сразу **после** головы `head`.

Структуры `list_head` хороши для реализации линейных списков, но использующие его программы почти всегда больше заинтересованы в некоторых более крупных структурах, которые увязываются в список

как его узлы. При этом широко используется макрос `list_entry()`, который по значению указателя структуры `list_head` восстанавливает указатель на экземпляр структуры, которая именно его содержит. Он вызывается следующим образом:

```
list_entry( struct list_head *ptr, type_of_struct, field_name );
```

Здесь `ptr` является указателем на используемую структуру `list_head`, `type_of_struct` является указаним **имени типа** структуры, содержащей в себе этот элемент связи (`ptr`), и `field_name` является **именем поля** связи (`struct list_head`) в этой структуре.

В показанной раньше для примера структуре `todo_struct` поле списка **имеет имя** `list`. Таким образом, когда мы хотели бы получить по указателю поля связи `listptr` — содержащую это поле структуру типа `struct todo_struct`, то могли бы выразить это такой строкой:

```
struct todo_struct *todo_ptr = list_entry( listptr, struct todo_struct, list );
```

Реализовать подобное (разработчикам ядра) можно только **макросом**, сделать это вызовом **функции** было бы нельзя. Макрос `list_entry()` несколько необычен и требует некоторого времени, чтобы привыкнуть, но его не так сложно использовать.

Обход связных списков достаточно прост: надо только использовать указатели `prev` и `next`. В качестве примера предположим, что мы хотим добавлять новый элемент в список так, чтобы сохранять список объектов `todo_struct`, отсортированный в порядке убывания поля `priority`. Функция добавления новой записи будет выглядеть примерно следующим образом:

```
void todo_add_entry( struct todo_struct *new ) {
   struct list_head *ptr;
   struct todo_struct *entry;
   /* голова списка поиска: todo_list */
   for( ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next ) {
      entry = list_entry( ptr, struct todo_struct, list );
      if( entry->priority < new->priority ) {
         list_add_tail( &new->list, ptr);
         return;
      }
   }
   list_add_tail( &new->list, &todo_list );
}
```

Однако, как правило, лучше использовать один из набора **предопределённых макросов** для операций, которые перебирают списки (итераторов). Например, предыдущий цикл мог бы быть написать так:

```
void todo_add_entry( struct todo_struct *new ) {
   struct list_head *ptr;
   struct todo_struct *entry;
   list_for_each( ptr, &todo_list ) {
      entry = list_entry( ptr, struct todo_struct, list );
      if( entry->priority < new->priority) {
         list_add_tail( &new->list, ptr );
         return;
      }
   }
   list_add_tail( &new->list, &todo_list );
}
```

Использование предусмотренных макросов помогает избежать простых ошибок программирования. Разработчики этих макросов также приложили некоторые усилия, чтобы они выполнялись производительно. Существует несколько вариантов **итераторов**:

```
list_for_each( struct list_head *cursor, struct list_head *list )
```

Этот макрос создаёт цикл for, который выполняется по одному разу с указателем `cursor`, присвоенным поочерёдно указателю на каждую последовательную позицию в списке (будьте осторожны с изменением списка при итерациях через него).

```
list_for_each_prev( struct list_head *cursor, struct list_head *list )
```

Эта версия выполняет итерации назад по списку.

```
list_for_each_safe( struct list_head *cursor, struct list_head *next, struct list_head *list )
```

Если операции в цикле могут **удалить** запись в списке, используйте эту версию: он просто сохраняет следующую запись в списке в `next` (дополнительный параметр) для продолжения цикла, поэтому не запутается, если запись, на которую указывает `cursor`, удаляется.

```
list_for_each_entry( type *cursor, struct list_head *list, member )
list_for_each_entry_safe( type *cursor, type *next, struct list_head *list, member )
```

Такие **макросы** облегчают процесс просмотр списка, содержащего структуры данного типа `type`. Здесь `cursor` является указателем на экземпляр охватывающей структуры (результат), содержащей поле связи, `member` является **именем структуры** связи `list_head` внутри содержащей структуры. С этими макросами нет необходимости помещать внутри цикла вызов макроса `list_entry()`.

В заголовках `<linux/list.h>` определены ещё ряд деклараций для описания динамических структур и манипуляций с ними.

Модуль использующий динамические структуры

Ниже показан пример модуля ядра (архив `list.tgz`), строящий, использующий и утилизирующий простейшую динамическую структуру в виде односвязного списка:

mod_list.c :

```
#include <linux/slab.h>
#include <linux/list.h>
MODULE_LICENSE( "GPL" );
static int size = 5;
module_param( size, int, S_IRUGO | S_IWUSR );    // размер списка как параметр модуля
struct data {
    int n;
    struct list_head list;
};
void test_lists( void ) {
   struct list_head *iter, *iter_safe;
   struct data *item;
   int i;
   LIST_HEAD( list );
   for( i = 0; i < size; i++ ) {
      item = kmalloc(  sizeof(*item), GFP_KERNEL );
      if( !item ) goto out;
      item->n = i;
      list_add( &(item->list), &list );
   }
   list_for_each( iter, &list ) {
      item = list_entry( iter, struct data, list );
      printk( KERN_INFO "[LIST] %d\n", item->n );
   }
out:
   list_for_each_safe( iter, iter_safe, &list ) {
      item = list_entry( iter, struct data, list );
      list_del( iter );
      kfree( item );
   }
}
static int __init mod_init( void ) {
   test_lists();
```

```
    return -1;
}
module_init( mod_init );

$ sudo /sbin/insmod ./mod_list.ko size=3
insmod: error inserting './mod_list.ko': -1 Operation not permitted
$ dmesg | tail -n3
[LIST] 2
[LIST] 1
[LIST] 0
```

Сложно структурированные данные

Одной только ограниченной структуры данных `struct list_head` достаточно для построения динамических структур практически произвольной степени сложности, как, например, сбалансированные B-деревья, красно-чёрные списки и другие. Именно поэтому ядро 2.6 было полностью переписано в части используемых списковых структур на использование `struct list_head`. Вот каким простым образом может быть представлено с использованием этих структур бинарное дерево:

```
struct my_tree {
    struct list_head left, right; /* левое и правое поддеревья */
    /* ... добавить другие зависимые от задачи поля */
};
```

Не представляет слишком большого труда для такого представления создать собственный набор функций его создания-инициализации и манипуляций с узлами такого дерева.

Обсуждение по инициализации объектов ядра

При обсуждении заголовков списков, было показано две (альтернативно, на выбор) возможности объявить и инициализировать такой заголовок списка:

- статический (переменная объявляется макросом и тут же делаются все необходимые для инициализации манипуляции):

```
LIST_HEAD( todo_list );
```

- динамический (переменная сначала объявляется, как и любая переменная элементарного типа, например, целочисленного, а только потом инициализируется указанием её адреса):

```
struct list_head todo_list;
INIT_LIST_HEAD( &todo_list );
```

Такая же дуальность (статика + динамика) возможностей будет наблюдаться далее много раз, например, относительно всех примитивов синхронизации (мютексов, семафоров, ...). Напрашивается вопрос: зачем такая избыточность возможностей и когда что применять? Дело в том, что очень часто (в большинстве случаев) такие переменные не фигурируют в коде автономно, а встраиваются в более сложные объемлющие структуры данных. Вот для таких встроенных объявлений и будет годиться только динамический способ инициализации. Единичные переменные проще создавать статически.

Служба времени

«Всему своё время и время всякой вещи под небом»
«Екклисиаст, III:1»

Как-то так сложилось мнение, что только-что законченная нами в рассмотрении тема динамического управления памятью является сложной темой. Но она то как раз является относительно простой. По настоящему сложной и неисчерпаемой темой (в любой операционной системе!) является служба времени. Ещё одной особенностью подсистемы времени, которой мы и воспользуемся не раз, является то, что нюансы

поведения службы времени, как ни одной другой службы, можно с одинаковым успехом анализировать как в пространстве ядра, так и в пользовательском пространстве — они и там и там выявляются аналогично, а мелкие различия только лучше позволяют понять наблюдаемое. Поэтому многие вопросы, относящиеся ко времени, можно проще изучать на коде пользовательского адресного пространства, чем ядра.

Во всех функциях времени, основным принципом остаётся положение, сформулированное расширением реального времени POSIX 1003b : временные интервалы **никогда** не могут быть короче, чем затребованные, но могут быть **сколь угодно больше** затребованных.

Информация о времени в ядре

Сложность подсистемы времени усугубляется тем, что для повышения точности или функциональности API времени, разработчики привлекают несколько разнородных и не синхронизированных источников временных меток (все, которые позволяет та или иная аппаратная платформа). Точный набор набор таких дополнительных возможностей определяется аппаратными возможностями самой платформы, более того, на одной и той же архитектурной платформе, например, x86 набор и возможности задатчиков времени обновляются с развитием и изменяются каждые 2-3 года, а, соответственно, изменяется всё поведение в деталях подсистемы времени. Но какие бы не были платформенные или архитектурные различия, нужно отчётливо разделять обязательный в любых условиях **системный таймер** и **дополнительные источники** информации времени в системе. Всё относящееся к системному таймеру является основой функционирования Linux и не зависит от платформы, все остальные альтернативные возможности являются зависимыми от реализации на конкретной платформе.

Ядро следит за течением времени с помощью прерываний системного таймера. Прерывания таймера генерируются аппаратно через постоянные интервалы **системным** таймером; этот интервал программируется во время загрузки Linux записью соответствующего коэффициента в аппаратный счётчик-делитель. Делается это в соответствии со одной из самых фундаментальных констант ядра — константы периода компиляции (определённой директивой `#define`) с именем `HZ` (tick rate). Значение этой константы, вообще то говоря, является архитектурно-зависимой величиной, определено оно в `<linux/param.h>`, значения по умолчанию в исходных текстах ядра имеют диапазон от 50 до 1200 тиков в секунду на различном реальном оборудовании, снижаясь до 24 в программных эмуляторах. Но для большинства платформ для ядра 2.6 выбраны значения HZ=1000, что соответствует периоду следования системных тиков в 1 миллисекунду — это достаточно мало для обеспечения хорошей динамики системы, но очень много в сравнении с временем выполнения единичной команды процессора. По прерыванию системного таймера происходят все важнейшие события в системе:

- Обновление значения времени работы системы (uptime), абсолютного времени (time of day);
- Проверка, не израсходовал ли текущий процесс свой квант времени, и если израсходовал, то выполнятся планирование выполнения нового процесса;
- Для SMP-систем выполняется проверка балансировки очередей выполнения планировщика, и если они не сбалансированы, то производится их балансировка;
- Выполнение обработчиков всех созданных динамических таймеров, для которых истек период времени;
- Обновление статистики по использованию процессорного времени и других ресурсов.

Снижение периода следования системных тиков обеспечивает лучшие динамические характеристики системы (например, в системе реального времени QNX период следования системных тиков может быть ужат до 10 микросекунд), но ниже какого-то предела уменьшение значения этого периода начинает значительно снижать общую производительность операционной системы (возрастают непроизводительные расходы на обслуживание частых прерываний).

Источник прерываний системного таймера

Источник прерываний системного таймера (определяющий последовательность тиков частоты `HZ`, и подсчитываемых в счётчике `jiffies`) — аппаратная микросхема системного таймера. В архитектуре x86 задатчиком есть микросхема по типу Intel 82C54, работающая от отдельного кварца стандартизованной частоты 1.1931816Мгц; далее эта частот делится на целочисленный делитель, записываемый в регистры 82C54.

```
$ cat /proc/interrupts
```

```
        CPU0
  0:    5737418          XT-PIC  timer
...
  8:          1          XT-PIC  rtc
```

При выбранном значении делителя 1193 обеспечивается частота последовательности прерываний таймера максимально близкой к выбранному в заголовочной файле `<linux/param.h>` значению `HZ` — 1000.152hz, что соответствует периоду (ticksize) 999847нс (рассогласование с 1мс составляет -0,0153%).

Примечание: ближайшее соседнее значение делителя 1194 даёт частоту и период 999.314hz и 1000680нс (рассогласование с 1мс составляет +0,068%), соответственно, но всегда используется значение периода с недостатком, в противном случае задержка, величина которой определена как:

```
struct timespec ts = { 0 /* секунды */, 1000500 /* нано секунды */ };
```

- могла бы (при некоторых прогонах) завершиться на первом тике, что противоречит требованиям POSIX 1003b о том, что временной интервал может быть больше, но ни в коем случае не меньше указанного!

Тот же принцип формирования периода системных тиков соблюдается и на любой другой аппаратной платформе, на которой выполняется Linux: целочисленный делитель счётчика, задающий максимальное приближенное аппаратное значения частоты к выбранному значению константы HZ с избытком (то есть период системного таймера с недостатком к значению 1/HZ). Это будет существенно важно для толкования полученных нами вскорости результатов тестов.

Дополнительные источники информации о времени

Кроме системного таймера, в системе может быть (в большой зависимости от процессорной архитектуры и степени развитости этой архитектуры, для процессоров x86, например) ещё несколько источников событий для временной шкалы: часы реального времени (RTC), таймеры контроллеров прерываний APIC, специальные счётчики процессорных тактов и другие. Эти источники временных шкал могут использоваться для уточнения значений интервалов системного таймера. С развитием и расширением возможностей любой аппаратной платформы, разработчики ядра стараются подхватить и использовать любые новые появившиеся аппаратные механизмы. Связано это желание с тем, что, как уже было сказано, стандартный период **системного** таймера чрезвычайно велик (но 2 порядка и более) времени выполнения единичной команды процессора - в масштабе времён выполнения команд период системного таймера очень большая величина, и интервальные значения, измеренные в шкале системных тиков, пытаются разными способами уточнить с привлечением дополнительных источников. Это приводит к тому, что близкие версии ядра на однотипном оборудовании разных лет изготовления могут использовать существенно различающиеся точности для оценивания временных интервалов:

- 2-х ядерный ноутбук уровня 2007г. :

```
$ cat /proc/interrupts
           CPU0       CPU1
  0:    3088755          0   IO-APIC-edge      timer
...
  8:          1          0   IO-APIC-edge      rtc0
...
 LOC:    2189937    2599255   Local timer interrupts
...
 RES:    1364242    1943410   Rescheduling interrupts
...
$ uname -r
2.6.32.9-70.fc12.i686.PAE
```

- 4-х ядерный поцесор образца 2011г. :

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:        127          0          0          0   IO-APIC-edge      timer
...
```

```
   8:          0          0          0          0   IO-APIC-edge      rtc0
...
 LOC:  460580288  309522125 2269395963  161407978   Local timer interrupts
...
 RES:     919591     983178     315144     626188   Rescheduling interrupts
...
$ uname -r
2.6.35.11-83.fc14.i686
```

- 4-х ядерный ноутбук 2014г., 64-бит система:

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:         32          0          0          0   IO-APIC-edge      timer
...
  8:          0          1          0          0   IO-APIC-edge      rtc0
...
LOC:   10631735   11158944   10831185   10010565   Local timer interrupts
...
RES:      89820      58446      69061      36388   Rescheduling interrupts
...
$ uname -r
3.13.10-200.fc20.x86_64
```

В общем виде это выглядит так: если на каком-то конкретном компьютере обнаружен тот или иной источник информации времени, то он будет использоваться для уточнения интервальных значений, если нет — то будет шкала системных тиков. Это означает, что на подобных экземплярах оборудования (различных экземплярах x86 десктопов, например, как наиболее массовых) один и тот же код, работающий с API времени, будет давать различные результаты (что очень скоро мы увидим).

Три класса задач во временной области

Существуют три класса задач, решаемых по временной области, это :

1. Измерение временных интервалов;
2. Выдержка пауз во времени;
3. Отложенные во времени действия.

В отношении задач каждого класса существуют свои ограничения, возможности использования дополнительных источников уточнения информации о времени и, как следствие, предельное временное разрешение, которое может быть достигнуто в каждом классе задач. Например, отложенные во времени действия (действия, планируемые по таймерам), чаще всего, привязываются к шкале системных тиков (тем же точкам во времени, где происходит и диспетчирование выполняемых потоков системой) — разрешение такой шкалы соответствует системным тикам, и это **миллисекундный** диапазон. Напротив, пассивное измерение временного интервала между двумя точками отсчёта в коде программы, вполне может основываться на таких простейших механизмах, как считанные значения счётчика тактовой частоты процессора, а это может обеспечивать разрешение шкалы времени в **наносекундном** диапазоне. Разница в разрешении между двумя рассмотренными случаями — 6 порядков!

Измерения временных интервалов

Пассивное измерение **уже прошедших** интервалов времени (например, для оценивания потребовавшихся трудозатрат, профилирования) — это простейший класс задач, требующих оперирования с функциями времени. Если мы зададимся целью измерять прошедший временной интервал в шкале системного таймера, то вся измерительная процедура реализуется простейшим образом:

```
u32 j1, j2;
...
j1 = jiffies;                     // начало измеряемого интервала
...
j2 = jiffies;                    // завершение измеряемого интервала
int interval = ( j2 – j1 ) / HZ; // интервал в секундах
```

Что мы и используем в первом примере модуля (архив `time.tgz`) из области механизмов времени, этот модуль всего лишь замеряет интервал времени, которое он был загружен в ядро:

interv.c :

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

static u32 j;

static int __init init( void ) {
   j = jiffies;
   printk( KERN_INFO "module: jiffies on start = %X\n", j );
   return 0;
}

void cleanup( void ) {
   static u32 j1;
   j1 = jiffies;
   printk( KERN_INFO "module: jiffies on finish = %X\n", j1 );
   j = j1 - j;
   printk( KERN_INFO "module: interval of life = %d\n", j / HZ );
   return;
}

module_init( init );
module_exit( cleanup );
```

Вот результат выполнения такого модуля — обратите внимание на хорошее соответствие временного интервала (15 секунд), замеренного в пространстве пользователя (командным интерпретатором) и интервального измерения в ядре:

```
$ date; sudo insmod ./interv.ko
Сбт Июл 23 23:18:45 EEST 2011
$ date; sudo rmmod interv
Сбт Июл 23 23:19:01 EEST 2011
$ dmesg | tail -n 50 | grep module:
module: jiffies on start = 131D080
module: jiffies on finish = 1320CCD
module: interval of life = 15
```

Счётчик системных тиков `jiffies` и специальные функции для работы с ним описаны в `<linux/jiffies.h>`, хотя вы обычно будете просто подключать `<linux/sched.h>`, который автоматически подключает `<linux/jiffies.h>`. Определяются два подобных имени: `jiffies` и `jiffies_64` (независимо от разрядности системы), представляющие значение в 32 и 64 бит, соответственно. Излишне говорить, что `jiffies` и `jiffies_64` должны рассматриваться как только читаемые. Счётчик `jiffies` считает системные тики от момента последней загрузки системы.

При последовательных считываниях `jiffies` может быть зафиксировано его переполнение (32 бит значение). Чтобы не заморачиваться с анализом, ядро предоставляет четыре однотипных макроса для сравнения двух значений счетчика импульсов таймера, которые корректно обрабатывают переполнение счетчиков. Они определены в файле `<linux/jiffies.h>` следующим образом:

```
#define time_after( unknown, known ) ( (long)(known) - (long)(unknown) < 0 )
#define time_before( unknown, known ) ( (long)(unknown) - (long)(known) < 0 )
#define time_after_eq( unknown, known ) ( (long)(unknown) - (long)(known) >= 0 )
#define time_before_eq( unknown, known ) ( (long)(known) - (long)(unknown) >= 0 )
```

Здесь unknown - это обычно значение переменной jiffies, а параметр known - значение, с которым его необходимо сравнить. Макросы возвращают значение true, если выполняются **соотношения** моментов времени unknown и known, в противном случае возвращается значение false.

Иногда, однако, необходимо обмениваться представлением времени с программами пользовательского пространства, которые, как правило, предоставляют значения времени структурами timeval и timespec. Эти две структуры предоставляют точное значение времени как структуру из двух чисел: секунды и микросекунды используются в старой и популярной структуре timeval, а в новой структуре timespec используются секунды и наносекунды. Ядро экспортирует четыре вспомогательные функции для преобразования значений времени выраженного в jiffies из/в эти структуры:

```
#include <linux/time.h>
unsigned long timespec_to_jiffies( struct timespec *value );
void jiffies_to_timespec( unsigned long jiffies, struct timespec *value );
unsigned long timeval_to_jiffies( struct timeval *value );
void jiffies_to_timeval( unsigned long jiffies, struct timeval *value );
```

Доступ к 64-х разрядному счётчику тиков не так прост, как доступ к jiffies. В то время, как на 64-х разрядных архитектурах эти две переменные являются фактически одной, доступ к значению jiffies_64 для 32-х разрядных процессоров не атомарный. Это означает, что вы можете прочитать неправильное значение, если обе половинки переменной обновляются, пока вы читаете их. Ядро экспортирует специальную вспомогательную функцию, которая делает правильное блокирование:

```
#include <linux/jiffies.h>
#include <linux/types.h>
u64 get_jiffies_64( void );
```

Но, как правило, на большинстве процессорных архитектур для измерения временных интервалов могут быть использованы другие дополнительные механизмы, учитывая именно простоту реализации такой задачи, что уже обсуждалось ранее. Такие дополнительные источники информации о времени позволяют получить много выше (на несколько порядков!) разрешение, чем опираясь на системный таймер (а иначе зачем нужно было бы привлекать новые механизмы?). Простейшим из таких прецизионных датчиков времени может быть регистр-счётчик периодов тактирующей частоты процессора с возможностью его программного считывания.

Наиболее известным примером такого регистра-счётчика является TSC (timestamp counter), введённый в x86 процессоры, начиная с Pentium и с тех пор присутствует во всех последующих моделях процессоров этого семейства, включая платформу x86_64. Это 64-х разрядный регистр, который считает тактовые циклы процессора, он может быть прочитан и из пространства ядра и из пользовательского пространства. После подключения <asm/msr.h> (заголовок для x86, означающий machine-specific registers), можное использовать один из макросов:

- rdtsc(low32, high32) - атомарно читает 64-х разрядное значение в две 32-х разрядные переменные;

- rdtscl(low32) - (чтение младшей половины) читает младшую половину регистра в 32-х разрядную переменную, отбрасывая старшую половину;

- rdtscll(var64) - читает 64-х разрядное значение в переменную long long;

Пример использования таких макросов:

```
unsigned long ini, end;
rdtscl( ini ); /* здесь выполняется какое-то действие ... */ rdtscl( end );
printk( "time was: %li\n", end – ini );
```

Более обстоятельный пример измерения временных интервалов, используя счётчик процессорных тактов, можно найти в файле memtim.c архива mtest.tgz примеров, посвящённому тестированию распределителя памяти.

Большинство других платформ также предлагают аналогичную (но в чём-то отличающуюся в деталях) функциональную возможность. Заголовки ядра, поэтому, включают архитектурно-независимую функцию, скрывающую существующие различия реализации, и которую можно использовать вместо `rdtsc()`. Она называется `get_cycles()` (определена в `<asm/timex.h>`). Её прототип:

```
#include <linux/timex.h>
cycles_t get_cycles( void );
```

Эта функция определена для любой платформы, и она всегда **возвращает нулевое значение** на платформах, которые не имеют реализации регистра счётчика циклов. Тип `cycles_t` является соответствующим целочисленным типом без знака для хранения считанного значения.

Примечание: Нулевое значение, возвращаемое `get_cycles()` на платформах, не предоставляющих соответствующей реализации, делает возможным обеспечить переносимость между аппаратными платформами тщательно прописанного кода (там, где это есть, используется `get_cycles()`, а там, где этой возможности нет, тот же код реализуется, опираясь на последовательность системных тиков). Подобный подход реализован в нескольких различных местах системы Linux.

Для наблюдения эффектов измерений в службе времени рассмотрим тестовое приложение (архив `time.tgz`), которое для такого анализа может быть, с равным успехом, реализовано как процесс в пространстве пользователя — наблюдаемые эффекты будут те же :

clock.c :

```
#include "libdiag.h"

int main( int argc, char *argv[] ) {
   printf( "%016llX\n", rdtsc() );
   printf( "%016llX\n", rdtsc() );
   printf( "%016llX\n", rdtsc() );
   printf( "%d\n", proc_hz() );
   return EXIT_SUCCESS;
};
```

Для измерения значений размерностей времени, мы подготовим небольшую целевую статическую библиотеку (`libdiag.a`), которую станем применять не только в этом тесте, но и в других примерах **пользовательского пространства**. Вот основные библиотечные модули:

Это «ручная» (для наглядности, на инлайновой ассемблерной вставке), реализация счётчика процессорных циклов `rdtsc()` для пользовательского пространства, которая выполняет те же функции, что и вызовы в ядре `rdtsc()`, `rdtscl()`, `rdtscll()`, или `get_cycles()`:

rdtsc.c :

```
#include "libdiag.h"

uint64_t rdtsc( void ) {
   union sc {
      struct { uint32_t lo, hi; } r;
      uint64_t f;
   } sc;
   __asm__ __volatile__ ( "rdtsc" : "=a"( sc.r.lo ), "=d"( sc.r.hi ) );
   return sc.f;
}
```

Функция калибровка затрат процессорных тактов на само выполнение вызова `rdtsc()`, делается это как два непосредственно следующих друг за другом вызова `rdtsc()`, для снижения погрешностей это значение усредняется по циклу:

calibr.c :

```
#include "libdiag.h"

#define NUMB 10
```

```
unsigned calibr( int rep ) {
   uint64_t n, m, sum = 0;
   n = m = ( rep >= 0 ? NUMB : rep );
   while( n-- ) {
      uint64_t cf, cs;
      cf = rdtsc();
      cs = rdtsc();
      sum += cs - cf;
   }
   return (uint32_t)( sum / m );
}
```

Измерение частоты процессора (число процессорных тактов за секундный интервал) на основе предыдущих функций:

proc_hz.c :

```
#include "libdiag.h"

uint64_t proc_hz( void ) {
   time_t t1, t2;
   int64_t cf, cs;
   time( &t1 );
   while( t1 == time( &t2 ) ) cf  = rdtsc(); // начало след. секунды
   while( t2 == time( &t1 ) ) cs  = rdtsc(); // начало след. секунды
   return cs - cf - calibr( 1000 );          // с учётом времени вызова rdtsc()
}
```

Перевод потока в реал-тайм режим диспетчирования (в частности, на FIFO дисциплину), что бывает очень полезно сделать при любых измерениях временных интервалов:

set_rt.c :

```
void set_rt( void ) {
   struct sched_param sched_p;            // Information related to scheduling priority
   sched_getparam( getpid(), &sched_p );  // Change the scheduling policy to SCHED_FIFO
   sched_p.sched_priority = 50;           // RT Priority
   sched_setscheduler( getpid(), SCHED_FIFO, &sched_p );
}
```

Выполним этот тест на компьютерах x86 самой разной архитектуры (1, 2, 4 ядра), времени изготовления, производительности и версий ядра (собственно, только для такого сравнения и есть целесообразность готовить такой тест):

```
$ cat /proc/cpuinfo
processor       : 0
...
model name      : Celeron (Coppermine)
...
cpu MHz         : 534.569
...
$ ./clock
0000005E00E366B5
0000005E00E887B8
0000005E00EC3F15
534551251
$ cat /proc/cpuinfo
processor      : 0
...
model name     : Genuine Intel(R) CPU           T2300  @ 1.66GHz
...
cpu MHz        : 1000.000
```

```
...
processor      : 1
...
model name     : Genuine Intel(R) CPU           T2300  @ 1.66GHz
...
cpu MHz        : 1000.000
$ ./clock
00001D4AAD8FBD34
00001D4AAD920562
00001D4AAD923BD6
1662497985
$ cat /proc/cpuinfo
processor      : 0
...
model name     : Intel(R) Core(TM)2 Quad  CPU   Q8200  @ 2.33GHz
...
cpu MHz        : 1998.000
...
processor      : 1
...
model name     : Intel(R) Core(TM)2 Quad  CPU   Q8200  @ 2.33GHz
...
cpu MHz        : 2331.000
processor      : 2
...
model name     : Intel(R) Core(TM)2 Quad  CPU   Q8200  @ 2.33GHz
...
cpu MHz        : 1998.000
...
processor      : 3
...
model name     : Intel(R) Core(TM)2 Quad  CPU   Q8200  @ 2.33GHz
...
cpu MHz        : 1998.000
$ ./clock
000000000E98F3BB
000000000E9A75E8
000000000E9A925F
2320044881
```

Наблюдать подобную картину сравнительно на различном оборудовании — чрезвычайно полезное и любопытное занятие, но мы не станем сейчас останавливаться на деталях наблюдаемого, отметим только высокую точность совпадения независимых измерений, и то, что `rdtsc()` (или обратная величина частоты) измеряет, собственно, не частоту работы процессора (или какого-то отдельно взятого процессора в SMP системе), а **тактирующую частоту процессоров** в системе.

Наконец, мы соберём элементарный модуль ядра, который выведет нам значения тех основных констант и переменных службы времени, о которых говорилось:

tick.c :

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

static int __init hello_init( void ) {
   unsigned long j;
   u64 i;
   j = jiffies;
   printk( KERN_INFO "jiffies = %lX\n", j );
```

```
    printk( KERN_INFO "HZ value = %d\n", HZ );
    i = get_jiffies_64();
    printk( "jiffies 64-bit = %016llX\n", i );
    return -1;
}
module_init( hello_init );
```

Выполнение:

```
$ sudo /sbin/insmod ./tick.ko
insmod: error inserting './tick.ko': -1 Operation not permitted
$ dmesg | tail -n3
jiffies = 24AB3A0
HZ value = 1000
jiffies 64-bit = 00000001024AB3A0
```

Временные задержки

Обеспечение заданной паузы в выполнении программного кода — это вторая из обсуждавшихся ранее классов задач из области работы со временем. Она уже не так проста, как задача измерения времени и имеет больше разнообразных вариантов реализации, это связано ещё и с тем, что требуемая величина обеспечиваемой паузы может быть в очень широком диапазоне: от миллисекунд и ниже, для обеспечения корректной работы оборудования и протоколов (например, обнаружение конца фрейма в протоколе Modbus), и до десятков часов при реализации работы по расписанию — размах до 6-7 порядков величины.

Основное требование к функции временной задержки выражено требованием, сформулированным в стандарте POSIX, в его расширении реального времени POSIX 1003.b: заказанная временная задержка может быть при выполнении сколь угодно более продолжительной, но не может быть ни на какую величину и не при каких условиях — короче. Это условие не так легко выполнить!

Реализация временной задержка всегда относится к одному из двух родов: активное ожидание и пассивное ожидание (блокирование процесса). Активное ожидание осуществляется выполнением процессором «пустых» циклов на протяжении установленного интервала, пассивное — переводом потока выполнения в блокированное состояние. Существует предубеждение, что реализация через активное ожидание — это менее эффективная и даже менее профессиональная реализация, а пассивная, напротив, более эффективная. Это далеко не так: всё определяется конкретным контекстом использования. Например, любой переход в блокированное состояние — это очень трудоёмкая операция со стороны системы (переключения контекста, смена адресного пространства и множество других действий), реализация коротких пауз способом активного ожидания может просто оказаться эффективнее (прямую аналогию чему мы увидим при рассмотрении примитивов синхронизации: семафоры и спинблокировки). Кроме того, в ядре во многих случаях (в контексте прерывания и, в частности, в таймерных функциях) просто запрещено переходить в блокированное состояние.

Активные ожидания могут выполняться выполняются теми же механизмами (в принципе, всеми), что и измерение временных интервалов. Например, это может быть код, основанный на шкале системных тиков, подобный следующему:

```
unsigned long j1 = jiffies + delay * HZ; /* вычисляется значение тиков для окончания задержки */
while ( time_before( jiffies, j1 ) )
    cpu_relax();
```

где:

- `time_before()` - макрос, вычисляющий просто разницу 2-х значений с учётом возможных переполнений (уже рассмотренный ранее);

- `cpu_relax()` - макрос, говорящий, что процессор ничем не занят, и в гипер-триэдинговых системах могущий (в некоторой степени) занять процессор ещё чем-то;

В конечном счёте, и такая запись активной задержки будет вполне приемлемой:

```
while ( time_before( jiffies, j1 ) );
```

Для коротких задержек определены (как макросы `<linux/delay.h>`) несколько функций **активного**

ожидания со прототипами:

```
void ndelay( unsigned long nanoseconds );
void udelay( unsigned long microseconds );
void mdelay( unsigned long milliseconds );
```

Хотя они и определены как макросы:

```
#ifndef mdelay
#define mdelay(n) (                             \
{                                               \
        static int warned=0;                    \
        unsigned long __ms=(n);                 \
        WARN_ON(in_irq() && !(warned++));       \
        while (__ms--) udelay(1000);            \
})
#endif
#ifndef ndelay
#define ndelay(x)       udelay(((x)+999)/1000)
#endif
```

Но в некоторых случаях интерес вызывают именно **пассивные** ожидания (переводящие поток в блокированное состояние), особенно при реализации достаточно продолжительных интервалов. Первое решение состоит просто в элементарном отказе от занимаемого процессора до наступления момента завершения ожидания:

```
#include <linux/sched.h>
while( time_before( jiffies, j1 ) ) {
   schedule();
}
```

Пассивное ожидание можно получить функцией:

```
#include <linux/sched.h>
signed long schedule_timeout( signed long timeout );
```

- где `timeout` - число тиков для задержки. Возвращается значение 0, если функция вернулась перед истечением данного времени ожидания (в ответ на сигнал). Функция `schedule_timeout()` **требует**, чтоб прежде вызова было установлено текущее состояние процесса, допускающее прерывание сигналом, поэтому типичный вызов выглядит следующим образом:

```
set_current_state( TASK_INTERRUPTIBLE );
schedule_timeout( delay );
```

Определено несколько функций ожидания, не использующие активное ожидание (`<linux/delay.h>`):

```
void msleep( unsigned int milliseconds );
unsigned long msleep_interruptible( unsigned int milliseconds );
void ssleep( unsigned int seconds );
```

Первые две функции помещают вызывающий процесс в пассивное состояние на заданное число **милли**секунд. Вызов `msleep()` является непрерываемым: можно быть уверенным, что процесс остановлен по крайней мере на заданное число миллисекунд. Если драйвер помещён в очередь ожидания и мы хотим использовать возможность принудительного пробуждения (сигналом) для прерывания пассивности, используем `msleep_interruptible()`. Возвращаемое значение `msleep_interruptible()` при естественном возврате 0, однако если этот процесс активизирован сигналом раньше, возвращаемое значение является числом миллисекунд, оставшихся от первоначально запрошенного периода ожидания. Вызов `ssleep()` помещает процесс в непрерываемое ожидание на заданное число секунд.

Рассмотрим разницу между активными и пассивными задержками, причём различие это абсолютно одинаково в ядре и пользовательском процессе, поэтому рассмотрение делается на выполнении процесса пространства пользователя (архив `time.tgz`):

pdelay.c :

```
#include "libdiag.h"
```

```
int main( int argc, char *argv[] ) {
   long dl_nsec[] = { 10000, 100000, 200000, 300000, 500000, 1000000, 1500000, 2000000, 5000000 };
   int c, i, j, bSync = 0, bActive = 0, cycles = 1000,
       rep = sizeof( dl_nsec ) / sizeof( dl_nsec[ 0 ] );
   while( ( c = getopt( argc, argv, "astn:r:" ) ) != EOF )
      switch( c ) {
         case 'a': bActive = 1; break;
         case 's': bSync = 1; break;
         case 't': set_rt(); break;
         case 'n': cycles = atoi( optarg ); break;
         case 'r': if( atoi( optarg ) > 0 && atoi( optarg ) < rep ) rep = atoi( optarg ); break;
         default:
            printf( "usage: %s [-a] [-s] [-n cycles] [-r repeats]\n", argv[ 0 ] );
            return EXIT_SUCCESS;
   }
   char *title[] = { "passive", "active" };
   printf( "%d cycles %s delay [millisec. == tick !] :\n", cycles,
           ( bActive == 0 ? title[ 0 ] : title[ 1 ] ) );
   unsigned long prs = proc_hz();
   printf( "processor speed: %d hz\n", prs );
   long cali = calibr( 1000 );
   for( j = 0; j < rep; j++ ) {
      const struct timespec sreq = { 0, dl_nsec[ j ] };    // наносекунды для timespec
      long long rb, ra, ri = 0;
      if( bSync != 0 ) nanosleep( &sreq, NULL );
      if( bActive == 0 ) {
         for( i = 0; i < cycles; i++ ) {
            rb = rdtsc();
            nanosleep( &sreq, NULL );
            ra = rdtsc();
            ri += ( ra - rb ) - cali;
         }
      }
      else {
         long long wpr = (long long) ( ( (double) dl_nsec[ j ] ) / 1e9 * prs );
         for( i = 0; i < cycles; i++ ) {
            rb = rdtsc() + cali;
            while( ( ra = rdtsc() ) - rb < wpr  ) {}
            ri +=  ra - rb;
         }
      }
      double del = ( (double)ri ) / ( (double)prs );
      printf( "set %5.3f => was %5.3f\n",
              ( ( (double)dl_nsec[ j ] ) / 1e9 ) * 1e3, del * 1e3 / cycles );
   }
   return EXIT_SUCCESS;
};
```

Активные задержки:

```
$ sudo nice -n-19 ./pdelay -n 1000 -a
1000 cycles active delay [millisec. == tick !] :
processor speed: 1662485585 hz
set 0.010 => was 0.010
set 0.100 => was 0.100
set 0.200 => was 0.200
set 0.300 => was 0.300
set 0.500 => was 0.500
set 1.000 => was 1.000
```

```
set 1.500 => was 1.500
set 2.000 => was 2.000
set 5.000 => was 5.000
```

Пассивные задержки (на разном ядре могут давать самый разнообразный **характер** результатов), вот картина наиболее характерная на относительно старых архитектурах и ядрах (и именно это классическая картина диспетчирования по системному таймеру, без привлечения дополнительных аппаратных уточняющих источников информации высокого разрешения):

```
$ uname -r
2.6.18-92.el5
$ sudo nice -n-19 ./pdelay -n 1000
1000 cycles passive delay [millisec. == tick !] :
processor speed: 534544852 hz
set 0.010 => was 1.996
set 0.100 => was 1.999
set 0.200 => was 1.997
set 0.300 => was 1.998
set 0.500 => was 1.999
set 1.000 => was 2.718
set 1.500 => was 2.998
set 2.000 => was 3.889
set 5.000 => was 6.981
```

Хотя цифры при малых задержках и могут показаться неожиданными, именно они объяснимы, и совпадут с тем, как это будет выглядеть в других POSIX операционных системах. Увеличение задержки на два системных тика (3 миллисекунды при заказе 1-й миллисекунды) нисколько не противоречит упоминавшемуся требованию стандарта POSIX 1003.b (и даже сделано в его обеспечение) и объясняется следующим:

- период первого тика после вызова не может «идти в зачёт» выдержки времени, потому как вызов `nanosleep()` происходит асинхронно относительно шкалы системных тиков, и мог бы прийтись ровно перед очередным системным тиком, и тогда выдержка в один тик была бы «зачтена» потенциально нулевому интервалу;
- следующий, второй тик пропускается именно из-за того, что величина периода системного тика чуть меньше миллисекунды (0. 999847мс, как это обсуждалось выше), и вот этот остаток «чуть» и приводит к ожиданию ещё одного очередного, не исчерпанного тика.

Как раз более необъяснимыми (хотя и более ожидаемыми по житейской логике) будут цифры на новых архитектурах и ядрах:

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
$ sudo nice -n-19 ./pdelay -n 1000
1000 cycles passive delay [millisec. == tick !] :
processor speed: 1662485496 hz
set 0.010 => was 0.090
set 0.100 => was 0.182
set 0.200 => was 0.272
set 0.300 => was 0.370
set 0.500 => was 0.571
set 1.000 => was 1.075
set 1.500 => was 1.575
set 2.000 => was 2.074
set 5.000 => was 5.079
```

Здесь определённо для получения такой разрешающей способности использованы другие дополнительные датчики временных шкал, отличных от системного таймера дискретностью в одну миллисекунду.

В любом случае, из результатов этих примеров мы должны сделать несколько заключений:

- при указании аргумента функции пассивной задержки порядка величины 3-5 системных тиков или менее, не стоит ожидать каких-то адекватных указанной величине интервалов ожидания, реально это может быть величина большая в разы...
- рассчёт на то, что активная задержка выполнится с большей точностью (и может быть задана с меньшей дискретностью) отчасти оправдан, но также на это не следует твёрдо рассчитывать: выполняющий активные циклы поток может быть вытеснен в блокированное состояние, и интервал ожидания будем суммироваться с временем блокировки, это ещё хуже (в смысле погрешности), чем в случае пассивных задержек;
- за счёт возможности вытеснения в блокированное состояние, временные паузы могут (с невысокой вероятностью) оказаться больше указанной величины в разы, и даже на несколько порядков, такую возможность нужно иметь в виду, и это **нормальное** поведение в смысле толкования требования стандарта POSIX реального времени.

Таймеры ядра

Последним классом рассматриваемых задач относительно времени будут таймерные функции. Понятие таймера существенно шире и сложнее в реализации, чем просто выжидание некоторого интервала времени, как мы рассматривали это ранее. Таймер (экземпляров которых может одновременно существовать достаточно много) должен **асинхронно** возбудить некоторое предписанное ему действие в указанный момент времени в будущем.

Ядро предоставляет драйверам API таймера: ряд функций для декларации, регистрации и удаления таймеров ядра:

```
#include <linux/timer.h>
struct timer_list {
   struct list_head entry;
   unsigned long expires;
   void (*function)( unsigned long );
   unsigned long data;
...
};

void init_timer( struct timer_list *timer );
struct timer_list TIMER_INITIALIZER( _function, _expires, _data );
void add_timer( struct timer_list *timer );
void mod_timer( struct timer_list *timer, unsigned long expires );
int del_timer( struct timer_list *timer );
```

- `expires` - значение `jiffies`, наступления которого таймер ожидает для срабатывания (**абсолютное** время) ;
- при срабатывании функция `function()` вызывается с `data` в качестве аргумента;
- чаще всего `data` — это преобразованный указатель на структуру;

Функция таймера в ядре выполняется в **контексте прерывания** (Не в контексте процесса! А конкретнее: в контексте обработчика прерывания системного таймера.), что накладывает на неё дополнительные ограничения:

- Не разрешён доступ к пользовательскому пространству. Из-за отсутствия контекста процесса, нет пути к пользовательскому пространству, связанному с любым определённым процессом.
- Указатель `current` не имеет смысла и не может быть использован, так как соответствующий код не имеет связи с процессом, который был прерван.
- Не может быть выполнен переход в блокированное состояние и переключение контекста. Код в контексте прерывания не может вызвать `schedule()` или какую-то из форм `wait_event(),` и не может вызвать любые другие функции, которые могли бы перевести его в пассивное состояние, семафоры и подобные примитивы синхронизации также не должны быть использованы, поскольку они могут переключать выполнение в пассивное состояние.

Код ядра может понять, работает ли он в контексте прерывания, используя макрос: `in_interrupt()`.

Примечание: утверждается, что а). в системе 512 списков таймеров, каждый из которых с фиксированным `expires`, б). они, в свою очередь, разделены на 5 групп по диапазонам `expires`, в). с течением времени (по мере приближения `expires`) списки перемещаются из группы в группу... Но это уже реализационные нюансы.

Таймеры высокого разрешения

Таймеры высокого разрешения появляются с ядра 2.6.16, структуры представления времени для них определяются в файлах `<linux/ktime.h>`. Поддержка осуществляется только в тех архитектурах, где есть поддержка аппаратных таймеров высокого разрешения. Определяется новый временной тип данных `ktime_t` — временной интервал в наносекундном выражении, представление его сильно разнится от архитектуры. Здесь же определяются множество функций установки значений и преобразований представления времени (многие из них определены как макросы, но здесь записаны как прототипы):

```
ktime_t ktime_set(const long secs, const unsigned long nsecs);
ktime_t timeval_to_ktime( struct timeval tv );
struct timeval  ktime_to_timeval( ktime_t kt );
ktime_t timespec_to_ktime( struct timespec ts );
struct timespec ktime_to_timespec( ktime_t kt );
```

Сами операции с таймерами высокого разрешения определяются в `<linux/hrtimer.h>`, это уже очень напоминает модель таймеров реального времени, вводимую для пространства пользователя стандартом POSIX 1003b:

```
struct hrtimer {
...
   ktime_t   _expires;
   enum hrtimer_restart (*function)(struct hrtimer *);
...
}
```

- единственным определяемым пользователем полем этой структуры является функция реакции `function`, здесь обращает на себя внимание прототип этой функции, которая возвращает:

```
enum hrtimer_restart {
   HRTIMER_NORESTART,
   HRTIMER_RESTART,
};

void hrtimer_init( struct hrtimer *timer, clockid_t which_clock, enum hrtimer_mode mode );
int hrtimer_start( struct hrtimer *timer, ktime_t tim, const enum hrtimer_mode mode );
extern int hrtimer_cancel( struct hrtimer *timer );
...
enum hrtimer_mode {
   HRTIMER_MODE_ABS = 0x0,   /* Time value is absolute */
   HRTIMER_MODE_REL = 0x1,   /* Time value is relative to now */
...
};
```

Параметр `which_clock` типа `clockid_t`, это вещь из области стандартов POSIX, то, что называется стандартом временной базис (тип задатчика времени): какую шкалу времени использовать, из общего числа определённых в `<linux/time.h>` (часть из них из POSIX, а другие расширяют число определений):

```
// The IDs of the various system clocks (for POSIX.1b interval timers):
#define CLOCK_REALTIME           0
#define CLOCK_MONOTONIC          1
#define CLOCK_PROCESS_CPUTIME_ID 2
#define CLOCK_THREAD_CPUTIME_ID  3
#define CLOCK_MONOTONIC_RAW      4
#define CLOCK_REALTIME_COARSE    5
```

```
#define CLOCK_MONOTONIC_COARSE    6
```

Примечание: Относительно временных базисов в Linux известно следующее:

- `CLOCK_REALTIME` — системные часы, со всеми их плюсами и минусами. Могут быть переведены вперёд или назад, в этой шкале могут попадаться «вставные секунды», предназначенные для корректировки неточностей представления периода системного тика. Это наиболее используемая в таймерах шкала времени.
- `CLOCK_MONOTONIC` — подобно `CLOCK_REALTIME`, но отличается тем, что, представляет собой постоянно увеличивающийся счётчик, в связи с чем, естественно, не могут быть изменены при переводе времени. Обычно это счётчик от загрузки системы.
- `CLOCK_PROCESS_CPUTIME_ID` — возвращает время затрачиваемое процессором относительно пользовательского процесса, время затраченное процессором на работу только с данным приложением в независимости от других задач системы. Естественно, что это базис для пользовательского адресного пространства.
- `CLOCK_THREAD_CPUTIME_ID` — похоже на `CLOCK_PROCESS_CPUTIME_ID`, но только отсчитывается время, затрачиваемое на один текущий поток.
- `CLOCK_MONOTONIC_RAW` — то же что и `CLOCK_MONOTONIC`, но в отличии от первого не подвержен изменению через сетевой протокол точного времени NTP.

Последние два базиса `CLOCK_REALTIME_COARSE` и `CLOCK_MONOTONIC_COARSE` добавлены недавно (2009 год), авторами утверждается (http://lwn.net/Articles/347811/), что они могут обеспечить гранулярность шкалы мельче, чем предыдущие базисы. Работу с различными базисами времени обеспечивают в пространстве пользователя малоизвестные API вида `clock_*()` (`clock_gettext(), clock_nanosleep(), clock_settime(), ...`), в частности, разрешение каждого из базисов можно получить вызовом:

```
long sys_clock_getres( clockid_t which_clock, struct timespec *tp );
```

Для наших примеров временной базис таймеров вполне может быть, например, `CLOCK_REALTIME` или `CLOCK_MONOTONIC`. Пример использования таймеров высокого разрешения (архив `time.tgz`) в периодическом режиме может быть показан таким модулем (код только для демонстрации техники написания в этом API, но не для рассмотрения возможностей высокого разрешения):

htick.c :

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/time.h>
#include <linux/ktime.h>
#include <linux/hrtimer.h>

static ktime_t tout;
static struct kt_data {
   struct hrtimer timer;
   ktime_t        period;
   int            numb;
} *data;

#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,19)
static int ktfun( struct hrtimer *var ) {
#else
static enum hrtimer_restart ktfun( struct hrtimer *var ) {
#endif
   ktime_t now = var->base->get_time();      // текущее время в типе ktime_t
   printk( KERN_INFO "timer run #%d at jiffies=%ld\n", data->numb, jiffies );
   hrtimer_forward( var, now, tout );
   return data->numb-- > 0 ? HRTIMER_RESTART : HRTIMER_NORESTART;
}

int __init hr_init( void ) {
   enum hrtimer_mode mode;
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,19)
```

```
   mode = HRTIMER_REL;
#else
   mode = HRTIMER_MODE_REL;
#endif
   tout = ktime_set( 1, 0 );       /* 1 sec. + 0 nsec. */
   data = kmalloc( sizeof(*data), GFP_KERNEL );
   data->period = tout;
   hrtimer_init( &data->timer, CLOCK_REALTIME, mode );
   data->timer.function = ktfun;
   data->numb = 3;
   printk( KERN_INFO "timer start at jiffies=%ld\n", jiffies );...
   hrtimer_start( &data->timer, data->period, mode );
   return 0;
}

void hr_cleanup( void ) {
   hrtimer_cancel( &data->timer );
   kfree( data );
   return;
}

module_init( hr_init );
module_exit( hr_cleanup );
MODULE_LICENSE( "GPL" );
```

Результат:

```
$ sudo insmod ./htick.ko
$ dmesg | tail -n5
timer start  at jiffies=10889067
timer run #3 at jiffies=10890067
timer run #2 at jiffies=10891067
timer run #1 at jiffies=10892067
timer run #0 at jiffies=10893067
$ sudo rmmod htick
```

Абсолютное время

Всё рассмотрение выше касалось измерения относительных временных интервалов (даже если эта относительность отсчитывается от достаточно отдалённой во времени точки загрузки системы, как в случае с `jiffies`). Реальное хронологическое время (абсолютное время) нужно ядру исключительно редко (если вообще нужно) - его вычисление и представление лучше оставить коду пространства пользователя. Тем не менее, в ядре абсолютное UTC время (время эпохи UNIX - отсчитываемое от 1 января 1970г.) хранится как:

```
struct timespec xtime;
```

В UNIX традиционно существует две структуры **точного** представления времени (как в ядре, так и в пространстве пользователя), полностью идентичные по своей функциональности:

```
#include <linux/time.h>
struct timespec {
   time_t tv_sec; /* секунды */
   long tv_nsec;  /* нано секунды */
}
...
struct timeval {
   time_t       tv_sec;  /* секунды */
   suseconds_t  tv_usec; /* микро секунды */
};
...
#define NSEC_PER_USEC   1000L
```

```
#define USEC_PER_SEC    1000000L
#define NSEC_PER_SEC    1000000000L
```

В виду не атомарности `xtime`, непосредственно использовать его нельзя, но есть некоторый набор API ядра для преобразования с хронологического времени а одну из форм и обратно:

– превращение хронологического времени в значение единиц `jiffies` :

```
#include <linux/time.h>
unsigned long mktime( unsigned int year, unsigned int mon, unsigned int day,
                      unsigned int hour, unsigned int min, unsigned int sec );
```

– текущее время с разрешением до тика:

```
#include <linux/time.h>
struct timespec current_kernel_time( void );
```

– текущее время с разрешением меньше тика (при наличии аппаратной поддержке для этого на используемой платформе, и очень сильно зависит от используемой платформы):

```
#include <linux/time.h>
void do_gettimeofday( struct timeval *tv );
```

Часы реального времени (RTC)

Часы реального времени — это сугубо аппаратное расширение, которое принципиально зависит от аппаратной платформы, на которой используется Linux. Это ещё одно расширение службы системных часов, на некоторых архитектурах его может и не быть. Используя такое расширение можно создать ещё одну независимую шкалу отсчётов времени, с которой можно связать измерения, или даже асинхронную активацию действий.

Убедиться наличии такого расширения на используемой аппаратной платформе можно по присутствию интерфейса к таймеру часов реального времени в пространстве пользователя. Такой интерфейс предоставляется (о чём чуть позже) через функции `ioctl()` драйвера присутствующего в системе устройства `/dev/rtc` :

```
$ ls -l /dev/rtc*
lrwxrwxrwx 1 root root      4 Апр 25 09:52 /dev/rtc -> rtc0
crw-rw---- 1 root root 254, 0 Апр 25 09:52 /dev/rtc0
```

В архитектуре Intel x86 устройство этого драйвера называется Real Time Clock (RTC). RTC предоставляет функцию для работы со 114-битовым значением в NVRAM. На входе этого устройства установлен осциллятор с частотой 32768 КГц, подсоединенный к энергонезависимой батарее. Некоторые дискретные модели RTC имеют встроенные осциллятор и батарею, тогда как другие RTC встраиваются прямо в контроллер периферийной шины (например, южный мост) чипсета процессора. RTC возвращает не только время суток, но, помимо прочего, является и программируемым таймером, имеющим возможность посылать системные прерывания (IRQ 8). Частота прерываний варьируется от 2 до 8192 Гц. Также RTC может посылать прерывания ежедневно, наподобие будильника. Все определения находим в `<linux/rtc.h>` :

```
struct rtc_time {
   int tm_sec;
   int tm_min;
   int tm_hour;
   int tm_mday;
   int tm_mon;
   int tm_year;
   int tm_wday;
   int tm_yday;
   int tm_isdst;
};
```

Только некоторые важные коды команд `ioctl()` :

```
#define RTC_AIE_ON    _IO( 'p', 0x01 ) /* Включение прерывания alarm  */
```

```
#define RTC_AIE__OFF  _IO( 'p', 0x02 ) /* ... отключение */
...
#define RTC_PIE_ON    _IO( 'p', 0x05)   /* Включение периодического прерывания */
#define RTC_PIE_OFF   _IO( 'p', 0x06)   /* ... отключение */
...
#define RTC_ALM_SET   _IOW( 'p', 0x07, struct rtc_time) /* Установка времени time  */
#define RTC_ALM_READ  _IOR( 'p', 0x08, struct rtc_time) /* Чтение времени alarm */
#define RTC_RD_TIME   _IOR( 'p', 0x09, struct rtc_time) /* Чтение времени RTC */
#define RTC_SET_TIME  _IOW( 'p', 0x0a, struct rtc_time) /* Установка времени RTC */
#define RTC_IRQP_READ _IOR( 'p', 0x0b, unsigned long)<> /* Чтение частоты IRQ */
#define RTC_IRQP_SET  _IOW( 'p', 0x0c, unsigned long)<> /* Установка частоты IRQ */
```

Пример использования RTC из пользовательской программы для считывания абсолютного значения времени (архив `time.tgz`):

rtcr.c :

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <string.h>
#include <linux/rtc.h>

int main( void ) {
   int fd, retval = 0;
   struct rtc_time tm;
   memset( &tm, 0, sizeof( struct rtc_time ) );
   fd = open( "/dev/rtc", O_RDONLY );
   if( fd < 0 ) printf( "error: %m\n" );
   retval = ioctl( fd, RTC_RD_TIME, &tm );  // Чтение времени RTC
   if( retval ) printf( "error: %m\n" );
   printf( "current time: %02d:%02d:%02d\n", tm.tm_hour, tm.tm_min, tm.tm_sec );
   close( fd );
   return 0;
}
```

```
$ ./rtcr
current time: 12:58:13
$ date
Пнд Апр 25 12:58:16 UTC 2011
```

Ещё одним примером (по мотивам [5], но сильно переделанным) покажем, как часы RTC могут быть использованы как независимый источник времени в программе, генерирующей периодические прерывания с высокой (значительно выше системного таймера) частотой следования:

rtprd.c :

```
#include <stdio.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <fcntl.h>
#include <pthread.h>
#include <linux/mman.h>
#include "libdiag.h"

unsigned long ts0, worst = 0, mean = 0;      // для загрузки тиков
unsigned long cali;
unsigned long long sum = 0;                  // для накопления суммы
int cycle = 0;
```

```
void do_work( int n ) {
   unsigned long now = rdtsc();
   now = now - ts0 - cali;
   sum += now;
   if( now > worst ) {
      worst = now;                          // Update the worst case latency
      cycle = n;
   }
   return;
}

int main( int argc, char *argv[] ) {
   int fd, opt, i = 0, rep = 1000, nice = 0, freq = 8192; // freq - RTC частота - hz
   /* Set the periodic interrupt frequency to 8192Hz
      This should give an interrupt rate of 122uS */
   while ( ( opt = getopt( argc, argv, "f:r:n") ) != -1 ) {
      switch( opt ) {
         case 'f' : if( atoi( optarg ) > 0 ) freq = atoi( optarg ); break;
         case 'r' : if( atoi( optarg ) > 0 ) rep = atoi( optarg ); break;
         case 'n' : nice = 1; break;
         default :
            printf( "usage: %s [-f 2**n] [-r #] [-n]\n", argv[ 0 ] );
            exit( EXIT_FAILURE );
      }
   };
   printf( "interrupt period set %.2f us\n", 1000000. / freq );
   if( 0 == nice ) {
      struct sched_param sched_p;            // Information related to scheduling priority
      sched_getparam( getpid(), &sched_p );  // Change the scheduling policy to SCHED_FIFO
      sched_p.sched_priority = 50;           // RT Priority
      sched_setscheduler( getpid(), SCHED_FIFO, &sched_p );
   }
   mlockall( MCL_CURRENT );                 // Avoid paging and related indeterminism
   cali = calibr( 10000 );
   fd = open( "/dev/rtc", O_RDONLY );       // Open the RTC
   unsigned long long prochz = proc_hz();
   ioctl( fd, RTC_IRQP_SET, freq );
   ioctl( fd, RTC_PIE_ON, 0 );              // разрешить периодические прерывания
   while ( i++ < rep ) {
      unsigned long data;
      ts0 = rdtsc();
      // блокировать до следующего периодического прерывния
      read( fd, &data, sizeof(unsigned long) );
      // выполнять периодическую работу ... измерять латентность
      do_work( i );
   }
   ioctl( fd, RTC_PIE_OFF, 0 );            // запретить периодические прерывания
   printf( "worst latency was %.2f us (on cycle %d)\n", tick2us( prochz, worst ), cycle );
   printf( "mean latency was %.2f us\n", tick2us( prochz, sum / rep ) );
   exit( EXIT_SUCCESS );
}
```

В примере прерывания RTC прерывают блокирующую операцию `read()` гораздо чаще периода системного тика. Очень показательно в этом примере запуск без перевода процесса (что делается по умолчанию) в реал-тайм диспетчирование (ключ `-n`), когда дисперсия временной латентности возрастает сразу на 2 порядка (это эффекты вытесняющего диспетчирования, которые должны **всегда** приниматься во внимание при планировании измерений временных интервалов):

```
$ sudo ./rtprd
interrupt period set 122.07 us
```

```
worst latency was 266.27 us (on cycle 2)
mean latency was 121.93 us
$ sudo ./rtprd -f16384
interrupt period set 61.04 us
worst latency was 133.27 us (on cycle 2)
mean latency was 60.79 us
$ sudo ./rtprd -f16384 -n
interrupt period set 61.04 us
worst latency was 8717.90 us (on cycle 491)
mean latency was 79.45 us
```

Показанный выше код пространства пользователя в заметной мере проясняет то, как и на каких интервалах могут использоваться часы реального времени. То же, каким образом время RTC считывается в ядре, не скрывается никакими обёртками, и радикально зависит от использованного оборудования RTC. Для наиболее используемого чипа Motorola 146818 (который в таком наименовании давно уже не производится, и заменяется дженериками), можно упоминание соответствующих макросов (и другую информацию для справки) найти в `<asm-generic/rtc.h>`:

```
spin_lock_irq( &rtc_lock ) ;
rtc_tm->tm_sec = CMOS_READ( RTC_SECONDS ) ;
rtc_tm->tm_min = CMOS_READ( RTC_MINUTES );
rtc_tm->tm_hour = CMOS_READ( RTC_HOURS );
...
spin_unlock_irq( &rtc_lock );
```

А все нужные для понимания происходящего определения находим в `<linux/mc146818rtc.h>`:

```
#define RTC_SECONDS 0
#define RTC_SECONDS_ALARM 1
#define RTC_MINUTES2
...
#define CMOS_READ(addr) ({ \
   outb_p( (addr), RTC_PORT(0) ); \
   inb_p( RTC_PORT(1) ); \
})
#define RTC_PORT(x)     (0x70 + (x))
#define RTC_IRQ 8
```

- в порт 0x70 записывается номер требуемого параметра, а по порту 0x71 считывается/записывается требуемое значение — так традиционно организуется обмен с данными памяти CMOS.

Время и диспетчеризация в ядре

Диспетчеризация (планироване) процессов в Linux выполняется строго **по системному таймеру**, на основании **динамически** пересчитываемых приоритетов. Приоритетов 140 (`MAX_PRIO`): 100 реального времени + 40 приоритетов «обычной» диспетчеризации, называемые ещё приоритетами `nice` (параметр `nice` в диапазоне от -20 до +19 — максимальный приоритет -20). Процессы, диспетчируемые по дисциплинам реального времени в Linux, это в достаточной мере экзотика, и они могут быть запущены только специальным образом (используя API диспетчеризации). Каждому процессу с сформированным приоритетом `nice` на каждом периоде диспетчирования, в зависимости от этого значения приоритета процесса, назначается **период активности** (timeslice) — 10-200 системных тиков, который динамически в ходе выполнения этого процесса может быть ещё расширен в пределах 5-800, в зависимости от характера интерактивности процесса (процессам, активно загружающим процессор, timeslice задаётся ниже, а активно взаимодействующим с пользователем — **повышается**). На этом построена схема диспетчеризации процессов в Linux сложности O(1) - не зависящая по производительности от числа подлежащих планированию процессов, которой очень гордятся разработчики ядра Linux (возможно, вполне оправдано).

Примечание: Новая система диспетчеризации O(1) построена на основе 2-х очередей: очередь **ожидающих** выполнения процессов, и очередь **отработавших** свой квант процессов. Из первой из них выбирается поочерёдно следующий процесс на выполнение, и после выработки им своего кванта, он сбрасывается во вторую. Когда очередь ожидающих опустошается,

очереди просто меняются местами: очередь отработавших становится новой очередью ожидающих, а пустая очередь ожидающих — становится очередью отработавших. Но всё это происходит так только **при отсутствии** процессов с установленной реалтайм диспетчеризацией (RR или FIFO), с ненулевым приоритетом реального времени. До тех пор, пока в системе будет находиться хотя бы один реалтайм процесс в состоянии **готовности** к выполнению (активный), ни один процесс нормального приоритета не будет выбираться на исполнение (на **данном процессоре**!).

Всё, что касается диспетчеризации процессов в ядре, весьма интересно и сложно. Но всё это уже далеко выходит за пределы нашего рассмотрения... Нам же здесь важно зафиксировать, что все диспетчеризуемые изменения состояний системы происходят строго в привязке к шкале **системных тиков**.

Параллелизм и синхронизация

«Две передние, старшие, ноги вели животное в одну сторону – за большой головой, а две задние, младшие, ноги – в противоположную, за снабжённым головой женским хвостом.»
Милорад Павич «Смерть святого Савы, или невидимая сторона Луны»

Ядро Linux является **вытесняющим** (преемптивным, preemptive): код ядра в состоянии вытеснить другие выполняющиеся задания, даже если они работают в режиме ядра. Среди разнообразных операционных систем весьма немногие имеют вытесняющее ядро, это некоторые коммерческие реализации UNIX, например, Solaris, AIX®. Но вытеснение появляется и имеет смысл только тогда, когда возникают возможности параллелизма, когда в ядре могут существовать **потоки** выполнения.

Механизм потоков ядра (kernel thread - появляющийся с ядра 2.5) предоставляет средство параллельного выполнения задач в ядре. Общей особенностью и механизмов потоков ядра, и примитивов для их синхронизации, является то, что они в принципиальной основе своей единообразны: что для пользовательского пространства, что для ядра — различаются тонкие нюансы и функции доступного API их использования. Поэтому, рассмотрение (и тестирование на примерах) работы механизмов синхронизации можно с равной степенью общности (или параллельно) проводить как в пространстве ядра, там и в пространстве пользователя, например, так как это сделано в [9].

Нужно отчётливо разделить два класса параллелизма (а особенно требуемых для их обеспечения синхронизаций), **природа** которых совершенно **различного** происхождения:

1. Логический параллелизм (или квази-параллелизм), обусловленный удобством разделения разнородных сервисов ядра, но реализующие потоки которых вытесняют друг друга, создавая только иллюзию параллельности. При этом синхронизация осуществляется исключительно классическими блокирующими механизмами, когда поток ожидает недоступных ему ресурсов переводясь в блокированное состояние.

2. Физический параллелизм (или реальный параллелизм), возникший только с широким распространением SMP (в виде многоядерности или/и гипертриэдинга), когда разные задачи ядра выполняются одновременно на различных процессорах. В этом случае широко начинают использоваться (наряду с классическими) активные примитивы синхронизации (спин-блокировки), когда один из процессоров просто ожидает требуемых ресурсов **выполняя пустые циклы ожидания**. Этот второй класс (активно развиваемый примерно с 2003-2005 г.г.) много крат усложняет картину происходящего (существуя одновременно с предыдущим классом), и доставляет большую головную боль разработчику. Но с ним придётся считаться, прогнозируя достаточно динамичное развитие тех направлений, что уже сегодня называется массивно-параллельными системами (примером чего может быть модель программирования CUDA компании NVIDIA), когда от систем с 2-4-8 процессоров SMP происходит переход к сотням и тысячам процессоров.

Механизм потоков ядра начал всё шире и шире использоваться от версии к версии ядер 2.6.x, на него даже было перенесено (переписано) ряд **традиционных** и давно существующих демонов Linux пользовательского уровня (в протоколе команд далее специально сохранены компоненты, относящиеся к

сетевой файловой подсистеме nfsd — одной из самых давних подсистем UNIX). В формате вывода команды ps потоки ядра выделяются квадратными скобками:

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 09:52 ?        00:00:01 /sbin/init
root         2     0  0 09:52 ?        00:00:00 [kthreadd]
root         3     2  0 09:52 ?        00:00:00 [migration/0]
root         4     2  0 09:52 ?        00:00:00 [ksoftirqd/0]
root         5     2  0 09:52 ?        00:00:00 [watchdog/0]
root         6     2  0 09:52 ?        00:00:00 [migration/1]
root         7     2  0 09:52 ?        00:00:00 [ksoftirqd/1]
root         8     2  0 09:52 ?        00:00:00 [watchdog/1]
root         9     2  0 09:52 ?        00:00:00 [events/0]
root        10     2  0 09:52 ?        00:00:00 [events/1]
...
root       438     2  0 09:52 ?        00:00:00 [kjournald]
root       458     2  0 09:52 ?        00:00:00 [kauditd]
...
root       518     1  0 09:52 ?        00:00:00 /sbin/udevd -d
root       858     2  0 09:53 ?        00:00:00 [tifm]
root       870     2  0 09:53 ?        00:00:00 [kmmcd]
...
root      1224     1  0 09:53 ?        00:00:00 /sbin/rsyslogd -c 4
root      1245     2  0 09:53 ?        00:00:00 [kondemand/0]
root      1246     2  0 09:53 ?        00:00:00 [kondemand/1]
rpc       1268     1  0 09:53 ?        00:00:00 rpcbind
...
rpcuser   1323     1  0 09:53 ?        00:00:00 rpc.statd
...
root      1353     2  0 09:53 ?        00:00:00 [rpciod/0]
root      1354     2  0 09:53 ?        00:00:00 [rpciod/1]
root      1361     1  0 09:53 ?        00:00:00 rpc.idmapd
...
root      1720     1  0 09:53 ?        00:00:00 rpc.rquotad
root      1723     2  0 09:53 ?        00:00:00 [lockd]
root      1724     2  0 09:53 ?        00:00:00 [nfsd4]
root      1725     2  0 09:53 ?        00:00:00 [nfsd]
root      1726     2  0 09:53 ?        00:00:00 [nfsd]
root      1727     2  0 09:53 ?        00:00:00 [nfsd]
root      1728     2  0 09:53 ?        00:00:00 [nfsd]
root      1729     2  0 09:53 ?        00:00:00 [nfsd]
root      1730     2  0 09:53 ?        00:00:00 [nfsd]
root      1731     2  0 09:53 ?        00:00:00 [nfsd]
root      1732     2  0 09:53 ?        00:00:00 [nfsd]
root      1735     1  0 09:53 ?        00:00:00 rpc.mountd
...
```

Для всех показанных в выводе потоков ядра родителем (PPID) является демон kthreadd (PID=2), который, как и процесс init не имеет родителя (PPID=0), и который запускается непосредственно при старте ядра. Число потоков ядра может быть весьма значительным:

```
$ ps -ef | grep -F '[' | wc -l
78
```

Функции организации работы с потоками и механизмы синхронизации для них доступны после включения заголовочного файла <linux/sched.h>. **Макрос** current возвращает указатель текущую исполняющуюся задачу в циклическом списке задач, на соответствующую ей запись struct task_struct :

```
struct task_struct {
   volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
   void *stack;
...
   int prio, static_prio, normal_prio;
...
   pid_t pid;
...
   cputime_t utime, stime, utimescaled, stimescaled;
...
}
```

Это основная структура, один экземпляр которой соответствует любой выполняющейся задаче: будь то поток (созданный вызовом `kernel_thread()`) ядра, пользовательский процесс (главный поток этого процесса), или один из пользовательских потоков POSIX, созданных вызовом `pthread_create(...)` в рамках единого процесса - Linux не знает разницы (исполнительной) между потоками и процессами, все они порождаются одним системным вызовом `clone()`. В единственном случае текущему исполняющемуся коду нет соответствия в виде записи `struct task_struct()` — это контекст прерывания (обработчик аппаратного прерывания, или, как частный случай, таймерная функция, которые мы уже рассматривали). Но и в этом случае указатель `current` указывает на определённую запись задачи, только это — последняя (до прерывания) выполнявшаяся задача, не имеющая никакого касательства к текущему выполняющемуся коду (текущему контексту), `current` в этом случае указывает на мусор. И на это обстоятельство нужно обращать особое внимание — оно может стать предметом очень серьёзных шибок!

Потоки ядра

Уже было сказано, что ядро Linux является **вытесняющим**, в отличие от большинства ядер других UNIX-подобных операционных систем. Но вытеснение имеет смысл только в контексте наличия механизма параллельных ветвей выполнения. Этот механизм и предоставляется таким понятием как **потоки ядра**. Потоки ядра в Linux имеют много общего с потоками пользовательского пространства (`pthread_*`) и процессами пользовательского пространства (приложениями). Объединяет их то, что каждый из них имеет свою единственную структуру `struct task_struct`, содержащую всю информацию о потоке, составляющую **контекст потока**. Все такие структуры (контекстные структуры задач) связаны в сложную динамическую структуру (списковую): начав от **любой** структуры можно обойти структуры **всех** задач, существующих в системе (что и делает нам команда: `ps -ef`). Это отличает все перечисленные сущности (задачи) от кода обработчиков прерываний, которые не имеют своей структуры `struct task_struct` и о которых говорят, что они выполняются **в контексте прерываний**. Указателем на структуру текущего контекста (если он есть), служит **макрос** без параметров `current`.

Создание потока ядра

Для создания нового потока ядра используем вызов:

```
int kernel_thread( int (*fn)(void *), void *arg, unsigned long flags );
```

Параметры такого вызова понятны: функция потока, безтиповой указатель — параметр, передаваемый этой функции, и флаги, обычные для Linux вызова `clone()`. Возвращаемое функцией значение — это PID вновь созданного потока (если он больше нуля, а если он отрицательный, то это признак того, что что-то не заладилось, и это код ошибки).

А вот он же среди экспортируемых символов ядра:

```
$ cat /proc/kallsyms | grep kernel_thread
c0407c44 T kernel_thread
...
```

Примечание: Позже, при рассмотрении обработчиков прерываний, мы увидим механизм рабочих очередей (`workqueue`), обслуживаемый потоками ядра. Должно быть понятно, что уже одного такого механизма высокого уровня достаточно для инициации параллельных действий в ядре (с неявным использованием потоков ядра). Здесь же мы пока рассмотрим только

низкоуровневые механизмы, которые и лежат в базисе таких возможностей.

Первый простейший пример для прояснения того, как создаются потоки ядра (архив `thread.tgz`):

mod_thr1.c :

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

static int param = 3;
module_param( param, int, 0 );

static int thread( void * data ) {
   printk( KERN_INFO "thread: child process [%d] is running\n", current->pid );
   ssleep( param );                                  /* Пауза 3 с. или как параметр укажет... */
   printk( KERN_INFO "thread: child process [%d] is completed\n", current->pid );
   return 0;
}

int test_thread( void ) {
   pid_t pid;
   printk( KERN_INFO "thread: main process [%d] is running\n", current->pid );
   pid = kernel_thread( thread, NULL, CLONE_FS );     /* Запускаем новый поток */
   ssleep( 5 );                                      /* Пауза 5 с. */
   printk( KERN_INFO "thread: main process [%d] is completed\n", current->pid );
   return -1;
}

module_init( test_thread );
```

В принципе, этот модуль ядра ничего и не выполняет, за исключением того, что запускает новый поток ядра. При выполнении этого примера мы получим что-то подобное следующему:

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
$ time sudo insmod ./mod_thr1.ko
insmod: error inserting './mod_thr1.ko': -1 Operation not permitted
real    0m5.025s
user    0m0.004s
sys     0m0.012s
$ sudo cat /var/log/messages | tail -n30 | grep thread:
Jul 24 18:43:57 notebook kernel: thread: main process [12526] is running
Jul 24 18:43:57 notebook kernel: thread: child process [12527] is running
Jul 24 18:44:00 notebook kernel: thread: child process [12527] is completed
Jul 24 18:44:02 notebook kernel: thread: main process [12526] is completed
```

Примечание: Если мы станем выполнять пример с задержкой дочернего процесса больше родительского, то получим (после завершения запуска, при завершении созданного потока ядра!) сообщение Oops ошибки ядра:

```
$ sudo insmod ./mod_thr1.ko param=7
insmod: error inserting './mod_thr1.ko': -1 Operation not permitted
$
Message from syslogd@notebook at Jul 24 18:51:00 ...
 kernel:Oops: 0002 [#1] SMP
...
$ sudo cat /var/log/messages | tail -n70 | grep thread:
Jul 24 18:50:53 notebook kernel: thread: main process [12658] is running
```

```
Jul 24 18:50:53 notebook kernel: thread: child process [12659] is running
Jul 24 18:50:58 notebook kernel: thread: main process [12658] is completed
```

Последний параметр `flags` вызова `kernel_thread()` определяет детальный, побитово устанавливаемый набор тех свойств, которыми будет обладать созданный потока ядра, так как это вообще делается в практике Linux вызовом `clone()` (в этом месте, создании потоков-процессов, наблюдается существенное отличие Linux от традиций UNIX/POSIX). Часто в коде модулей можно видеть создание потока с таким набором флагов:

```
kernel_thread( thread_function, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD );
```

Свойства потока

Созданному потоку ядра (как и пользовательским процессам и потокам) присущ целый ряд параметров (`<linux/sched.h>`), часть которых будет иметь значения по умолчанию (такие, например, как параметры диспетчеризации), но которые могут быть и изменены. Для работы с параметрами потока используем следующие API:

1. Взаимно однозначное соответствие PID потока и соответствующей ему основной структуры данных, записи о задаче, которая уже обсуждалась (`struct task_struct`) — устанавливается в обоих направлениях вызовами:

```
static inline pid_t task_pid_nr( struct task_struct *tsk ) {
   return tsk->pid;
}
struct task_struct *find_task_by_vpid( pid_t nr );
```

Или, пользуясь описаниями из `<linux/pid.h>` :

```
// find_vpid() find the pid by its virtual id, i.e. in the current namespace
extern struct pid *find_vpid( int nr );
enum pid_type {
   PIDTYPE_PID,
   PIDTYPE_PGID,
   PIDTYPE_SID,
   PIDTYPE_MAX
};
struct task_struct *pid_task( struct pid *pid, enum pid_type );
struct task_struct *get_pid_task( struct pid *pid, enum pid_type );
struct pid *get_task_pid( struct task_struct *task, enum pid_type type );
```

В коде модуля это может выглядеть так:

```
struct task_struct *tsk;
tsk = find_task_by_vpid( pid );
```

Или так:

```
tsk = pid_task( find_vpid( pid ), PIDTYPE_PID );
```

2. Дисциплина планирования и параметры диспетчеризации, предписанные потоку, могут быть установлены в новые состояния так:

```
struct sched_param {
   int sched_priority;
};
int sched_setscheduler( struct task_struct *task, int policy, struct sched_param *parm );
// Scheduling policies
#define SCHED_NORMAL  0
#define SCHED_FIFO    1
#define SCHED_RR      2
#define SCHED_BATCH   3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE    5
```

3. Другие вызовы, имеющие отношение к приоритетам процесса:

```
void set_user_nice( struct task_struct *p, long nice );
int task_prio( const struct task_struct *p );
int task_nice( const struct task_struct *p );
```

4. Разрешения на использование выполнения на разных процессорах в SMP системах (афинити-маска процесса):

```
extern long sched_setaffinity( pid_t pid, const struct cpumask *new_mask );
extern long sched_getaffinity( pid_t pid, struct cpumask *mask );
```

- где (<linux/cpumask.h>):

```
typedef struct cpumask { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
```

Вообще, во всём связанном с созданием нового потока в ядре, прослеживаются прямые аналогии с созданием параллельных ветвей в пользовательском пространстве, что очень сильно облегчает работу с такими механизмами.

Новый интерфейс потоков

Несколько позже для потоков был добавлен API высокого уровня (в сравнении с `kernel_thread()`), упрощающий создание и заверение потоков (<linux/kthread.h>), а вызов `kernel_thread()` был объявлен устаревшим. Всё, сказанное выше относительно свойств и использования созданных потоков, остаётся в силе, новый интерфейс касается только самих фактов создания и завершения потока:

```
$ cat /proc/kallsyms | grep ' T ' | grep kthread
c043c7d7 T kthread_bind
c045b158 T kthread_should_stop
c045b171 T kthreadd
c045b2a2 T kthread_stop
c045b332 T kthread_create
```

Создание потока в высокоуровневом интерфейсе выполняет:

```
struct task_struct *kthread_create( int (*threadfn)(void *data),
                                    void *data, const char namefmt[], ... )
```

Принципиально отличие здесь то, что возвращается не PID созданного потока, а указатель структуры задачи. Поток таким вызовом создаётся в блокированном (ожидающем) состоянии, и для запуска (выполнения) должен быть разбужен вызовом `wake_up_process()`. Поскольку это весьма частая последовательность действий, то там же (<linux/kthread.h>) определён макрос (поэтому не ищите его в `/proc/kallsyms`):

```
kthread_run( threadfn, data, namefmt, ... )
```

Возвращает `struct task_struct*` или отрицательный код ошибки `ERR_PTR(-ENOMEM)`.

Начиная с третьего параметра функция `kthread_create()` и макрос `kthread_run()` подобны вызовам `printk()` или `sprintf()` с переменным числом параметров: третий параметр — это форматная строка (шаблон), а все последующие параметры — это значения, заполняющие этот формат. Получившаяся в итоге строка является идентификатором (именем) потока, под которым его знает ядро и под которым его показывает, например, команда `ps`.

Гораздо интереснее дело обстоит с завершением. Созданный поток может проверять (чаще всего периодически) необходимость завершения неблокирующим вызовом:

```
int kthread_should_stop( void );
```

Если внешний по отношению к функции потока код хочет завершить поток, то он вызывает для этого потока:

```
int kthread_stop( struct task_struct* );
```

Обнаружив это (по результату `kthread_should_stop()`) функция потока завершается. Обычно в коде потоковой функции это выглядит примерно так:

```
...
while( !kthread_should_stop() ) {
   // выполняемая работа потоковой функции
}
```

```
return 0;
...
```

Всё сказанное гораздо проще в комплексе увидеть на примере. Этот пример (архив `thread.tgz`) сложнее остальных, но он стоит того, чтобы его изучить детально:

mod_thr3.c :

```
#include <linux/module.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/jiffies.h>

static int N = 2;            // N - число потоков
module_param( N, int, 0 );


static char *sj( void ) {    // метка времени
   static char s[ 40 ];
   sprintf( s, "%08ld :", jiffies );
   return s;
}

static char *st( int lvl ) { // метка потока
   static char s[ 40 ];
   sprintf( s, "%skthread [%05d:%d]", sj(), current->pid, lvl );
   return s;
}

static int thread_fun1( void* data ) {
   int N = (int)data - 1;
   struct task_struct *t1 = NULL;
   printk( "%s is parent [%05d]\n", st( N ), current->parent->pid );
   if( N > 0 )
      t1 = kthread_run( thread_fun1, (void*)N, "my_thread_%d", N );
   while( !kthread_should_stop() ) {
      // выполняемая работа потоковой функции
      msleep( 100 );
   }
   printk( "%s find signal!\n", st( N ) );
   if( t1 != NULL ) kthread_stop( t1 );
   printk( "%s is completed\n", st( N ) );
   return 0;
}

static int test_thread( void ) {
   struct task_struct *t1;
   printk( "%smain process [%d] is running\n", sj(), current->pid );
   t1 = kthread_run( thread_fun1, (void*)N, "my_thread_%d", N );
   msleep( 10000 );
   kthread_stop( t1 );
   printk( "%smain process [%d] is completed\n", sj(), current->pid );
   return -1;
}

module_init( test_thread );
MODULE_LICENSE( "GPL" );
```

В этом коде запускается сколь угодно много потоков ядра (параметр N=... загрузки модуля), причём:

– потоки запускают последовательно один другого, поток `i` запускает поток `i + 1`;

– все потоки используют **одну** потоковую функцию;

– весь диагностический вывод сопровождается метками времени (`jiffies`), что позволяет подробно проследить хронологию событий;

– после выдержки паузы (10 сек.) главный поток посылает команду завершения (`kthread_stop()`), а далее потоки так же по цепочке посылают такую же команду друг другу.

Вот как это выглядит на исполнении:

```
$ time sudo insmod mod_thr3.ko N=3
insmod: error inserting 'mod_thr3.ko': -1 Operation not permitted
 real    0m10.140s
user    0m0.006s
sys     0m0.010s
$ ps -ef | grep '\[' | grep 'my_'
root     14603     2  0 19:00 ?        00:00:00 [my_thread_3]
root     14604     2  0 19:00 ?        00:00:00 [my_thread_2]
root     14605     2  0 19:00 ?        00:00:00 [my_thread_1]
$ dmesg | tail -n40 | grep -v audit
34167405 : main process [14602] is running
34167410 : kthread [14603:2] is parent [00002]
34167410 : kthread [14604:1] is parent [00002]
34167410 : kthread [14605:0] is parent [00002]
34177414 : kthread [14603:2] find signal!
34177511 : kthread [14604:1] find signal!
34177516 : kthread [14605:0] find signal!
34177516 : kthread [14605:0] is completed
34177516 : kthread [14604:1] is completed
34177516 : kthread [14603:2] is completed
34177516 : main process [14602] is completed
```

Здесь хорошо видно, что:

– порядок, в котором потоки создаются, и порядок, в котором они получают сигнал на завершение — **совпадают**...

– но порядок фактического завершения — в точности **обратный**, потому, что на выполнении `kthread_stop()` поток блокируется и ожидает завершения дочернего потока, и только после этого имеет право завершиться;

– видны задержки с малой дискретностью (100мс., см. код) между получением команды завершения, и отправкой его, из основного цикла потоковой функции, дальше по иерархии потоков;

– видно, что для выполняющегося потока родительским потоком становится PID=2 (демон `kthreadd`), хотя всё происходящее мы и наблюдаем ещё при выполнении запускающего **процесса** `insmod` (в функции инициализации модуля) — для созданных потоков выполняется операция **демонизации**.

Как уже легко видеть (даже без детального анализа), такой подход существенно упрощает синхронизацию завершения потоков, о которой мы будем говорить далее.

Синхронизация завершения

В предыдущем примере, попутно с основной иллюстрацией, была показана синхронизированная работа потоков: порождавшие потоки сигнализировали порождённым о необходимости их завершения, после чего дожидались этого их завершения. Это один из возможных видов синхронизации потоков. Другой, очень часто возникающий, случай, это так называемый **пул потоков** (статический или динамический, но сейчас мы будем говорит о статическом пуле): для распараллеливания работы создаётся несколько однотипных потоков, с **единой** для всех потоковой функцией. После последовательного запуска всех потоков пула порождающая единица ожидает завершения работы **всех** порождённых потоков.

В следующем примере (архив `tfor.tgz`) показан типовой, и не совсем понятный на первый взгляд, трюк, но повсеместно применяемый для решения такой задачи (в данном случае нас совершенно не интересует то, на каких примитивах синхронизации выполняется синхронизация завершения потоков, для этого могут использоваться разные примитивы, здесь важен принцип того, как это делается):

mod_for.c :

```
#include <linux/module.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/jiffies.h>
#include <linux/semaphore.h>

#include "../prefix.c"

#define NUM 3
static struct completion finished[ NUM ];

#define DELAY 1000
static int thread_func( void* data ) {
   int num = (int)data;
   printk( "! %s is running\n", st( num ) );
   msleep( DELAY - num );
   complete( finished + num );
   printk( "! %s is finished\n", st( num ) );
   return 0;
}

#define IDENT "for_thread_%d"
static int test_mlock( void ) {
   struct task_struct *t[ NUM ];
   int i;
   for( i = 0; i < NUM; i++ )
      init_completion( finished + i );
   for( i = 0; i < NUM; i++ )
      t[ i ] = kthread_run( thread_func, (void*)i, IDENT, i );
   for( i = 0; i < NUM; i++ )
      wait_for_completion( finished + i );
   printk( "! %s is finished\n", st( NUM ) );
   return -1;
}

module_init( test_mlock );
MODULE_LICENSE( "GPL" );
```

Включаемый файл `prefix.c` содержит описания диагностических функций `sj()` и `st()`, которые мы уже видели в предыдущем примере.

Прежде обратим внимание ещё на один небольшой трюк, который очень часто применяют в потоковом программировании, это идёт от потоков POSIX: когда при создании потока нужно передать ему в качестве параметра единственное **скалярное** значение, а потоковая функция ожидает указатель на данные потока, то к указателю приводится непосредственно это скалярное значение, никогда не бывшее указателем. В нашем случае это целочисленный индекс, 2-й параметр вызова:

```
kthread_run( thread_func, (void*)i, IDENT, i );
```

Но вернёмся к синхронизации завершений потоков. Дочерние потоки в этом примере создаются **последовательно** (от 0-го до 2-го) в цикле. Завершаться они могут в реальной ситуации в произвольные времена и в произвольном порядке, в примере искусственно смоделирована самая неблагоприятная ситуация, при которой потоки завершаются в порядке обратном их порождению. Но ожидается завершение потоков (в блокированном состоянии ожидающей программной единицы) опять таки всё в том же **последовательном**

порядке от 0-го до 2-го:

```
for( i = 0; i < NUM; i++ )
   wait_for_completion( finished + i );
```

Это не ошибка — это общеупотребимая практика! Нам нужно ожидание завершения всех порождённых потоков. Освободившись на очередном блокирующем ожидании завершения i-го потока, ожидающий код на последующих номерах уже **раньше** завершившихся потоков j>i просто не будет блокироваться, не задерживается, и мгновенно проскакивает их в цикле. Полностью такой цикл выйдет из блокированных состояний **только** когда завершатся **все** порождённые потоки:

```
$ sudo insmod mod_for.ko
insmod: error inserting 'mod_for.ko': -1 Operation not permitted
$ dmesg | tail -n30 | grep !
! 05019276 : kthread [08114:0] is running
! 05019276 : kthread [08115:1] is running
! 05019276 : kthread [08116:2] is running
! 05020275 : kthread [08116:2] is finished
! 05020276 : kthread [08115:1] is finished
! 05020277 : kthread [08114:0] is finished
! 05020277 : kthread [08113:3] is finished
```

Синхронизации в коде

Существует множество примитивов синхронизации, как теоретически проработанных, так и конкретно используемых и доступных в ядре Linux, и число их постоянно возрастает. Эта множественность связана, главным образом, с борьбой за эффективность (производительность) выполнения кода — для отдельных функциональных потребностей вводятся новые, более эффективные для этих **конкретных** нужд примитивы синхронизации. Тем не менее, основная сущность работы всех примитивов синхронизации остаётся одинаковой, в том ровно виде, как её впервые описал Э. Дейкстрой в своей знаменитой работе 1968г. «Взаимодействие последовательных процессов».

Критические секции кода и защищаемые области данных

Для решения задачи синхронизации в ядре Linux существует множество механизмов синхронизации (сами объекты синхронизации называют примитивами синхронизации) и появляются всё новые и новые... , некоторые из механизмов вводятся даже для поддержки единичных потребностей. Условно, по функциональному использованию, примитивы синхронизации можно разделить на (хотя такое разделение часто оказывается весьма условным):

- Примитивы для защиты фрагментов исполняемого кода (критических секций) от одновременного (или псевдо-одновременного) исполнения. Классический пример: мьютекс, блокировки чтения-записи...
- Примитивы для защиты областей данных от несанкционированного изменений: атомарные переменные и операции, счётные семафоры...

Механизмы синхронизации

Обычно все предусмотренные версией ядра примитивы синхронизации доступные после включения заголовочного файла `<linux/sched.h>`. Ниже будут рассмотрены только некоторые из механизмов, такие как:

- переменные, локальные для каждого процессора (per-CPU variables), интерфейс которых описан в файле `<linux/percpu.h>`;
- атомарные переменные (описаны в архитектурно-зависимых файлах `<atomic*.h>`);
- спин-блокировки (`<linux/spinlock.h>`);
- сериальные (последовательные) блокировки (`<linux/seqlock.h>`);
- семафоры (`<linux/semaphore.h>`);
- семафоры чтения и записи (`<linux/rwsem.h>`);

– мьютексы реального времени (<linux/rtmutex.h>);

– механизмы ожидания завершения (<linux/completion.h>);

Рассмотрение механизмов синхронизаций далее проведено как-раз в обратном порядке, с ожидания завершения, потому, что это естественным образом продолжает начатое выше рассмотрение потоков ядра.

Сюда же, к механизмам синхронизации, можно, хотя и достаточно условно, отнести механизмы, предписывающие заданный порядок выполнения операций, и препятствующие его изменению, например в процессе оптимизации кода (обычно их так и рассматривают совместно с синхронизациями, по принципу: «ну, надо же их где-то рассматривать?»).

Условные переменные и ожидание завершения

Естественным сценарием является запуск некоторой задачи в отдельном потоке и последующее ожидание завершения ее выполнения (см. аварийное завершение выше). В ядре нет аналога функции ожидания завершения потока, вместо нее требуется явно использовать механизмы синхронизации (аналогичные POSIX 1003.b определению барьеров pthread_barrier_t). Использование для ожидания какого-либо события обычного семафора не рекомендуется: в частности, реализация семафора оптимизирована исходя из предположения, что обычно (основную часть времени жизни) они открыты. Для этой задачи лучше использовать не семафоры, а специальный механизм ожидания выполнения - completion (в терминологии **ядра** Linux он называется условной переменной, но разительно отличается от условной переменной как её понимает стандарт POSIX). Этот механизм (<linux/completion.h>) позволяет одному или нескольким потокам ожидать наступления какого-то события, например, завершения другого потока, или перехода его в состояние готовности выполнять работу. Следующий пример демонстрирует запуск потока и ожидание завершения его выполнения (это минимальная модификация для сравнения примера запуска потока ранее):

mod_thr2.c :

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

static int thread( void * data ) {
   struct completion *finished = (struct completion*)data;
   struct task_struct *curr = current;      /* current - указатель на дескриптор текущей задачи */
   printk( KERN_INFO "child process [%d] is running\n", curr->pid );
   msleep( 10000 );                         /* Пауза 10 с. */
   printk( KERN_INFO "child process [%d] is completed\n", curr->pid );
   complete( finished );                    /* Отмечаем факт выполнения условия. */
   return 0;
}

int test_thread( void ) {
   DECLARE_COMPLETION( finished );
   struct task_struct *curr = current;
   printk( KERN_INFO "main process [%d] is running\n", curr->pid );
   pid_t pid = kernel_thread( thread, &finished, CLONE_FS ); /* Запускаем новый поток */
   msleep( 5000 );                                           /* Пауза 5 с. */
   wait_for_completion( &finished );                         /* Ожидаем выполнения условия */
   printk( KERN_INFO "main process [%d] is completed\n", curr->pid );
   return -1;
}

module_init( test_thread );
```

Выполнение этого примера разительно отличается от его предыдущего прототипа (обратите внимание на временные метки сообщений!):

```
$ sudo insmod ./mod_thr2.ko
insmod: error inserting './mod_thr2.ko': -1 Operation not permitted
$ sudo cat /var/log/messages | tail -n4
Apr 17 21:20:23 notebook kernel: main process [12406] is running
Apr 17 21:20:23 notebook kernel: child process [12407] is running
Apr 17 21:20:33 notebook kernel: child process [12407] is completed
Apr 17 21:20:33 notebook kernel: main process [12406] is completed
$ ps -A | grep 12406
$ ps -A | grep 12407
$
```

Переменные типа `struct completion` могут определяться либо как в показанном примере статически, макросом:

```
DECLARE_COMPLETION( name );
```

Либо инициализироваться динамически:

```
void init_completion( struct completion * );
```

Примечание: Всё разнообразие в Linux как потоков ядра (`kernel_thread()`), так и параллельных процессов (`fork()`) и потоков пространства пользователя (`pthread_create()`) обеспечивается тем, что потоки и процессы в этой системе фактически не разделены принципиально, и те и другие создаются единым системным вызовом `clone()` - все различия создания определяются набором флагов вида `CLONE_*` для создаваемой задачи (последний параметр `kernel_thread()` нашего примера).

Объект на сегодня описан (`<linux/completion.h>`) как очередь FIFO (стек) потоков, ожидающих события:

```
/*
 * struct completion - structure used to maintain state for a "completion"
 * This is the opaque structure used to maintain the state for a "completion".
 * Completions currently use a FIFO to queue threads that have to wait for
 * the "completion" event.
 */
struct completion {
        unsigned int done;
        wait_queue_head_t wait;
};
```

Основные операции:

```
extern void wait_for_completion( struct completion* );
extern unsigned long wait_for_completion_timeout( struct completion *x,
                                                  unsigned long timeout );
extern bool try_wait_for_completion( struct completion *x );
```

Это операции блокирования и ожидания освобождение...

```
extern void complete( struct completion* );
extern void complete_all( struct completion* );
```

А это операции освобождения ожидающих потоков: одного на вершине стека, последнего пришедшего, или всех ожидающих в очереди, соответственно.

Атомарные переменные и операции

Атомарные переменные — это наименее ресурсоёмкие средства обеспечения атомарного выполнения операций (там, где их минимальных возможностей достаточно). Реализуются в платформенно зависимой части кода ядра. Важные качества атомарных переменных и операций: а). компилятор (по ошибке, пытаясь повысить эффективность кода) не будет оптимизировать операции обращения к атомарным переменным, б). атомарные операции скрывают различия между реализациями для различных аппаратных платформ.

Функции, реализующие атомарные операции можно разделить на 2 группы по способу выполнения: а).атомарные операции, устанавливающие новые значения и б). атомарные операции, которые обновляют значения, при этом возвращая предыдущее установленное значение (обычно это функции вида `test_and_*()`). С другой стороны, по представлению данных, с которыми они оперируют, атомарные операции также делятся на 2 группы по типу объекта: а).оперирующие с целочисленными значениями (арифметические) и б).оперирующие с последовательным набором бит. Атомарных операций, в итоге, великое множество, и далее обсуждаются только некоторые из них.

Битовые атомарные операции

Определены в `<asm-generic/bitops.h>` и целым каталогом описаний `<asm-generic/bitops/*.h>`. Битовые атомарные операции выполняют действия над обычными операндами типа `unsigned long`, первым операндом вызова является номер бита (0 — младший, ограничения на старший номер не вводится, для 32-бит процессоров это 31, для 64-бит процессоров 63):

`void set_bit( int n, void *addr );` - установить n-й бит

`void clear_bit( int n, void *addr );` - очистить n-й бит

`void change_bit( int n, void *addr );` - инвертировать n-й бит

`int test_and_set_bit( int n, void *addr );` - установить n-й бит и возвратить предыдущее значение этого бита

`int test_and_clear_bit( int n, void *addr );` - очистить n-й бит и возвратить предыдущее значение этого бита

`int test_and_change_bit( int n, void *addr );` - инвертировать n-й бит и возвратить предыдущее значение этого бита

`int test_bit( int n, void *addr );` - возвратить значение n-го бита

Пример того, как могут использоваться битовые атомарные переменные:

```
unsigned long word = 0;
set_bit( l, &word );        /* атомарно устанавливается бит 1 */
clear_bit( 1, &word );      /* атомарно очищается бит 1 */
change_bit( 1, &word );     /* атомарно инвертируется бит 1, теперь он опять установлен */
if( test_and_clear_bit( 1, &word ) ) {  /* очищается бит 1, возвращается значение этого бита 1 */
   /* в таком виде условие выполнится ... */
}
```

Арифметические атомарные операции

Реализуются в машинно-зависимом коде, описаны, например:

```
$ ls /lib/modules/`uname -r`/build/include/asm-generic/atomic*
/lib/modules/2.6.32.9-70.fc12.i686.PAE/build/include/asm-generic/atomic64.h
/lib/modules/2.6.32.9-70.fc12.i686.PAE/build/include/asm-generic/atomic.h
/lib/modules/2.6.32.9-70.fc12.i686.PAE/build/include/asm-generic/atomic-long.h
```

Эта группа атомарных операций работает над операндами специального типа (в отличие от битовых операций). Вводятся специальные типы: `atomic_t, atomic64_t, atomic_long_t,` ...

`ATOMIC_INIT( int i )` - объявление и инициализация в значение i переменной типа `atomic_t`

`int atomic_read( atomic_t *v );` - считывание значения в целочисленную переменную

`void atomic_set( atomic_t *v, int i );` - установить переменную v в значение i

`void atomic_add ( int i, atomic_t *v ) ;` - прибавить значение i к переменной v

`void atomic_sub( int i, atomic_t *v ) ;` - вычесть значение i из переменной v

`void atomic_inc( atomic_t *v ) ;` - инкремент v

`void atomic_dec( atomic_t *v ) ;` - декремент v

`int atomic_sub_and_test( int i, atomic_t *v );` - вычесть i из переменной v , возвратить true, если результат равен нулю, и false в противном случае

`int atomic_add_negative( int i, atomic_t *v );` - прибавить i к переменной v, возвратить true, если результат операции меньше нуля, иначе возвратить false

`int atomic_dec_and_test( atomic_t *v );` - декремент v, возвратить true, если результат равен нулю, и false в противном случае

`int atomic_inc_and_test( atomic_t *v );` - инкремент v, возвратить true, если результат равен нулю, и false в противном случае

Объявление атомарных переменных и запись атомарных операций не вызывает сложностей (аналогична работе с обычными переменными):

```
atomic_t v = ATOMIC_INIT( 111 );  /* определение переменной и инициализация ее значения */
atomic_add( 2, &v ) ;             / * v = v + 2  */
atomic_inc( &v );                 / * v++ */
```

В поздних версиях ядра набор атомарных переменных существенно расширен такими типами (64 бит), такими как:

```
typedef struct {
   long long counter;
} atomic64_t;
typedef atomic64_t atomic_long_t;
```

И соответствующими для них операциями:

```
ATOMIC64_INIT( long long ) ;
long long atomic64_add_return( long long a, atomic64_t *v );
long long atomic64_xchg( atomic64_t *v, long long new );
...
ATOMIC_LONG_INIT( long )
void atomic_long_set( atomic_long_t *l, long i );
long atomic_long_add_return( long i, atomic_long_t *l );
int atomic_long_sub_and_test( long i, atomic_long_t *l );
...
```

Локальные переменные процессора

Переменные, закреплённые за процессором (per-CPU data). Определены в `<linux/percpu.h>`. Основное достоинство таких переменных в том, что если некоторую функциональность можно разумно распределить между такими переменными, то они не потребуют взаимных блокировок доступа в SMP. API, предоставляемые для работы с локальными данными процессора, на время работы с такими переменными запрещают вытеснение в режиме ядра.

Вторым свойством локальных данных процессора является то, что такие данные позволяют существенно уменьшить недостоверность данных, хранящихся в кэше. Это происходит потому, что процессоры поддерживают свои кэши в синхронизированном состоянии. Если один процессор начинает работать с данными, которые находятся в кэше другого процессора, то первый процессор должен обновить содержимое своего кэша. Постоянное аннулирование находящихся в кэше данных, именуемое перегрузкой кэша (cash thrashing), существенно снижает производительность системы (до 3-4-х раз). Использование данных, связанных с процессорами, позволяет приблизить эффективность работы с кэшем к максимально возможной, потому что в идеале каждый процессор работает только со своими данными.

Предыдущая модель

Эта модель существует со времени ядер 2.4, но она остаётся столь же функциональной и широко используется и сейчас; в этой модели локальные данные процессора представляются как массив (любой

структурной сложности элементов), который индексируется номером процессора (начиная с 0 и далее...), работа этой модели базируется на вызовах:

`int get_cpu();` - получить номер текущего процессора и запретить вытеснение в режиме ядра.

`put_cpu();` - разрешить вытеснение в режиме ядра.

Пример работы в этой модели:

```
int data_percpu[] = { 0, 0, 0, 0 };
int cpu = get_cpu();
data_percpu[ cpu ]++;
put_cpu();
```

Понятно, что поскольку запрет вытеснения в режиме ядра является принципиально важным условием, код, работающий с локальными переменными процессора, **не должен переходить в блокированное состояние** (по собственной инициативе). Почему код, работающий с локальными переменными процессора не должен вытесняться? :

- Если выполняющийся код вытесняется и позже восстанавливается для выполнения на другом процессоре, то значение переменной `cpu` больше не будет актуальным, потому что эта переменная будет содержать номер другого процессора.

- Если некоторый другой код вытеснит текущий, то он может параллельно обратиться к переменной `data_percpu[]` на том же процессоре, что соответствует состоянию гонок за ресурс.

Новая модель

Новая модель введена рассчитывая на будущие развитие, и на обслуживание весьма большого числа процессоров в системе, она упрощает работу с локальными переменными процессора, но на настоящее время ещё не так широко используется.

Статические определения (на этапе компиляции):

```
DEFINE_PER_CPU( type, name );
```

- создается переменная типа `type` с именем `name`, которая имеет отдельный экземпляр для каждого процессора в системе, если необходимо объявить такую переменную с целью избежания предупреждений компилятора, то необходимо использовать другой макрос:

```
DECLARE_PER_CPU( type, name );
```

Для работы с экземплярами этих переменных используются макросы:

`- get_cpu_var( name );` - вызов возвращает L-value экземпляра указанной переменной на текущем процессоре, при этом запрещается вытеснение кода в режиме ядра.

`- put_cpu_var( name );` - разрешает вытеснение.

Ещё один вызов возвращает L-value экземпляра локальной переменной другого процессора:

`- per_cpu( name, int cpu );` - этот вызов не запрещает вытеснение кода в режиме ядра и не обеспечивает никаких блокировок, для его использования необходимы внешние блокировки в коде.

Пример статически определённой переменной:

```
DECLARE_PER_CPU( long long, xxx );
get_cpu_var( xxx )++;
put_cpu_var( xxx );
```

Динамические определения (на этапе выполнения) — это другая группа API: динамически выделяют области фиксированного размера, закреплённые за процессором:

```
void *alloc_percpu( type );
void *__alloc_percpu( size_t size, size_t align );
void free_percpu( const void *data );
```

Функции размещения возвращают указатель на экземпляр области данных, а для работы с таким указателем вводятся вызовы, аналогичные случаю статического распределения:

- `get_cpu_ptr( ptr );` - вызов возвращает указатель (типа void*) на экземпляра указанной переменной на текущем процессоре, при этом запрещается вытеснение кода в режиме ядра.

- `put_cpu_ptr( ptr );` - разрешает вытеснение.

- `per_cpu_ptr( ptr, int cpu );` - возвращает указатель на экземпляра указанной переменной на **другом** процессоре.

Пример динамически определённой переменной:

```
long long *xxx = (long long*)alloc_percpu( long long );
++*get_cpu_ptr( xxx );
put_cpu_var( xxx );
```

Требование не блокируемости кода, работающего с локальными данными процесса, остаётся актуальным и в этом случае.

Примечание: Легко видеть, что новая модель (будь это группа API, работающая с самими переменными, или с указателями на них) не так уж значительно отличается от предыдущей: устранена необходимость явного индексирования массива экземпляров по номеру процессора; это делается внутренними скрытыми механизмами, и окончательно возвращается уже индексированный экземпляр, связанный с текущим процессором.

Блокировки

Различные виды блокировок используются для того, чтобы оградить критический участок кода от одновременного исполнения. В этом смысле блокировки гораздо ближе к защите участков кода, чем к защите областей данных, хотя семафоры, например, (не бинарные) используются, главным образом, именно для ограничения доступа к данным: классические задачи производители-потребители.

До появления и широкого распространения SMP, когда параллелизмы были квази-параллелизмами, блокировки использовались в своём классическом варианте (Э. Дейкстра), они защищали критические области от последовательного доступа несколькими вытесненными процессами. Такие механизмы работают на вытеснении запрашивающих процессов в блокированное состояние до времени освобождения критических ресурсов. Эти блокировки мы будем называть **пассивными** блокировками. При таких блокировках процессор прекращает (в точке блокирования) выполнение текущего процесса и переключается на выполнение другого процесса (возможно idle).

Принципиально другой вид блокировок — **активные** блокировки — появляются только в SMP системах, когда процессор в ожидании недоступного пока ресурса не переводится в блокированное состояние, а «накручивает» в ожидании освобождения ресурса «пустые» циклы. В этом случае, процессор не освобождается на выполнение другого ожидающего процесса в системе, а продолжает активное выполнение («пустых» циклов) в контексте текущего процесса.

Эти два рода блокировок (каждый из которых включает несколько подвидов) принципиально отличаются:

- возможностью использования: пассивно заблокировать (переключить контекст) можно только такую последовательность кода, которая имеет свой собственный контекст (запись задачи), куда позже можно вернуться (активировать процесс) — в обработчиках прерываний или тасклетах это не так;
- эффективностью: активные блокировки не всегда проигрывают пассивным в производительности, переключение контекста в системе это очень трудоёмкий процесс, поэтому для ожидания короткого интервала времени активные блокировки могут оказаться даже эффективнее, чем пассивные;

Семафоры (мьютексы)

Семафоры ядра определены в `<linux/semaphore>`. Так как задачи, которые конфликтуют при захвате блокировки, переводятся в состояние ожидания и в этом состоянии ждут, пока блокировка не будет

освобождена, семафоры хорошо подходят для блокировок, которые могут удерживаться в течение длительного времени. С другой стороны, семафоры не оптимальны для блокировок, которые удерживаются в течение очень короткого периода времени, так как накладные затраты на перевод процессов в состояние ожидания могут превысить время, в течение которого удерживается блокировка. Существует очевидное ограничение на использование семафоров в ядре: их невозможно использовать в том коде, который не должен перейти в блокированное состояние, например, при обработке верхней половины прерываний.

В то время как спин-блокировки позволяют удерживать блокировку только одной задаче в любой момент времени, количество задач (`count`), которым разрешено одновременно удерживать семафор (владеть семафором), может быть задано при декларации переменной семафора:

```
struct semaphore {
   spinlock_t lock;
   unsigned int count;
   struct list_head wait_list;
};
```

Если значение `count` больше 1, то семафор называется счетным семафором, и он допускает количество потоков, которые одновременно удерживают блокировку, не большее чем значение счетчика использования (`count`). Часто встречается ситуация, когда разрешенное количество потоков, которые одновременно могут удерживать семафор, равно 1 (как и для спин-блокировок), в этом семафоры называются бинарными семафорами, или взаимоисключающими блокировками (`mutex`, мьютекс, потому что он гарантирует взаимоисключающий доступ — mutual exclusion). Бинарные семафоры (мьютексы) используются для обеспечения взаимоисключающего доступа к фрагментам кода, называемым критической секцией, и в таком качестве и состоит их наиболее частое использование.

Примечание: Независимо от того, определено ли поле владельца захватившего мютекс (как это делается по разному в различных POSIX-совместимых ОС), принципиальными особенностями мютекса, вытекающими из его логики, в отличии от счётного семафора будет то, что: а). у захваченного мютекса всегда будет и **единственный** владелец, его захвативший, и б). освободить блокированные на мютексе потоки (освободить мютекс) может только один владеющий мютексом поток; в случае счётного семафора освободить блокированные на семафоре потоки может **любой** из потоков, владеющий семафором.

Статическое определение и инициализация семафоров выполняется макросом:

```
static DECLARE_SEMAPHORE_GENERIC( name, count );
```

Для создания взаимоисключающей блокировки (mutex), что используется наиболее часто, есть более короткая запись:

```
static DECLARE_MUTEX( name );
```

- где в обоих случаях `name` — это имя переменной типа семафор.

Но чаще семафоры создаются динамически, как часть больших структур данных. В таком случае для инициализации счётного семафора используется функция:

```
void sema_init( struct semaphore *sem, int val );
```

А вот такая же инициализация для бинарных семафоров (мютексов) — макросы:

```
init_MUTEX( struct semaphore *sem );
init_MUTEX_LOCKED( struct semaphore *sem );
```

В операционной системе Linux для захвата семафора (мютекса) используется операция `down()`, она уменьшает его счетчик на единицу. Если значение счетчика больше или равно нулю, то блокировка захватывается успешно (задача может входить в критический участок). Если значение счетчика (после декремента) меньше нуля, то задание помещается в очередь ожидания и процессор переходит к выполнению других задач. Метод `up()` используется для того, чтобы освободить семафор (после завершения выполнения критического участка), его выполнение увеличивает счётчик семафора на единицу.

Операции над семафорами:

```
void down( struct semaphore *sem );
```

```
int down_interruptible( struct semaphore *sem );
int down_killable( struct semaphore *sem );
int down_trylock( struct semaphore *sem );
int down_timeout( struct semaphore *sem, long jiffies );
void up( struct semaphore *sem );
```

`down_interruptible()` - выполняет попытку захватить семафор. Если эта попытка неудачна, то задача переводится в блокированное состояние с флагом `TASK_INTERRUPTIBLE` (в структуре задачи). Такое состояние процесса означает, что задание может быть возвращено к выполнению с помощью сигнала, а такая возможность обычно очень ценная. Если сигнал приходит в то время, когда задача блокирована на семафоре, то задача возвращается к выполнению, а функция `down_interruptible()` возвращает значение `-EINTR`.

`down()` - переводит задачу в блокированное состояние ожидания с флагом `TASK_UNINTERRUPTIBLE`. В большинстве случаев это нежелательно, так как процесс, который ожидает на освобождение семафора, не будет отвечать на сигналы.

`down_trylock()` - используется для неблокирующего захвата семафора. Если семафор уже захвачен, то функция немедленно возвращает ненулевое значение. В случае успешного захвата семафора возвращается нулевое значение и захватывается блокировка.

`down_timeout()` - используется для попытки захвата семафора на протяжении интервала времени `jiffies` системных тиков.

`up()` - инкрементирует счётчик семафора, если есть блокированные на семафоре потоки, то **один** из них может захватить блокировку (принципиальным является то, что какой конкретно поток из числа заблокированных - **непредсказуемо**).

Спин-блокировки

Блокирующая попытка входа в критическую секцию при использовании семафоров означает потенциальный перевод задачи в блокированное состояние и переключение контекста, что является дорогостоящей операцией. Для синхронизации в случае, когда: а). контекст выполнения не позволяет переходить в блокированное состояние (контекст прерывания), или б). требуется кратковременная блокировка без переключение контекста - используются спин-блокировки (`spinlock_t`), представляющие собой активное ожидание освобождения в пустом цикле. Если необходимость синхронизации связана только с наличием в системе нескольких процессоров, то для небольших критических секций следует использовать спин-блокировку, основанную на простом ожидании в цикле. Спин-блокировка может быть только бинарной. По `spinlock_t` достаточно много определений разбросано по нескольким заголовочным файлам:

```
$ ls spinlock*
spinlock_api_smp.h  spinlock_api_up.h  spinlock.h  spinlock_types.h  spinlock_types_up.h
spinlock_up.h

typedef struct {
   raw_spinlock_t raw_lock;
...
} spinlock_t;
```

Для инициализации `spinlock_t` (и родственного типа `rwlock_t`, о котором детально ниже) раньше (и в литературе) использовались макросы:

```
spinlock_t lock = SPIN_LOCK_UNLOCKED;
rwlock_t lock = RW_LOCK_UNLOCKED;
```

Но сейчас мы можем читать в комментариях:

```
// SPIN_LOCK_UNLOCKED and RW_LOCK_UNLOCKED defeat lockdep state tracking and are hence deprecated.
```

- то есть, эти макроопределения объявлены не поддерживаемыми, и могут быть исключены в любой последующей версии. Для определения и инициализации используем новые макросы (эквивалентные по смыслу записанным выше) вида:

```
DEFINE_SPINLOCK( lock );
DEFINE_RWLOCK( lock );
```

- это, как и обычно, статические определения отдельных (автономных) переменных типа `spinlock_t`. И так же,

как и для других примитивов, может быть динамическая инициализация ранее объявленной переменной (чаще эта переменная — поле в составе более сложной структуры):

```
void spin_lock_init( spinlock_t *sl );
```

Основной интерфейс `spinlock_t` (основная пара операций: захват и освобождение):

```
spin_lock ( spinlock_t *sl );
spin_unlock( spinlock_t *sl );
```

Если при компиляции ядра не установлено SMP (использование много-процессорности) и не конфигурировано вытеснение кода в ядре (наличие одновременно 2-х этих условий), то `spinlock_t` вообще не компилируются (на их месте остаются пустые места) за счёт препроцессорных директив условной трансляции.

Примечание: В отличие от реализаций в некоторых других операционных системах, спин-блокировки в операционной системе Linux не рекурсивны. Это означает, что код:

```
DEFINE_SPINLOCK( lock );
spin_lock( &lock );
spin_lock( &lock );
```

- обречён на дэдлок — процессор будет активно выполнять этот фрагмент до бесконечности (то есть происходит деградация системы — число доступных в системе процессоров уменьшается)...

Вот такой рекурсивный захват спин-блокировки может неявно происходить в обработчике прерываний, поэтому перед захватом такой блокировки нужно запретить прерывания на локальном процессоре. Это общий случай, поэтому для него предоставляется специальный интерфейс:

```
DEFINE_SPINLOCK( lock );
unsigned long flags;
spin_lock_irqsave( &lock, flags );
/* критический участок ... */
spin_unlock_irqre_store( &lock, flags );
```

Для спин-блокировки определены ещё такие интерфейсы, как:

- попытка захвата без блокирования, если блокировка уже захвачена, функция возвратит ненулевое значение:

```
int spin_try_lock( spinlock_t *sl );
```

- возвращает ненулевое значение, если блокировка в данный момент захвачена:

```
int spin_is_locked( spinlock_t *sl );
```

Блокировки чтения-записи

Особым, но часто встречающимся, случаем синхронизации являются случай «читателей» и «писателей». Читатели только читают состояние некоторого ресурса, и поэтому могу осуществлять к нему параллельный доступ. Писатели изменяют состояние ресурса, и в силу этого писатель должен иметь к ресурсу монопольный доступ (только один писатель), причем чтение ресурса (для всех читателей) в этот момент времени так же должно быть заблокировано. Для реализации блокировок чтения-записи в ядре Linux существуют отдельные версии для семафоров и спин-блокировок. Мьютексы реального времени не имеют реализации для случая читателей и писателей.

Примечание: Обратим здесь внимание на то, что в точности той же функциональности мы могли бы достигнуть и используя классические примитивы синхронизации (мьютекс или спинлок), просто захватывая критический участок независимо от типа предстоящей операции. Блокировки чтения-записи введены из соображений **эффективности** реализации для очень типового случая применения.

В случае семафоров, вместо структуры `struct semaphore` вводится `struct rw_semaphore`, а набор интерфейсных функций захвата освобождения (простые `down()/up()`) расширяется до:

`down_read( &rwsem );` - попытка захватить семафор для чтения

`up_read( &rwsem );` - освобождение семафора для чтения

`down_write( &rwsem );` - попытка захватить семафор для записи

`up_write( &rwsem );` - освобождение семафора для записи

Семантика этих операций следующая:

- если семафор ещё не захвачен, то любой захват (`down_read(), down_write()`) будет успешным (без блокирования);
- если семафор захвачен уже для **чтения**, то последующие сколь угодно много попыток захвата семафора для чтения (`down_read()`) будут завершаться успешно (без блокирования), но запрос на захват такого семафора для записи (`down_write()`) закончится блокированием;
- если семафор захвачен уже для **записи**, то любая последующая попытка захвата семафора (независимо, `down_read()` это или `down_write()`) закончится блокированием;

Статически определенный семафор чтения-записи создаётся макросом:

```
static DECLARE_RWSEM( name );
```

Семафоры чтения-записи, которые создаются динамически, должны быть инициализированы с помощью функции:

```
void init_rwsem( struct rw_semaphore *sem );
```

Примечание: Из описаний инициализаторов видно, что семафоры чтения-записи являются исключительно бинарными (не счётными), то есть (в терминологии Linux) фактически не семафорами, а мютексами.

Пример того, как могут быть использованы семафоры чтения-записи при работе (обновлении и считывании) циклических списков Linux (о которых мы говорили ранее):

```
struct data {
   int value;
   struct list_head list;
};
static struct list_head list;
static struct rw_semaphore rw_sem;
int add_value( int value ) {
   struct data *item;
   item = kmalloc( sizeof(*item), GFP_ATOMIC );
   if ( !item ) goto out;
   item->value = value;
   down_write( &rw_sem );              /* захватить для записи */
   list_add( &(item->list), &list );
   up_write( &rw_sem );                /* освободить по записи */
   return 0;
out:
   return -ENOMEM;
}
int is_value( int value ) {
   int result = 0;
   struct data *item;
   struct list_head *iter;
   down_read( &rw_sem );               /* захватить для чтения */
   list_for_each( iter, &list ) {
      item = list_entry( iter, struct data, list );
      if( item->value == value ) {
         result = 1; goto out;
      }
   }
out:
```

```
    up_read( &rw_sem );                /* освободить по чтению */
    return result;
}
void init_list( void ) {
    init_rwsem( &rw_sem );
    INIT_LIST_HEAD( &list );
}
```

Точно так же, как это сделано для семафоров, вводится и блокировка чтения-записи для спин-блокировки:

```
typedef struct {
    raw_rwlock_t raw_lock;
...
} rwlock_t;
```

С набором операций:

```
read_lock( rwlock_t *rwlock );
read_unlock( rwlock_t *rwlock );
write_lock( rwlock_t *rwlock );
write_unlock ( rwlock_t *rwlock );
```

Примечание: Если при компиляции ядра не установлено SMP и не конфигурировано вытеснение кода в ядре, то `spinlock_t` вообще не компилируются (на их месте остаются пустые места), а, значит, соответственно и `rwlock_t`.

Примечание: Блокировку, захваченную для чтения, уже нельзя далее «повышать» до блокировки, захваченной для записи; последовательность операторов:

```
read_lock( &rwlock );
write_lock( &rwlock );
```

- гарантирует нам дэдлок, так как при захвате блокировки на запись будет выполняться периодическая проверка, пока все потоки, которые захватили блокировку для чтения, ее не освободили, это касается и текущего потока, который не сделает этого никогда... Но несколько потоков **чтения** безопасно могут захватывать одну и ту же блокировку чтения-записи, поэтому один поток также может безопасно рекурсивно захватывать одну и ту же блокировку для чтения несколько раз, например в обработчике прерываний без запрета прерываний.

На момент создания механизма блокировок чтения-записи, их использованию прогнозировали значительное повышение производительности, и они вызывали заметный энтузиазм разработчиков. Но последующая практика показала, что этому механизму присуща скрытая опасность того, что при **высокой и равномерной** плотности запросов чтения, запрос на модификацию (запись) структуры записи может отсрочиваться на неограниченно большие интервалы времени. Об этой особенности нужно помнить, взвешивая применение этого механизма в своём коде. Частично смягчить это ограничение пытается следующий подвид блокировок — сериальные блокировки.

Сериальные (последовательные) блокировки

Это пример одного только из нескольких механизмов синхронизации, которые и блокировками по существу (в полной мере) не являются... Это подвид блокировок чтения-записи. Такой механизм добавлен для получения эффективных по времени реализаций. Описаны в `<linux/seqlock.h>`, для их представления вводится тип `seqlock_t`:

```
typedef struct {
    unsigned sequence;
    spinlock_t lock;
} seqlock_t;
```

Такой элемент блокировки создаётся и инициализируется статически :

```
seqlock_t lock = SEQLOCK_UNLOCKED;
```

Или эквивалентная динамическая инициализация:

```
seqlock_t lock;
seqlock_init( &lock );
```

Доступ на чтение работает получая целочисленное значение (без знака) последовательности (ключ) на входе в защищаемую критическую секцию. На выходе из этой секции это значение должно сравниваться с текущим таким значением (на момент завершения); если есть несоответствие, то значит секция (за это время!) обрабатывалась операциями записи, и проделанное чтение должно быть повторено. В результате, код читателя имеет вид подобный:

```
seqlock_t lock = SEQLOCK_UNLOCKED;
unsigned int seq;
do {
   seq = read_seqbegin( &lock );
   /* ... */
} while read_seqretry( &lock, seq );
```

Блокировка по записи реализована через спин-блокировку. Писатели должны получить эксклюзивную спин-блокировку, чтобы войти в критическую секцию, защищаемую последовательной блокировкой. Чтобы это сделать, код писателя делает вызов функции:

```
static inline void write_seqlock( seqlock_t *sl ) {
   spin_lock(&sl->lock);
   ++sl->sequence;
   smp_wmb();
}
```

Снятие блокировки записи выполняет другая функция:

```
static inline void write_sequnlock( seqlock_t *sl ) {
   smp_wmb();
   sl->sequence++;
   spin_unlock(&sl->lock);
}
```

Здесь любопытно то, что писатель делает инкремент ключа последовательности дважды: после захвата спин-блокировки и перед её освобождением. В этой связи интересно посмотреть реализацию того, как читатель получает своё начальное значение ключа вызовом `read_seqbegin()`:

```
static __always_inline unsigned read_seqbegin( const seqlock_t *sl ) {
   unsigned ret;
repeat:
   ret = sl->sequence;
   smp_rmb();
   if( unlikely( ret & 1 ) ) {
      cpu_relax();
      goto repeat;
   }
   return ret;
}
```

Отсюда понятно, что если читатель запросит код последовательного доступа в то время, когда в критической секции находится писатель (писатель сделал начальный инкремент, но не сделал завершающий), то запросивший читатель будет выполнять пустые циклы ожидания до тех пор, пока писатель не покинет секцию.

Существует также вариант `write_tryseqlock()`, которая возвращает ненулевое значение, если она не смогла получить блокировку.

Если механизмы последовательной блокировки должны быть использованы в обработчике прерываний, то должны использоваться специальные (безопасные) версии API всех показанных выше вызовов (макросы):

```
unsigned int read_seqbegin_irqsave( seqlock_t* lock, unsigned long flags );
int read_seqretry_irqrestore( seqlock_t *lock, unsigned int seq, unsigned long flags );
void write_seqlock_irqsave( seqlock_t *lock, unsigned long flags );
void write_seqlock_irq( seqlock_t *lock );
```

```
void write_sequnlock_irqrestore( seqlock_t *lock, unsigned long flags );
void write_sequnlock_irq( seqlock_t *lock );
```

- где `flags` — просто заранее зарезервированная область сохранения IRQ флагов.

Мьютексы реального времени

Кроме обычных мьютексов (как бинарного подвида семафоров), в ядре создан новый интерфейс для мьютексов реального времени (`rt_mutex`). Это механизм достаточно позднего времени, его рассмотрение будем проводить на ядре:

```
$ uname -r
2.6.37.3
```

Структура мьютекса реального времени (`<linux/rtmutex.h>`), если исключить из рассмотрения её отладочную часть:

```
// RT Mutexes: blocking mutual exclusion locks with PI support
struct rt_mutex {                 //  The rt_mutex structure
   raw_spinlock_t      wait_lock; // spinlock to protect the structure
   struct plist_head   wait_list; // head to enqueue waiters in priority order
   struct task_struct *owner;     // the mutex owner
...
};
```

Характерным является присутствие поля `owner`, что характерно для любых вообще мьютексов POSIX (и отличает их от семафоров), это уже обсуждалось ранее. Там же определяется весь API для работы с этим примитивом, который не предлагает ничего необычного:

```
#define DEFINE_RT_MUTEX( mutexname )
void __rt_mutex_init( struct rt_mutex *lock,
                      const char *name ); // name используется в отладочной части
void rt_mutex_destroy( struct rt_mutex *lock );
void rt_mutex_lock( struct rt_mutex *lock );
int rt_mutex_trylock( struct rt_mutex *lock );
void rt_mutex_unlock( struct rt_mutex *lock );
```

Очень любопытно определяется признак захваченности мьютекса:

```
inline int rt_mutex_is_locked( struct rt_mutex *lock ) {
   return lock->owner != NULL;
}
```

Инверсия и наследование приоритетов

Мьютексы реального времени доступны только тогда, когда ядро собрано с параметром `CONFIG_RT_MUTEXES`, что проверяем так:

```
# cat /boot/config-2.6.32.9-70.fc12.i686.PAE | grep RT_MUTEX
CONFIG_RT_MUTEXES=y
# CONFIG_DEBUG_RT_MUTEXES is not set
# CONFIG_RT_MUTEX_TESTER is not set
```

В отличие от регулярных мьютексов, мьютексы реального времени обеспечивают наследование приоритетов (priority inheritance, PI), что является одним из нескольких немногих известных способов, препятствующих возникновению инверсии приоритетов (priority inversion). Если RT мьютекс захвачен процессом A, и его пытается захватить процесс B (более высокого приоритета), то:

- процесс B блокируется и помещается в очередь ожидающих освобождения процессов `wait_list` (в описании структуры `rt_mutex`);
- при необходимости, этот список ожидающих процессов переупорядочивается в порядке приоритетов ожидающих процессов;

– приоритет владельца мьютекса (текущего выполняющегося процесса) B повышается до приоритета ожидающего процесса A (максимального приоритета из ожидающих в очереди процессов);
– это и обеспечивает избежание потенциальной инверсии приоритетов.

Примечание: Эти действия затрагивают глубины управления процессами, для этого в `<linux/sched.h>` определяется специальный вызов :

```
void rt_mutex_setprio( struct task_struct *p, int prio );
```

И парный ему:

```
static inline int rt_mutex_getprio( struct task_struct *p ) {
   return p->normal_prio;
}
```

Из этой inline реализации хорошо видно, что в основной структуре описания процесса:

```
struct task_struct {
...
   int prio, static_prio, normal_prio;
...
}
```

- необходимо теперь иметь **несколько** полей приоритета, из которых поле `prio` является динамическим приоритетом, согласно которому и происходит диспетчеризация процессов в системе, а поле приоритета `normal_prio` остаётся неизменным, по значению которого происходит восстановление приоритета после освобождения мьютекса реального времени.

Множественное блокирование

В системах с большим количеством блокировок (ядро именно такая система), необходимость проведения более чем одной блокировки за раз не является необычной для кода. Если какие-то операции должны быть выполнены с использованием двух различных ресурсов, каждый из которых имеет свою собственную блокировку, часто нет альтернативы, кроме получения обоих блокировок. Однако, получение множества блокировок может быть крайне опасным:

```
DEFINE_SPINLOCK( lock1, lock2 );
...
spin_lock ( &lock1 ); /* 1-й фрагмент кода */
spin_lock ( &lock2 );
...
spin_lock ( &lock2 ); /* где-то в совсем другом месте кода... */
spin_lock ( &lock1 );
```

- такой образец кода, в конечном итоге, когда-то обречён на бесконечное блокирование (dead lock).

Если есть необходимость захвата нескольких блокировок, то единственной возможностью есть а). один тот же порядок захвата, б). и освобождения блокировок, в). порядок освобождения обратный порядку захвата, и г). так это должно выглядеть в каждом из фрагментов кода. В этом смысле предыдущий пример может быть переписан так:

```
spin_lock ( &lock1 ); /* так должно быть везде, где использованы lock1 и lock2 */
spin_lock ( &lock2 );
   /* ... здесь выполняется действие */
spin_unlock ( &lock2 );
spin_unlock ( &lock1 );
```

На практике обеспечить такую синхронность работы с блокировками в различных фрагментах кода крайне проблематично! (потому, что это может касаться фрагментов кода разных авторов).

Уровень блокирования

Нужно обратить внимание на такой отдельный вопрос как уровень блокирования. Очень часто, особенно когда это касается защиты критического участка кода, а не ограждения структуры данных, блокирование для синхронизации можно осуществлять на разных уровнях. Простейший пример этого мог бы быть фрагмент вида:

```
static DECLARE_MUTEX( sema );
down( &sema );
for( int i = 0; i < n; i++ ) {
   // здесь делается нечто монопольное за время T
}
up( &sema );
```

Этот же фрагмент может быть выполнен по-другому, что **функционально** эквивалентно:

```
static DECLARE_MUTEX( sema );
for( int i = 0; i < n; i++ ) {
   down( &sema );
   // здесь делается нечто монопольное за время T
   up( &sema );
}
```

Но во втором случае потенциальное блокирование любых других потоков, потребовавших семафора будет представляться как `n` раздельных интервалов длительностью `T`, а в первом как один сплошной интервал протяжённостью `n*T`. Но такой интервал ожидания часто — это время латентности системы. Реальные программные системы это сложные образования, где глубина вложенных компонент во много раз больше единицы, как в показанном условном примере. Общее правило состоит в том, что блокирование (синхронизация) должно осуществляться **на как можно более глубоком уровне**, даже если это потребует многократного увеличения числа обращений к примитиву синхронизации.

Примечание: На протяжении многих лет (фактически от рождения в 1991г.) в ядре Linux существовала так называемая глобальная блокировка ядра: если кто-либо в системе захватывал такую блокировку, то **все** последующие запросы глобальной блокировки, откуда бы они не исходили, блокировались. Это страшная вещь! И радикально избавиться от глобальной блокировки (до этого только постоянно сужалась область её применимости) с большим трудом удалось только к ядру 3.X к 2011г.

А теперь пример (архив `mlock.tgz`), который не столько ценен сам по себе, но живые эксперименты с которым позволяют очень тонко прочувствовать как происходит синхронизация потоков, и как эта синхронизация может быть сделана на самых различных уровнях:

mlock.c :

```
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/jiffies.h>
#include <linux/semaphore.h>

#include "../prefix.c"

static int num = 2;            // num - число рабочих потоков
module_param( num, int, 0 );
static int rep = 100;          // rep - число повторений (объём работы)
module_param( rep, int, 0 );
static int sync = -1;          // sync - уровень на котором синхронизация
module_param( sync, int, 0 );
static int max_level = 2;      // max_level - уровень глубины вложенности
module_param( max_level, int, 0 );

static DECLARE_MUTEX( sema );
static long locked = 0;
```

```
static long loop_func( int lvl ) {
   long n = 0;
   if( lvl == sync ) { down( &sema ); locked++; }
   if( 0 == lvl ) {
      const int tick = 1;
      msleep( tick );                      // выполняемая работа потока
      n = 1;
   }
   else {
      int i;
      for( i = 0; i < rep; i++ ) {
         n += loop_func( lvl - 1 );
      }
   }
   if( lvl == sync ) up( &sema );
   return n;
}

struct param {                             // параметр потока
   int    num;
   struct completion finished;
};

#define IDENT "mlock_thread_%d"
static int thread_func( void* data ) {
   long n = 0;
   struct param *parent = (struct param*)data;
   int num = parent->num - 1;              // порядковый номер потока (локальный!)
   struct task_struct *t1 = NULL;
   struct param parm;
   printk( "! %s is running\n", st( num ) );
   if( num > 0 ) {
      init_completion( &parm.finished );
      parm.num = num;
      t1 = kthread_run( thread_func, (void*)&parm, IDENT, num );
   }
   n = loop_func( max_level );             // рекурсивный вызов вложенных циклов
   if( t1 != NULL )
      wait_for_completion( &parm.finished );
   complete( &parent->finished );
   printk( "! %s do %ld units\n", st( num ), n );
   return 0;
}

static int test_mlock( void ) {
   struct task_struct *t1;
   struct param parm;
   unsigned j1 = jiffies, j2;
   if( sync > max_level ) sync = -1;   // без синхронизации
   printk( "! repeat %d times in %d levels; synch. in level %d\n",
           rep, max_level, sync );
   init_completion( &parm.finished );
   parm.num = num;
   t1 = kthread_run( thread_func, (void*)&parm, IDENT, num );
   wait_for_completion( &parm.finished );
   printk( "! %s is finished\n", st( num ) );
   j2 = jiffies - j1;
   printk( "!! working time was %d.%1d seconds, locked %ld times\n",
           j2 / HZ, ( j2 * 10 ) / HZ % 10, locked );
```

```
    return -1;
}

module_init( test_mlock );
MODULE_LICENSE( "GPL" );
```

Модуль достаточно сложный (не по коду, а по логике), поэтому некоторые краткие комментарии:

- включаемый файл определяет функцию `st()` для форматирования диагностики о потоке (с меткой времени `jiffies`), эту функцию мы уже видели ранее в обсуждении создания потоков;
- потоки (числом `num` — параметр запуска модуля) запускают друг друга последовательно, и завершаются в обратном порядке: каждый поток ожидает завершения им порождённого;
- «работа» потока состоит в циклическом (параметр: `rep` раз) выполнении рекурсивной функции `loop_func()`;
  `for( j1; ... )`
- рекурсия, вообще то говоря, крайне рискованная вещь в модулях ядра, из-за ограниченности и фиксированного размера стека ядра, но в данном случае а). это иллюстрационная задача (и она, попутно, показывает возможность рекурсии в коде ядра), б). функция сознательно имеет минимальной число локальных (стековых) переменных, в). причина, которая весит больше всех остальных вместе взятых — рекурсия позволяет создать структуру вложенных циклов **переменной** и произвольно большой глубины вложенности (параметр `max_level` модуля), вызов `loop_func( N )` эквивалентен:
  ```
  for( j2; ... )
     for( j3; ... )
     ...
        for( jN; ... )
  ```
- варьируя параметр модуля `sync`, можно заказывать, на какой глубине вложенных циклов потоки станут пытаться синхронизироваться захватом семафора `sema`: `sync=0` — на самом глубоком уровне имитации «работы» потока, `sync=1` — уровнем выше, ... `sync=max_level` — на максимально возможном верхнем уровне охватывающего цикла, `sync<0` или `sync>max_level` — вообще не синхронизироваться, не пытаться получить доступ к семафору `sema`;
- модуль выполнен в уже любимой нами манере исполнения как пользовательская задача, ничего не устанавливающая в ядре, но выполняющаяся в режиме защиты супервизора.

Ну, а дальше остаётся только многократно экспериментировать... Вот **экспоненциальная** степень роста объёма в зависимости от глубины вложенности:

```
$ sudo insmod mlock.ko rep=10 num=2 max_level=2 sync=-1
insmod: error inserting 'mlock.ko': -1 Operation not permitted
$ dmesg | tail -n 30 | grep !
! repeat 10 times in 2 levels; synch. in level -1
! 02094515 : kthread [05336:1] is running
! 02094515 : kthread [05337:0] is running
! 02094716 : kthread [05337:0] do 100 units
! 02094716 : kthread [05336:1] do 100 units
! 02094716 : kthread [05335:2] is finished
!! working time was 0.2 seconds, locked 0 times
$ sudo insmod mlock.ko rep=10 num=2 max_level=4 sync=-1
insmod: error inserting 'mlock.ko': -1 Operation not permitted
$ dmesg | tail -n 30 | grep !
! repeat 10 times in 4 levels; synch. in level -1
! 01915560 : kthread [05275:1] is running
! 01915560 : kthread [05276:0] is running
! 01935606 : kthread [05276:0] do 10000 units
! 01935608 : kthread [05275:1] do 10000 units
! 01935608 : kthread [05274:2] is finished
!! working time was 20.0 seconds, locked 0 times
```

А вот различия времени выполнения (в `num=5` раз!) в зависимости от синхронизации потоков или её отсутствия:

```
$ sudo insmod mlock.ko rep=10 num=5 max_level=3 sync=-1
```

```
insmod: error inserting 'mlock.ko': -1 Operation not permitted
$ dmesg | tail -n 30 | grep '!!'
!! working time was 2.0 seconds, locked 0 times
$ sudo insmod mlock.ko rep=10 num=5 max_level=3 sync=0
insmod: error inserting 'mlock.ko': -1 Operation not permitted
$ dmesg | tail -n 30 | grep '!!'
!! working time was 10.0 seconds, locked 5000 times
$ sudo insmod mlock.ko rep=10 num=5 max_level=3 sync=1
insmod: error inserting 'mlock.ko': -1 Operation not permitted
$ dmesg | tail -n 30 | grep '!!'
!! working time was 10.0 seconds, locked 500 times
$ sudo insmod mlock.ko rep=10 num=5 max_level=3 sync=2
insmod: error inserting 'mlock.ko': -1 Operation not permitted
$ dmesg | tail -n 30 | grep '!!'
!! working time was 10.0 seconds, locked 50 times
$ sudo insmod mlock.ko rep=10 num=5 max_level=3 sync=3
insmod: error inserting 'mlock.ko': -1 Operation not permitted
$ dmesg | tail -n 30 | grep '!!'
!! working time was 10.0 seconds, locked 5 times
```

Очень показательно в выводе число обращений (locked) к семафору: на одном и том же периоде выполнения число обращений изменяется на 3 **порядка**, во столько же раз «гуляет» продолжительность единичного акта захвата примитива синхронизации, то, с чего началось обсуждение этого раздела.

Этот пример не показал зависимости общего итогового времени выполнения от глубины уровня синхронизации, это связано с симметричностью уровней вложенности, и нежеланием ещё более усложнять код примера. В реальных задачах, тем не менее, соблюдается общее правило: чем выше выбран уровень синхронизация — тем больше затраты времени на выполнение.

Предписания порядка выполнения

Механизмы, предписывающие порядок выполнения кода, к синхронизирующим механизмам относятся весьма условно, они не являются непосредственно синхронизирующими механизмами, но рассматриваются всегда вместе с ними (по принципу: надо же их где-то рассматривать?). Рассматривают их совместно с синхронизациями потому, что их роднит **единственное** сходство: и те и другие могут влиять на порядок выполнения операторов, и изменять его в «плавном» последовательном выполнении операторов кода.

Аннотация ветвлений

Одним из таких механизмов являются определённые в `<linux/compiler.h>` макросы `likely()` и `unlikely()`, которые иногда называют аннотацией ветвлений, например:

```
if( unlikely(...) ) {
   /* сделать нечто редкостное */
};
```

Или:

```
if( likely(...) ) {
   /* обычное прохождение вычислений */
}
else {
   /* что-то нетрадиционное */
};
```

Те, кто помнит в минимальном объёме специфику выполнения процессорной инструкции `jmp`, вспомнит, что при её выполнении происходит разгрузка (и перезагрузка) конвейера уже частично декодированных последующих команд. Кроме того, в игру может включиться и кеш памяти, который может потребовать перезагрузки в дальнюю `jmp` область (а это разница в скорости обычно в 2-4 **раза**). А если это так, и если во всяком ветвлении (условном переходе) одна из ветвей обязательно должна выполнять `jmp`, то можно дать

указание компилятору компилировать код так, чтобы веткой с `jmp` оказалась ветка с наименьшей вероятностью (частотой) выполнения. Таким образом, аннотацией ветвлений нужны для повышения производительности выполнения кода (иногда заметно ощутимой), никаким другим образом на выполнение они не влияют. Кроме того, такие предписания б). делают код более читабельным, в). недопустимы (не определены) в пространстве пользовательского кода (**только в ядре**).

Примечание: Подобные оптимизации становятся актуальными с появлением в процессорах конвейерных вычислений с предсказыванием. На других платформах, отличных от Intel x86, они могут быть на сегодня и не столь ощутимыми (это нужно уточнять детальным просмотром архитектуры перед началом разработки).

Барьеры

Другим примером предписаний порядка выполнения являются барьеры в памяти, препятствующие в процессе оптимизации переносу операций чтения и записи через объявленный барьер. Например, при записи фрагмента кода:

```
a = 1;
b = 2;
```

- порядок выполнения операция, вообще то говоря, непредсказуем, причём последовательность (во времени) выполнения операций может изменить а). компилятор из соображений оптимизации, б). процессор (периода выполнения) из соображений аппаратной оптимизации работы с шиной памяти. В этом случае это совершенно нормально, более того, даже запись операторов:

```
a = 1;
b = a + 1;
```

- будет гарантировать отсутствие перестановок в процессе оптимизации, так как компилятор «видит» операции в едином контексте (фрагменте кода). Но в других случаях, когда операции производятся из различных мест кода нужно гарантировать, что они не будут перенесены через определённые барьеры. Операции (макросы) с барьерами объявлены в `</asm-generic/system.h>`, на сегодня все они (`rmb(), wmb(), mb(), ...`) определены одинаково:

```
#define   mb()  asm volatile ("": : :"memory")
```

Все они препятствуют выполнению операций с памятью после такого вызова до завершения всех операций, записанных до вызова.

Ещё один макрос объявлен в `<linux/compiler.h>`, он препятствует компилятору при оптимизации переставлять между собой операторы до этого вызова и после него:

```
void barrier( void );
```

Задачи

1. Проведите проверку максимальных блоков памяти, которые в вашей системе могут быть выделены механизмами `kmalloc()` и `vmalloc()`, соответственно. Сделайте это с более высокой точностью, чем показано приближённо в тексте (сделайте модуль, который определяет эти значения с высокой точностью).

2. Оцените порядок интервала времени, на котором происходит **переполнение** счётчика системных тиков ядра `jiffies`.

3. Для начала просто "сосчитайте" (динамически, в ядре) сколько процессоров присутствуют в вашей текущей системе (архитектуре).

4. Вам предстоит выполнить некоторые ординарные действия в ядре, но параллельно, средствами потоков ядра. Создайте, как минимум, 4 разных способа формирования выполнения в отдельном потоке:

4.1. Пользуясь старым вызовом `kernel_thread()`. Для этого способа определите версию ядра, после которой этот способ не будет работать, и объясните причину почему.

4.2. Пользуясь новым вызовом `kthread_run()` (или `kthread_create()`).

4.3. Используя очередь рабочих процедур: `queue_work()`.

4.4. Используя планирование тасклета `tasklet_schedule()`.

5. Синхронизации. Делаем модуль, который запускает N потоков ядра (N сделать параметром загрузки). Потоки отрабатывают (делают инкремент своих счётчиков циклов) и прерываются (синхронно завершаются) из запускающего потока по истечению интервала работы T.

   5.1. Сделать это завершая рабочие потоки поочерёдным `kthread_stop()` из вызывающего потока, и анализом `kthread_should_stop()` в циклах рабочих потоков.

   5.2. В предыдущем варианте `kthread_stop()` обязан ожидать завершения очередного потока и только после этого приступать к следующему, из за этого их продолжительность будет незначительно, но отличаться. Сделайте синхронизацию на **едином** для всех потоков **битовом** атомарном флаге.

   5.3. То же, что и в предыдущем случае, но на **арифметическом** атомарном флаге.

6. Спин-блокировки. Возьмите любой вариант из предыдущих задач, и добавьте внутрь цикла потоковой функции захват и освобождение (в начале и конце тела цикла) активной спин-блокировки, **общей** для всех рабочих потоков. Наблюдайте и покажите:

   - Радикальное падение производительности от столь незначительного дополнения;

   - Полную остановку системы при N+1 рабочих потоках, где N — число процессоров в вашей системе;

7. Смените в предыдущей задаче спин-блокировку на, например, **мютекс реального времени** и убедитесь, что многопоточное приложение и в этом случае продолжает выполняться как однопоточное, последовательно. Но при этом не могут возникнуть эффекты деградации системы, даже при большом числе параллельных потоков.

8.

12. Расширенные возможности программирования

«В правильном вопросе всегда уже скрыт ответ,
и в правильном поиске уже угадывается искомое»
Протоиерей Андрей Ткачев

Существует великое многообразие возможностей для программиста, пишущего в адресном пространстве ядра. На практике не все они востребованы в равной мере широко, некоторые используются заметно реже, чем типовые средства, разбираемые до сих пор. Но все эти более изощрённые варианты весьма значимы для детального понимания происходящего в ядре, а поэтому хотя бы некоторые из них должны быть коротко рассмотрены.

Примечание: Многие из таких возможностей (реализующие действия, **подобные** аналогичным таким же операциям в пространстве пользователя — в более привычном контексте), в литературе и обсуждениях по ядру относятся к общей группе API под названием «хелперы» (или «хелперы пространства пользователя»), где информацию о них и следует искать.

Большинство обсуждаемых далее возможностей редко и скупо описываются в публикациях и упоминаются в обсуждениях. Но информацию о реализации и использовании подобных возможностей вы всегда можете получить путём экспериментирования с тестирующим кодом над «живой» операционной системой.

Операции с файлами данных

Операции с данными в именованных файлах (разных: регулярных файлах, FIFO и др.) не относятся к тем возможностям, которыми код ядра (модуля) должен **активно** пользоваться, для того не видно особых оснований[23] (так же, например, как и с операциями с абсолютным хронологическим временем). Но, во-первых, такие операции вполне возможны, а во-вторых, существует, как минимум, одна ситуация, когда такая возможность насущно необходима: это чтение конфигурационных данных модуля (при запуске) из его конфигурационных файлов. Как будет показано, такие возможности не только осуществимы из кода ядра, они, более того, осуществимы несколькими альтернативными способами.

Смотрим это на примерах (архив `file.tgz`):

mod_file.c :

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

static char* file = NULL;
module_param( file, charp, 0 );

#define BUF_LEN 255
#define DEFNAME "/etc/yumex.profiles.conf";
static char buff[ BUF_LEN + 1 ] = DEFNAME;

static int __init kread_init( void ) {
    struct file *f;
    size_t n;
    if( file != NULL ) strcpy( buff, file );
    printk( "*** openning file: %s\n", buff );
    f = filp_open( buff, O_RDONLY, 0 );
```

23 Попытка модуля ядра активно работать с файлами данных уже должна настораживать, как возможный признак похой архитектурной проработки подсистемы.

```
    if( IS_ERR( f ) ) {
        printk( "*** file open failed: %s\n", buff );
        return -ENOENT;
    }

    n = kernel_read( f, 0, buff, BUF_LEN );
    if( n ) {
        printk( "*** read first %d bytes:\n", n );
        buff[ n ] = '\0';
        printk( "%s\n", buff );
    } else {
        printk( "*** kernel_read failed\n" );
        return -EIO;
    }

    printk( "*** close file\n" );
    filp_close( f, NULL );
    return -EPERM;
}

module_init( kread_init );
MODULE_LICENSE( "GPL" );
```

Теперь смотрим как это работает (используемый в примере текстовый файл `./xxx` был подготовлен заранее, и содержит несколько строк текста, а файл с именем `./yyy` отсутствует, и указан в примере как недозволенное имя файла):

```
$ sudo insmod mod_file.ko file=./xxx
insmod: error inserting 'mod_file1.ko': -1 Operation not permitted
$ dmesg | tail -n100 | grep '^\*'
*** openning file: ./xxx
*** read first 39 bytes:
*1 .........
*2 .........
*3 .........
*** close file
$ cat ./xxx
*1 .........
*2 .........
*3 .........
```

Пытаемся читать несуществующий файл, обратите внимание, как изменился код ошибки загрузки модуля:

```
$ sudo insmod mod_file.ko file=./yyy
insmod: error inserting 'mod_file1.ko': -1 Unknown symbol in module
$ dmesg | tail -n20 | grep '^\*'
*** openning file: ./yyy
*** file open failed: ./yyy
```

А вот файл из каталога конфигураций `/etc`, это уже ближе к реальным потребностям:

```
$ sudo insmod mod_file.ko file=/etc/yumex.profiles.conf
insmod: error inserting 'mod_file.ko': -1 Operation not permitted
$ cat /etc/yumex.profiles.conf
[main]
lastprofile = yum-enabled$
$ dmesg | tail -n 40
...
**** openning file: /etc/yumex.profiles.conf
**** read first 32 bytes:
[main]
lastprofile = yum-enabled
```

```
*** close file
```

Предыдущий пример использовал специальный вызов ядра `kernel_read()`, предназначенный только для такой цели. Но в следующий пример использует совершенно другой набор API ядра: вызовы имён, экспортируемых **виртуальной** файловой системой (VFS, вызовы вида `vfs_*()`):

mod_vfs.c :

```
include <linux/module.h>
#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/uaccess.h>

static char* file = NULL;
module_param( file, charp, 0 );

#define BUF_LEN 255
#define DEFNAME "/etc/yumex.profiles.conf";
static char buff[ BUF_LEN + 1 ] = DEFNAME;

static int __init kread_init( void ) {
    struct file *f;
    size_t n;
    long l;
    loff_t file_offset = 0;

    mm_segment_t fs = get_fs();
    set_fs( get_ds() );

    if( file != NULL ) strcpy( buff, file );
    printk( "*** openning file: %s\n", buff );
    f = filp_open( buff, O_RDONLY, 0 );

    if( IS_ERR( f ) ) {
        printk( "*** file open failed: %s\n", buff );
        l = -ENOENT;
        goto fail_oupen;
    }

    l = vfs_llseek( f, 0L, 2 ); // 2 means SEEK_END
    if( l <= 0 ) {
        printk( "*** failed to lseek %s\n", buff );
        l = -EINVAL;
        goto failure;
    }
    printk( "*** file size = %d bytes\n", (int)l );

    vfs_llseek( f, 0L, 0 );    // 0 means SEEK_SET
    if( ( n = vfs_read( f, buff, l, &file_offset ) ) != l ) {
        printk( "*** failed to read\n" );
        l = -EIO;
        goto failure;
    }
    buff[ n ] = '\0';
    printk( "%s\n", buff );
    printk( KERN_ALERT "**** close file\n" );
    l = -EPERM;
failure:
    filp_close( f, NULL );
fail_oupen:
```

```
    set_fs( fs );
    return (int)l;
}

module_init( kread_init );
MODULE_LICENSE( "GPL" );
```

Этот вариант сложнее в использовании, но он более гибкий в своих возможностях. Гибкость состоит в возможности использования всего набора функций файловых операция (таких, как показанная в примере `vfs_llseek()`), а не только узкого подмножества вызовов. А сложность состоит в том, что операции виртуальной системы «заточены» на работу с буферами в пространстве **пользователя**, и проверяют принадлежность адресов-параметров на принадлежность этому пространству. Для выполнения тех же операций с данными пространства ядра, нужно снять эту проверку на время операции и восстановить её после. Что и достигается использованием макровызовов: `get_fs()`, `set_fs()`, `get_ds()`. Но за этим надо тщательно следить, иначе операция завершится с кодом: `Bad address`.

И снова убеждаемся в работоспособности модуля над тем же файлом данных:

```
$ sudo insmod mod_vfs.ko file=./xxx
insmod: error inserting 'mod_vfs.ko': -1 Operation not permitted
$ dmesg | tail -n30 | grep '^\*'
*** openning file: ./xxx
*** file size = 39 bytes
*1 .........
*2 .........
*3 .........
**** close file
```

Запись в файл из модуля, возможно, ещё более редко востребованная операция, чем чтение. Но и она, во-первых, совершенно возможна, и во-вторых, бывает полезна, например, для протоколирования событий и, особенно, в целях отладки (возможности которой крайне сужены в случае ядра). Следующий пример демонстрирует такую возможность:

mdw.c :

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

static char* log = NULL;
module_param( log, charp, 0 );

#define BUF_LEN 255
#define DEFLOG "./module.log"
#define TEXT "..............\n"

static struct file *f;

static int __init init( void ) {
   ssize_t n = 0;
   loff_t offset = 0;
   mm_segment_t fs;
   char buff[ BUF_LEN + 1 ] = DEFLOG;
   if( log != NULL ) strcpy( buff, log );
   f = filp_open( buff,
                  O_CREAT | O_RDWR | O_TRUNC,
                  S_IRUSR | S_IWUSR );
   if( IS_ERR( f ) ) {
      printk( "! file open failed: %s\n", buff );
      return -ENOENT;
   }
```

```
    printk( "! file open %s\n", buff );
    fs = get_fs();
    set_fs( get_ds() );
    strcpy( buff, TEXT );
    if( ( n = vfs_write( f, buff, strlen( buff ), &offset ) ) != strlen( buff ) ) {
         printk( "! failed to write: %d\n", n );
         return -EIO;
    }
    printk( "! write %d bytes\n", n );
    printk( "! %s", buff );
    set_fs( fs );
    filp_close( f, NULL );
    printk( "! file close\n" );
    return -1;
}
module_init( init );
MODULE_LICENSE( "GPL" );
```

Вот как выглядит работа этого кода:

```
$ sudo insmod mdw.ko
insmod: error inserting 'mdw.ko': -1 Operation not permitted
$ dmesg | tail -n40 | grep !
! file open ./module.log
! write 16 bytes
! ...............
! file close
$ ls -l *.log
-rw------- 1 root root 16 Дек 31 21:23 module.log
 $ sudo cat module.log
...............
```

Обратите внимание, создаваемый и записываемый файл журнала `module.log` создаётся от имени владельца `root` (а код модуля и исполняется `insmod` только от этого имени), для последующей работы с таким файлом протокола вам, возможно, придётся поменять его владельца.

Запуск новых процессов из ядра

Можно ли запустить новый пользовательский процесс из кода модуля? Интуитивный ответ: наверняка да, поскольку **все и каждый** процессы, выполняющиеся в системе, запущены именно из кода ядра.

Новые процессы пользовательского пространства могут запускаться кодом ядра, по форме аналогично тому, как запускаются они и в пользовательском коде вызовами группы `exec*()`. Но по содержанию это действие имеет несколько другой смысл. Процессы из пользовательского кода создаются в два шага. Первоначально выполняется `fork()`, которым создаётся **новое адресное пространство**, являющееся абсолютным дубликатом порождающего процесса. Это адресное пространство позже и становится пространством нового процесса, когда в нём производится вызов одной из функций семейства `exec()`. В пространстве ядра это должно происходить по-другому, здесь нельзя создать дубликат ядерного пространства. Для выполнения запуска нового процесса здесь предоставляется специальный вызов `call_usermodehelper()`.

Простейший пример демонстрирует возможность порождения новых процессов в системе по инициативе ядра (архив `exec.tgz`):

`mod_exec.c :`

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

```
static char *str;
module_param( str, charp, S_IRUGO );

int __init exec_init( void ) {
   int rc;
   char *argv[] = { "wall", "\nthis is wall message ", NULL },
        *envp[] = { NULL },
        msg[ 80 ];
   if( str ) {
      sprintf( msg, "\n%s", str );
      argv[ 1 ] = msg;
   }
   rc = call_usermodehelper( "/usr/bin/wall", argv, envp, 0 );
   if( rc ) {
      printk( KERN_INFO "failed to execute : %s\n", argv[ 0 ] );
      return rc;
   }
   printk( KERN_INFO "execute : %s %s\n", argv[ 0 ], argv[ 1 ] );
   msleep( 100 );
   return -1;
}

module_init( exec_init );
MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_VERSION( "2.1" );
```

Вызов `call_usermodehelper()` получает параметры точно так же, как **системный** вызов пользовательского пространства `execve()` (через который выполняются все прочие **библиотечные** вызовы семейства `exec*()`), детали смотрите в справочной странице :

```
$ man 2 execve
```

Вот как срабатывает созданный модуль при нормальном течении исполнения:

```
$ sudo insmod mod_exec.ko

Broadcast message from root@notebook.localdomain (Mon Jan 30 18:13:10 2012):
this is wall message
insmod: error inserting 'mod_exec.ko': -1 Operation not permitted
```

Или вот так, если при загрузке модуля указан параметр:

```
$ sudo insmod mod_exec.ko str="new_string"

Broadcast message from root@notebook.localdomain (Mon Jan 30 18:22:59 2012):
new_string
insmod: error inserting 'mod_exec.ko': -1 Operation not permitted
$ dmesg | tail -n30 | grep -v ^audit
execute : wall
new_string
```

Модуль успешно загружается, видно нормальный запуск автономного пользовательского приложения, выводя **широковещательное** сообщение на все терминалы системы. Если приложение не может быть запущено, чаще всего из-за неправильно указанного **полного** путевого имени файла запускаемой программы, код завершения загрузки модуля будет другим. Это существенно важно, учитывая, что модули исполняются в практически «глухо-немом» режиме:

```
$ sudo insmod mod_exec.ko
```

```
insmod: error inserting 'mod_exec.ko': -1 Unknown symbol in module
$ dmesg | tail -n30 | grep -v ^audit
failed to execute : /bin/wall
```

Особенность и ограниченность метода запуска приложения вызовом `call_usermodehelper()` из ядра состоит в том, что процесс запускается **без управляющего терминала** и с нестандартным для него окружением! Это легко видеть, если в качестве пользовательского процесса использовать утилиту `echo`, а строки запуска изменить (в архиве для сравнения содержится такой модуль `mod_execho.c`):

```
char *argv[] = { "/bin/echo", "this is wall message", NULL };
...
rc = call_usermodehelper( "/bin/echo", argv, envp, 0 );
```

Здесь результатом будет:

```
$ sudo insmod mod_execho.ko
insmod: error inserting 'mod_execho.ko': -1 Operation not permitted
$ dmesg | tail -n30 | grep -v ^audit
execute : /bin/echo echo message
```

Здесь имеет место **нормальное** выполнение утилиты `echo`, но мы не увидим результата её работы (вывода на терминал), поскольку у неё просто нет этого управляющего терминала.

Всю основную работу в модуле по созданию и запуску процесса, как легко видеть, выполняет вызов `call_usermodehelper()` (`<linux/kmod.h>`), детали которого понятны из его прототипа:

```
static inline int call_usermodehelper( char *path, char **argv, char **envp, enum umh_wait wait );
enum umh_wait {
   UMH_NO_WAIT = -1,   /* don't wait at all */
   UMH_WAIT_EXEC = 0,  /* wait for the exec, but not the process */
   UMH_WAIT_PROC = 1,  /* wait for the process to complete */
};
```

Сигналы UNIX

Сигналы являются одной из не столь уж многих **концептуальных** основ операционных систем семейства UNIX, придающих им характерный вид. Сигналы UNIX (именно такое полное название применяется для них в литературе) часто могут быть не видны «на поверхности» пользователю, но играют значительную роль внутри системы — достаточно обратить внимание на то, что без помощи сигналов мы не смогли бы принудительно завершить ни один процесс в системе (нажимая `^C` или выполняя команды `kill, killall`). Детальное углубление в теорию сигналов увело бы нас слишком далеко от наших намерений, но в сферу дальнейшего рассмотрения будет попадать только статус сигналов UNIX в ядре.

Да, речь идёт именно о возможности отправки сигналов процессу пространства пользователя, или потоку пространства ядра. То, что мы привычно делаем в пользовательском пространстве командой `kill`. Точно так же, как и запуск нового процесса, по аналогии с пользовательским пространством, можно посылать из ядра сигналы UNIX как пользовательским процессам, так и другим потокам пространства ядра (вспоминаем, что ядро Linux многопоточное и, более того, разные потоки могут выполняться **одновременно** различными процессорами в многоядерной архитектуре). Возможность сигнальных взаимодействий должна быть понятна просто из рассмотрения того факта, что большинство вызовов API ядра, которые рассматривались на протяжении всего протяжённого предыдущего рассмотрения, таких, которые переводят текущее выполнение в блокированное состояние, имеют **две альтернативные** формы: безусловную, ожидающую наступления финального условия, и форму, допускающую прерывание ожидания получением **сигнала**. Остаётся вопрос: **как** реализовать такую возможность?

Для уяснения возможностей использования сигналов из ядра (и в ядре) я воспользовался несколько видоизменённым (архив `signal.tgz`) проектом из [6]. Идея этого в меру громоздкого, 3-х компонентного теста проста:

– пользовательское **приложение** `sigreq` («мишень» на которую направляются сигналы), которое регистрирует получаемые сигналы;

– **модуль** ядра `ioctl_signal.ko`, которому можно «заказать»: какому пользовательскому процессу (любому, по PID) отсылать сигнал (это и будет «мишень», в качестве целеуказания мы будем указывать `sigreq`);

– диалоговое пользовательское **приложение** `ioctl`, который указывает модулю ядра `ioctl_signal.ko`: какой именно сигнал отсылать (по номеру) и какому процессу «мишени» (PID) ; процесс `ioctl` будет передавать команды для `ioctl_signal.ko` посредством вызовов `ioctl()`, что уже рассматривалось ранее при рассмотрении драйверов символьных устройств.

Общие определения, необходимые для команд `ioctl()`, вынесены в отдельный файл `ioctl.h`:

ioctl.h :

```
#define MYIOC_TYPE 'k'
#define MYIOC_SETPID   _IO(MYIOC_TYPE,1)
#define MYIOC_SETSIG   _IO(MYIOC_TYPE,2)
#define MYIOC_SENDSIG  _IO(MYIOC_TYPE,3)
#define SIGDEFAULT SIGKILL
```

Команды `ioctl()`, которые отрабатываются модулем: `MYIOC_SETPID` - установить PID процесса, которому будет направляться сигнал; `MYIOC_SETSIG` - установить номер отсылаемого сигнала; `MYIOC_SENDSIG` — команда отправить сигнал.

Собственно код модуля:

ioctl_signal.c :

```
#include <linux/module.h>
#include "ioctl.h"
#include "lab_miscdev.h"

static int sig_pid = 0;
static struct task_struct *sig_tsk = NULL;
static int sig_tosend = SIGDEFAULT;

static inline long mycdrv_unlocked_ioctl( struct file *fp, unsigned int cmd, unsigned long arg ) {
   int retval;
   switch( cmd ) {
      case MYIOC_SETPID:
         sig_pid = (int)arg;
         printk( KERN_INFO "Setting pid to send signals to, sigpid = %d\n", sig_pid );
         /* sig_tsk = find_task_by_vpid (sig_pid); */
         sig_tsk = pid_task( find_vpid( sig_pid ), PIDTYPE_PID );
         break;
      case MYIOC_SETSIG:
         sig_tosend = (int)arg;
         printk( KERN_INFO "Setting signal to send as: %d \n", sig_tosend );
         break;
      case MYIOC_SENDSIG:
         if( !sig_tsk ) {
            printk( KERN_INFO "You haven't set the pid; using current\n" );
            sig_tsk = current;
            sig_pid = (int)current->pid;
         }
         printk( KERN_INFO "Sending signal %d to process ID %d\n", sig_tosend, sig_pid );
         retval = send_sig( sig_tosend, sig_tsk, 0 );
         printk( KERN_INFO "retval = %d\n", retval );
         break;
      default:
```

```
            printk( KERN_INFO " got invalid case, CMD=%d\n", cmd );
            return -EINVAL;
      }
      return 0;
   }

   static const struct file_operations mycdrv_fops = {
      .owner = THIS_MODULE,
      .unlocked_ioctl = mycdrv_unlocked_ioctl,
      .open = mycdrv_generic_open,
      .release = mycdrv_generic_release
   };

   module_init( my_generic_init );
   module_exit( my_generic_exit );
   MODULE_AUTHOR("Jerry Cooperstein");
   MODULE_LICENSE("GPL v2");
```

В этом файле содержится интересующий нас обработчик функций `ioctl()`, все остальные операции модуля (создание символьного устройства `/dev/mycdrv`, `open()`, `close()`, ...) - отнесены во включаемый файл `lab_miscdev.h`, общий для многих примеров, и не представляющий интереса — всё это было подробно рассмотрено ранее, при рассмотрении операций символьного устройства.

Пока остановим внимание на группе функций, находящих запись процесса (`struct task_struct` — в циклической очереди процессов ядра) по его PID, что близко смыкается с задачей запуска процесса, рассматриваемой выше — для этого используется инструментарий:

```
#include <linux/sched.h>
struct task_struct *find_task_by_vpid( pid_t nr );
#include <linux/pid.h>
struct pid *find_vpid( int nr );
struct task_struct *pid_task( struct pid *pid, enum pid_type );
enum pid_type {
   PIDTYPE_PID,
   PIDTYPE_PGID,
   PIDTYPE_SID,
   PIDTYPE_MAX
};
```

Тестовая задача, выполняющая в диалоге с пользователем последовательность команда `ioctl()` над модулем, последовательно: установку PID процесса, установку номера сигнала, отправку команды на посылку сигнала:

ioctl.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <signal.h>
#include "ioctl.h"

static void sig_handler( int signo ) {
   printf( "---> signal %d\n", signo );
}

int main( int argc, char *argv[] ) {
   int fd, rc;
   unsigned long pid, sig;
   char *nodename = "/dev/mycdrv";
   pid = getpid();
```

```
   sig = SIGDEFAULT;
   if( argc > 1 ) sig = atoi( argv[ 1 ] );
   if( argc > 2 ) pid = atoi( argv[ 2 ] );
   if( argc > 3 ) nodename = argv[ 3 ];
   if( SIG_ERR == signal( sig, sig_handler ) )
      printf( "set signal handler error\n" );
   /* open the device node */
   fd = open( nodename, O_RDWR );
   printf( "I opened the device node, file descriptor = %d\n", fd );
   /* send the IOCTL to set the PID */
   rc = ioctl( fd, MYIOC_SETPID, pid );
   printf("rc from ioctl setting pid is = %d\n", rc );
   /* send the IOCTL to set the signal */
   rc = ioctl( fd, MYIOC_SETSIG, sig );
   printf("rc from ioctl setting signal is = %d\n", rc );
   /* send the IOCTL to send the signal */
   rc = ioctl( fd, MYIOC_SENDSIG, "anything" );
   printf("rc from ioctl sending signal is = %d\n", rc );
   /* ok go home */
   close( fd );
   printf( "FINISHED, TERMINATING NORMALLY\n");
   exit( 0 );
}
```

Тестовая задача, являющаяся оконечным приёмником-регистраторов отправляемых сигналов («мишень»):

sigreg.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include "ioctl.h"

static void sig_handler( int signo ) {
   printf( "---> signal %d\n", signo );
}

int main( int argc, char *argv[] ) {
   unsigned long sig = SIGDEFAULT;
   printf( "my own PID is %d\n", getpid() );..
   sig = SIGDEFAULT;
   if( argc > 1 ) sig = atoi( argv[ 1 ] );
   if( SIG_ERR == signal( sig, sig_handler ) )
      printf( "set signal handler error\n" );
   while( 1 ) pause();
   exit( 0 );
}
```

В этом приложении (как и в предыдущем) для установки обработчика сигнала используется старая, так называемая «ненадёжная модель» обработки сигналов, использованием вызова `signal()`, но в данном случае это никак не влияет на достоверность получаемых результатов.

Начнём проверку с конца: просто с отправки процессу регистратору сигнала консольной командой `kill`, но прежде нужно уточниться с доступным в реализации нашей операционной системы набором сигналов (этот список для разных операционных систем может не очень значительно, но отличаться):

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
```

```
21) SIGTTIN      22) SIGTTOU      23) SIGURG       24) SIGXCPU      25) SIGXFSZ
26) SIGVTALRM    27) SIGPROF      28) SIGWINCH     29) SIGIO        30) SIGPWR
31) SIGSYS       34) SIGRTMIN     35) SIGRTMIN+1   36) SIGRTMIN+2   37) SIGRTMIN+3
38) SIGRTMIN+4   39) SIGRTMIN+5   40) SIGRTMIN+6   41) SIGRTMIN+7   42) SIGRTMIN+8
43) SIGRTMIN+9   44) SIGRTMIN+10  45) SIGRTMIN+11  46) SIGRTMIN+12  47) SIGRTMIN+13
48) SIGRTMIN+14  49) SIGRTMIN+15  50) SIGRTMAX-14  51) SIGRTMAX-13  52) SIGRTMAX-12
53) SIGRTMAX-11  54) SIGRTMAX-10  55) SIGRTMAX-9   56) SIGRTMAX-8   57) SIGRTMAX-7
58) SIGRTMAX-6   59) SIGRTMAX-5   60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
```

Для проверок функционирования может быть использован (почти) любой из этого набора сигналов UNIX, выберем безобидный (не имеющий специального предназначения) сигнал `SIGUSR1` (сигнал номер 10):

```
$ ./sigreq 10
my own PID is 10737
---> signal 10
$ kill -n 10 10737
```

Вот как отреагировал процесс регистратор на получение сигнала с терминала (в порядке теста). А теперь выполним весь комплекс: процесс `ioctl` последовательностью вызовов `ioctl()` заставляет загруженный модуль ядра `ioctl_signal` отправить указанный сигнал процессу `sigreq` :

```
$ sudo insmod ioctl_signal.ko
$ lsmod | head -n2
Module                  Size  Used by
lab3_ioctl_signal       2053  0.
$ dmesg | tail -n2
Succeeded in registering character device mycdrv
$ cat /sys/devices/virtual/misc/mycdrv/dev
10:56
$ ls -l /dev | grep my
crw-rw----  1 root root     10,  56 Май  6 17:15 mycdrv
$ ./ioctl 10 11684
I opened the device node, file descriptor = 3
rc from ioctl setting pid is = 0
rc from ioctl setting signal is = 0
rc from ioctl sending signal is = 0
FINISHED, TERMINATING NORMALLY
$ dmesg | tail -n14
Succeeded in registering character device mycdrv
 attempting to open device: mycdrv:
  MAJOR number = 10, MINOR number = 56
   successfully open  device: mycdrv:
I have been opened  1 times since being loaded
ref=1
Setting pid to send signals to, sigpid = 11684
Setting signal to send as: 10.
Sending signal 10 to process ID 11684
retval = 0
closing character device: mycdrv:
$ ./sigreq 10
my own PID is 11684
---> signal 10
^C
```

Отправку сигнала в этой реализации осуществляет вызов `send_sig()`, он, и ещё большая группа подобных функций, связанных с отправкой сигналов, определены в `<linux/sched.h>`, некоторые из которых:

```
int send_sig_info( int signal, struct siginfo *info, struct task_struct *task );
int send_sig( int signal, struct task_struct *task, int priv );
int kill_pid_info( int signal, struct siginfo *info, struct pid *pid );
int kill_pgrp( struct pid *pid, int signal, int priv );
int kill_pid( struct pid *pid, int signal, int priv );
```

```
int kill_proc_info( int signal, struct siginfo *info, pid_t pid );
```

Описания достаточно сложной структуры `siginfo` (это так называемая схема сигналов реального времени) включено из заголовочных файлов пространства пользователя (`/usr/include/asm-generic/siginfo.h`):

```
typedef struct siginfo {
   int si_signo;
   int si_errno;
   int si_code;
...
}
```

Тема сигналов чрезвычайно важная — на них основаны все механизмы асинхронных уведомлений, например, работа пользовательских API `select()` и `pool()`, или асинхронных операций ввода-вывода. Но тема сигналов и одна из самых слабо освещённых в литературе[24].

Вокруг экспорта символов ядра

Теперь углубимся детально в самое путанное понятие из области ядра - **экспорт символов** ядра. Для того, чтобы имя пространства ядра было доступно для связывания в вашем модуле, для этого имени должны выполняться два условия: а). имя должно иметь глобальную область видимости (в вашем модуле такие имена не должны объявляться `static`) и б). имя должно быть явно объявлено **экспортируемым**, должно быть явно записано параметром макроса `EXPORT_SYMBOL` (или `EXPORT_SYMBOL_GPL`, что далеко не одно и то же). Мы уже встречались с понятием экспортирования в начале нашего экскурса в технику модулей ядра, но только сейчас сможем разобраться с нею детально (наработав определённый багаж для создания тестовых модулей). Выберем для сравнительного изучения два сходных системных вызова, `sys_open()` и `sys_close()` :

```
$ cat /proc/kallsyms | grep ' T ' | grep sys_open
c04deb28 T do_sys_open
c04dec0c T sys_openat
c04dec35 T sys_open
$ cat /proc/kallsyms | grep ' T ' | grep sys_close
c04dea99 T sys_close
```

Оба имени `sys_open` и `sys_close` известны в таблице символов ядра как глобальные имена в секции кода (T). Сделаем (архив `export.tgz`) простейший модуль ядра:

md_0c.c :

```
#include <linux/module.h>
extern int sys_close( int fd );
static int __init sys_init( void ) {
   void* Addr;
   Addr = (void*)sys_close;
   printk( "sys_close address: %p\n", Addr );
   return -1;
}
module_init( sys_init );
$ sudo insmod md_0c.ko
insmod: error inserting 'md_0c.ko': -1 Operation not permitted
$ dmesg
sys_close address: c04dea99
```

Всё идет так, как и можно было предполагать, и адрес обработчика системного вызова `sys_close()` (экспортированный ядром и полученный в ходе выполнения) совпадает с значением, полученным ранее из `/proc/kallsyms`. Теперь точно такой же модуль, но относительно обработчика для симметричного системного

24 Похоже, что это связано с тем, что сиганлы — один из базовых и самых старых механизмов UNIX/POSIX, и создаётся иллюзия, что там всё уже давно описано.

вызова sys_open():

md_0o.c :

```
#include <linux/module.h>
extern int sys_open( int fd );
static int __init sys_init( void ) {
   void* Addr;
   Addr = (void*)sys_open;
   printk( KERN_INFO "sys_open address: %p\n", Addr );
   return -1;
}
module_init( sys_init );
```

Здесь описанный прототип sys_open() не соответствует реальному формату вызова обработчика системного вызова для функции open(), но это не имеет значения, так как мы не собираемся производить вызов, а только получаем адрес для связывания... Но адрес то как раз и не получается:

```
$ make
...
  MODPOST 2 modules
WARNING: "sys_open" [/home/olej/2011_WORK/LINUX-books/examples.DRAFT/sys_call_table/md_0o.ko]
 undefined!
...
$ sudo insmod md_0o.ko
insmod: error inserting 'md_0o.ko': -1 Unknown symbol in module
$ dmesg
md_0o: Unknown symbol sys_open
```

Такой модуль не может быть загружен, потому как он противоречит правилам целостности ядра: содержит неразрешённый внешний символ — этот символ **не экспортируется ядром для связывания**.

Ссылаться по именам к объектам в коде своего модуля мы можем только к тем именам, которые экспортированы (либо ядром, либо любым **ранее** загруженным модулем ядра). Где мы можем уточнить какие из символов ядра являются экспортируемыми, а какие нет? Когда речь идёт о нескольких десятков тысяч символов ядра:

```
$ cat /proc/kallsyms | wc -l
69698
```

Ищем эту информацию вот здесь:

```
$ cat /lib/modules/`uname -r`/build/Module.symvers | wc -l
9594
```

Вот такой примерно формат имеет каждая строка файла Module.symvers, соответствующая описанию одного экспортируемого символа:

- имя символа (2-я колонка);
- модуль, который экспортирует символ, с указанием **пути** к файлу модуля, или vmlinux, если символ экспортируется непосредственно ядром;
- тип экспортирования, например, EXPORT_SYMBOL_GPL;

```
$ cat /lib/modules/`uname -r`/build/Module.symvers | grep sys_
0x00000000     sys_close      vmlinux EXPORT_SYMBOL
0x00000000     sys_copyarea   vmlinux EXPORT_SYMBOL
0x00000000     fb_sys_write   vmlinux EXPORT_SYMBOL_GPL
0x00000000     nfnetlink_subsys_register     net/netfilter/nfnetlink       EXPORT_SYMBOL_GPL
0x00000000     sys_imageblit  vmlinux EXPORT_SYMBOL
0x00000000     sys_fillrect   vmlinux EXPORT_SYMBOL
0x00000000     nfnetlink_subsys_unregister   net/netfilter/nfnetlink       EXPORT_SYMBOL_GPL
0x00000000     sys_tz  vmlinux EXPORT_SYMBOL
0x00000000     fb_sys_read    vmlinux EXPORT_SYMBOL_GPL
```

Видим (команды чуть выше), что число экспортируемых символов на порядок (10:1) меньше общего числа имён ядра, среди них, в частности, есть `sys_close`, но нет `sys_open` (кроме того, дополнительно показан источник экспорта, в показанном примере это ядро `vmlinux`, но могут быть и модули, и тип экспорта: `EXPORT_SYMBOL` или `EXPORT_SYMBOL_GPL`).

Если же сборка модуля производится в отдельном каталоге (на период отработки), и интерес представляет контроль символов, экспортируемых этим модулем, то информацию о них ищем в локальном файле `Module.symvers` в рабочем каталоге сборки.

Не экспортируемые символы ядра

Означает ли показанное выше, что **только** экспортируемые символы ядра доступны в коде нашего модуля. Нет, это означает только, что **рекомендуемый** способ связывания **по имени** (по **абсолютному адресу** имени) применим только к экспортируемым именам. Экспортирование обеспечивает ещё один дополнительный рубеж контроля для обеспечения целостности ядра (целостность **монолитного** ядра, как видим, это совсем не игрушки) — минимальная некорректность приводит к полному краху операционной системы, иногда при этом она даже не успевает сделать: Oops... Особенно это относится к модулям, подобным тем, к рассмотрению которых мы приступаем (архив `call_table.tgz`).

Как оказывается, и все другие имена (функций, переменных, структур) из `/proc/kallsyms` доступны для использования нашему модулю, если модуль возьмёт их оттуда сам. В простейшем для понимания изложении это могло бы выглядеть так, что модуль должен вычитать `/proc/kallsyms` (чтение мы уже рассматривали раньше), найти там адрес интересующего его имени (а интересовать меня будет `sys_call_table`, как самое фундаментальное понятие ядра), и далее ним воспользоваться... Это **очень грубая** схема, как будет показано дальше, это даже не схема, а модель, объясняющая принцип. Но она полностью реализована в примере модуля `mod_rct.c`, здесь я приведу только центральную часть кода, перебирающую символы ядра:

```
...
static char* file = "/proc/kallsyms";
...
    char buff[ BUF_LEN + 1 ] = "";
   f = filp_open( file, O_RDONLY, 0 );
    while( 1 ) {
       char *p = buff, *find;
       int k;
       *p = '\0';
       do {
           if( ( k = kernel_read( f, n, p++, 1 ) ) < 0 ) {
               printk( "+ failed to read\n" );
               return -EIO;
           };
           *p = '\0';
           if( 0 == k ) break;
           n += k;
           if( '\n' == *( p - 1 ) ) break;
       } while( 1 );
       if( ( debug != 0 ) && ( strlen( buff ) > 0 ) ) {
           if( '\n' == buff[ strlen( buff ) - 1 ] ) printk( "+ %s", buff );
           else printk( "+ %s|\n", buff );
       }
       if( 0 == k ) break;   // EOF

       if( NULL == ( find = strstr( buff, "sys_call_table" ) ) ) continue;
       put_table( buff );
    }
   printk( "+ close file: %s\n", file );
    filp_close( f, NULL );
```

```
...
```

Видно, что это сложно, громоздко и натужно, но сам принцип такое решение хорошо разъясняет. Вот как выглядит исполнение этого модуля (я запускаю его с `time` и видно, что даже на весьма быстром процессоре перебор десятков тысяч имён занимает до секунды):

```
$ time sudo insmod mod_rct.ko
insmod: error inserting 'mod_rct.ko': -1 Operation not permitted
real    0m0.728s
user    0m0.003s
sys     0m0.719s
$ dmesg | tail -n4
[57478.736476] + openning file: /proc/kallsyms
[57478.912136] + sys_call_table address = c07c2438
[57478.912140] + sys_call_table : c044e80c c0443af8 c0408a04 c04e39e3 c04e3a45 c04e2e59 c04e1e59
c0443dce c04e2ead c04ed654  ...
[57479.453508] + close file: /proc/kallsyms
```

Формат вывода `dmesg` и все адреса обработчиков в таблице `sys_call_table` отличаются от показываемых чуть ранее — эта демонстрация производится на другом ядре (очень скоро я объясню тому причину):

```
$ uname -r
2.6.35.14-96.fc14.i686.PAE
```

Только что мы в коде модуля определили тот сакральный адрес таблицы (селектора) системных вызовов, который так старательно прячут разработчики ядра, считая его не безопасным. Помимо адреса таблицы `sys_call_table` модуль выводит 10 первых точек входа этой таблицы. Проверим что представляют из себя эти адреса обратным поиском:

```
$ cat /proc/kallsyms | grep c044e80c
c044e80c T sys_restart_syscall
$ cat /proc/kallsyms | grep c0443af8
c0443af8 T sys_exit
$ cat /proc/kallsyms | grep c0408a04
c0408a04 t ptregs_fork
$ cat /proc/kallsyms | grep c04e39e3
c04e39e3 T sys_read
$ cat /proc/kallsyms | grep c04e3a45
c04e3a45 T sys_write
$ cat /proc/kallsyms | grep c04e2e59
c04e2e59 T sys_open
$ cat /proc/kallsyms | grep c04e1e59
c04e1e59 T sys_close
$ cat /proc/kallsyms | grep c0443dce
c0443dce T sys_waitpid
$ cat /proc/kallsyms | grep c04e2ead
c04e2ead T sys_creat
$ cat /proc/kallsyms | grep c04ed654
c04ed654 T sys_link
```

Это в точности начало того массива адресов обработчиков системных вызовов Linux, индексы которого мы смотрели в начале одной из предыдущих глав:

```
$ cat /usr/include/asm/unistd_32.h
...
#define __NR_restart_syscall   0
#define __NR_exit              1
#define __NR_fork              2
#define __NR_read              3
#define __NR_write             4
#define __NR_open              5
#define __NR_close             6
#define __NR_waitpid           7
```

```
#define __NR_creat             8
#define __NR_link              9
#define __NR_unlink           10
...
```

Недостаток того, что показано выше — его громоздкость и **искусственность**. Но `/proc/kallsyms` есть ни что иное, как только внешнее отображение структур ядра, и, к счастью, ядро предоставляет нам вызов `kallsyms_lookup_name()` (экспортирует его), позволяющий делать поиск в этих структурах. Вот насколько всё стало проще:

mod_kct.c :

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

static int __init ksys_call_tbl_init( void ) {
   void** sct = (void**)kallsyms_lookup_name( "sys_call_table" );
   printk( "+ sys_call_table address = %p\n", sct );
   if( sct ) {
      int i;
      char table[ 120 ] = "sys_call_table : ";
      for( i = 0; i < 10; i++ )
         sprintf( table + strlen( table ), "%p ", sct[ i ] );
      printk( "+ %s ...\n", table );
   }
   return -EPERM;
}

module_init( ksys_call_tbl_init );
MODULE_LICENSE( "GPL" );
```

Всё! Вот как происходит исполнение этого варианта, это в точности тот же результат, и это в 30 раз быстрее по времени (объём вычислений):

```
$ time sudo insmod mod_kct.ko
insmod: error inserting 'mod_kct.ko': -1 Operation not permitted
real    0m0.022s
user    0m0.005s
sys     0m0.013s
$ dmesg | tail -n2
[59595.918185] + sys_call_table address = c07c2438
[59595.918193] + sys_call_table : c044e80c c0443af8 c0408a04 c04e39e3 c04e3a45 c04e2e59 c04e1e59
c0443dce c04e2ead c04ed654  ...
```

Но! ... В каждой бочке мёда должна быть своя ложка дёгтя — проверяем то же, но в чуть более ранней версии ядра:

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
$ make
...
WARNING: "kallsyms_lookup_name" [/home/olej/2011_WORK/LINUX-
books/examples.DRAFT/sys_call_table/call_table/mod_kct.ko] undefined!
```

Конечно! И в ядре 2.6.32 такое имя есть:

```
$ cat /proc/kallsyms | grep kallsyms_ | grep T
c046ca7c T module_kallsyms_on_each_symbol
c046e815 T module_kallsyms_lookup_name
c0471581 T kallsyms_lookup
c04716ec T kallsyms_lookup_size_offset
c0471764 T kallsyms_on_each_symbol
```

```
c04717f2 T kallsyms_lookup_name
```

Но оно не экспортируется:

```
$ cat /lib/modules/`uname -r`/build/Module.symvers | grep kallsyms_
0x00000000     kallsyms_on_each_symbol       vmlinux EXPORT_SYMBOL_GPL
```

Этот вызов ядра стал экспортируемым где-то между версиями ядра 2.6.32 и 2.6.35 (или примерно между пакетными дистрибутивами издания лета 2010г. и весны 2011г.). В более ранних дистрибутивах воспользоваться таким сервисом вам не удастся (это и есть та причина, по которой в сравнительное рассмотрение вовлечены различные ядра, на что указывалось выше).

Но этому несчастью легко помочь! Для этого воспользуемся другим экспортируемым именем, которое мы только-что видели в таблице символов — `kallsyms_on_each_symbol`, этот вызов обеспечивает выполнение заказанной пользовательской функции последовательно **для всех** имён ядра. Он сложнее, и здесь нужны краткие пояснения. Этот вызов имеет прототип (`<linux/kallsyms.h>`):

```
int kallsyms_on_each_symbol( int (*fn)(void *, const char *, struct module *, unsigned long),
                             void *data );
```

Первым параметром (`fn`) он получает указатель на вашу пользовательскую функцию, которая и будет последовательно вызываться **для всех символов** в таблице ядра, а вторым (`data`) — указатель на **произвольный** блок данных (параметров), который будет передаваться **в каждый** вызов этой функции `fn()`. Это достаточно обычная практика, подобные аналогии мы видим, например, при создании нового потока (как потока ядра, так и потока пользовательского пространства в POSIX API). Прототип пользовательской функции `fn`, которая циклически вызывается для каждого имени:

```
int func( void *data, const char *symb, struct module *mod, unsigned long addr );
```

где:

`data` — блок параметров, заполненный в вызывающей единице, и переданный из вызова функции `kallsyms_on_each_symbol()` (2-й параметр вызова), как это описано выше, здесь, как раз, и хорошо передать имя того символа, который мы разыскиваем;

`symb` — символьное изображение (строка) имени из таблицы имён ядра, которое обрабатывается на **текущем** вызове `func`;

`mod` — модуль ядра, к которому относится обрабатываемый символ;

`addr` — адрес символа в адресном пространстве ядра (собственно, то, что мы и ищем) ;

Перебор имён таблицы ядра можно прервать на текущем шаге (из соображений эффективности, если мы уже обработали требуемые нам символы), если пользовательская функция `func` возвратит **ненулевое** значение. Этого более чем достаточно для того, чтобы построить следующий, 3-й эквивалент нашим предыдущим модулям:

mod_koes.c :

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

static int nsym = 0;
static unsigned long taddr = 0;

int symb_fn( void* data, const char* sym, struct module* mod, unsigned long addr ) {
   nsym++;
   if( 0 == strcmp( (char*)data, sym ) ) {
      printk( "+ sys_call_table address = %lx\n", addr );
      taddr = addr;
      return 1;
   }
   return 0;
};

static int __init ksys_call_tbl_init( void ) {
```

```
    int n = kallsyms_on_each_symbol( symb_fn, (void*)"sys_call_table" );
    if( n != 0 ) {
       int i;
       char table[ 120 ] = "sys_call_table : ";
       printk( "+ find in position %d\n", nsym );
       for( i = 0; i < 10; i++ ) {
          unsigned long sa = *( (unsigned long*)taddr + i );
          sprintf( table + strlen( table ), "%p ", (void*)sa );
       }
       printk( "+ %s ...\n", table );
    }
    else printk( "+ symbol not found\n" );
    return -EPERM;
}

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

Для убедительности, возвратимся и выполним этот код в ядре 2.6.32, где у нас не работал предыдущий пример:

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
$ time sudo insmod mod_koes.ko
insmod: error inserting 'mod_koes.ko': -1 Operation not permitted
real    0m0.042s
user    0m0.005s
sys     0m0.027s
$ dmesg | tail -n30 | grep +
+ sys_call_table address = c07ab3d8
+ find in position 25239
+ sys_call_table : c044ec61 c0444f63 c040929c c04e149d c04e12fc c04dec35 c04dea99 c0444767
c04dec60
$ cat /proc/kallsyms | wc -l
69423
$ cat /proc/kallsyms | grep c04dec35
c04dec35 T sys_open
```

Хорошо видно, что:

- это те же результаты, что и в первом примере (читающем `/proc/kallsyms`);
- намного (в ~20 раз) короче время выполнения;
- для нахождения требуемого символа читалась не вся таблица имён ядра (`69423` символов), а только около (`25239`) одной её трети;

Вот с этого места мы можем использовать в своём коде не только экспортируемые, но и **любые имена** ядра, и, более того, можем это делать для ядра любой версии!

Использование не экспортируемых символов

Теперь мы умеем получать в своём коде модуля адреса любых, не только экспортируемых ядром символов. Это прямой путь к их практическому использованию. Первейшим применением, и просто просящимся к реализации, была бы подмена функции обработчика системного вызова, о которой мы говорили ранее. Но именно потому, что это тривиально, мы не станем этим заниматься[25]. А сделаем куда более красивый трюк (архив `new_sys.tgz`): в качестве иллюстрации возможности и реализации технических приёмов, мы выполним пользовательский библиотечный[26] вызов `printf()` из кода модуля ядра! Выглядит это так:

25 Кроме того, писателям вирусов тоже нужно оставить часть работы на самостоятельную проработку?
26 О которых мы говорили в начале книги, что из ядра нельзя вызывать библиотечные вызовы POSIX API. Из ядра можно всё! Но только

mod_wrc.c :

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

static unsigned long waddr = 0;
static char buf[ 80 ];
static int len;

int symb_fn( void* data, const char* sym, struct module* mod, unsigned long addr ) {
   if( 0 == strcmp( (char*)data, sym ) ) {
      waddr = addr;
      return 1;
   }
   else return 0;
}

/* <linux/syscalls.h>
asmlinkage long sys_write(unsigned int fd, const char __user *buf,
                          size_t count); */
static asmlinkage long (*sys_write) (
   unsigned int fd, const char __user *buf, size_t count );

static int do_write( void ) {
   int n;
   mm_segment_t fs = get_fs();
   set_fs( get_ds() );
   sys_write = (void*)waddr;
   n = sys_write( 1, buf, len );
   set_fs(fs);
   return n;
}

static int __init wr_init( void ) {
   int n = kallsyms_on_each_symbol( symb_fn, (void*)"sys_write" );
   if( n != 0 ) {
      sprintf( buf, "адрес системного обработчика sys_write = %lx\n", waddr );
      len  = strlen( buf ) + 1;
      printk( "+ [%d]: %s", len, buf );
      n = do_write();
      printk( "+ write return : %d\n", n );
   }
   else
      printk( "+ %d: symbol not found\n", n );
   return -EPERM;
}

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

Собственно, выполняем мы в этом примере не `printf()`, но вспомнив, что **библиотечный** вызов `printf()`, как обсуждалось ранее, является не чем иным, как последовательность **библиотечного** вызова `sprintf()` с последующим **системным** вызовом `write( 1, ... )` — мы фактически решаем поставленную задачу, так как мы имитируем в точности ту же последовательность вызовов:

```
$ sudo insmod mod_wrc.ko
адрес системного обработчика sys_write = c04e12fc
```

осторожно.

```
insmod: error inserting 'mod_wrc.ko': -1 Operation not permitted
$ dmesg | tail -n30 | grep +
+ [77]: адрес системного обработчика sys_write = c04e12fc
+ write return : 77
```

Вывод строки происходит **до** вывода о завершении выполнения кода модуля, и на **графический терминал** (X11), мы видели уже такое мельком при рассмотрении системных вызовов. Естественно, вызов `write()` произведен из функции инициализации модуля, и, поэтому, осуществляет вывод на управляющий терминал **вызывающего** процесса, в данном случае такой процесс — `insmod`. Для проверки (адреса обработчика `sys_write` ... хотя что тут проверять?) проделаем:

```
$ cat /proc/kallsyms | grep T | grep sys_write
c04e12fc T sys_write
c04e196b T sys_writev
c05f99fc T fb_sys_write
```

В коде модуля нет ничего нетривиального, за исключением, возможно, маленького вопроса: откуда я взял прототип **для своей** новой функции `sys_write()`, которой позже присвою адрес системного обработчика вызова `sys_write`? Вот это определение о котором идёт речь:

```
asmlinkage long sys_write(unsigned int fd, const char __user *buf,
                          size_t count);
```

Конечно же — бесстыдно списал из заголовочного файла `<linux/syscalls.h>`. О чём даже вписал комментарий в код модуля, чтобы это не забыть. И советую и вам в точности так же поступать в отношении и **всех других** системных вызовов. Потому, что в данном случае игра идёт уже «на грани фола», и любое «не угадал» в отношении прототипа немедленно чревато полным крахом системы. В частности, краеугольным для некоторых системных обработчиков, да и других не экспортируемых функций ядра, будет наличие или отсутствие определения : `asmlinkage` — при его наличии параметры вызова будут поочерёдно (от первого до последнего) заносится в регистры процессора, а при его отсутствии — заталкиваться в стек (от последнего к первому) вызывающим процессом, в соответствии с соглашениями о вызове языка C.

Этот пример порождает ещё ряд интересных вопросов, и было бы жалко не удовлетворить любопытство в отношении них. Сработает такой вызов только для вывода в **графический терминал** (X11), или и в **текстовую консоль** (<Ctrl><Alt><F2>, например)? Да, сработает.

А зачем здесь в качестве строки вывода использовалась замысловатая русскоязычная строка, что совсем не приветствуется в программировании ядра? А потому, что в тракте прохождения сообщений от ядра задействовано слишком много последовательных слоёв и компонент (реализация `printk()`, демон журнала, файл журнала, терминальная система UNIX, терминал, визуализатор `dmesg`, ...), и такой вывод может быть хорошей «проверкой на вшивость» согласованности и прозрачности всех этих компонент. И он любопытен... Проверим предварительно установки:

```
$ echo $LANG
ru_RU.UTF-8
```

И установим уровень диагностики ниже порога вывода (**только** с терминала `root`, не `sudo` — мы об этом говорили ранее), иначе просто бессмысленно запускать модуль из консоли:

```
# cat /proc/sys/kernel/printk
3       4       1       7
# echo 8 > /proc/sys/kernel/printk
# cat /proc/sys/kernel/printk
8       4       1       7
```

Выполним **в консоли** всё то же действие:

```
# sudo insmod mod_wrc.ko
...
```

И мы увидим в консоли достаточно странную картину:

- вывод `write()` (который выполняется модулем) выведет ожидаемую строку: «адрес ...»

- вывод `dmesg` и `cat /var/log/messages` (выполняемых в **консоли**!) выведет: «адрес ...»

- любимая народная программа `hello_world` (для проверки) выведет в консоли русскоязычную строку: «Привет мир!»

- а вот диагностика **на консоль** `printk()` из ядра выведет чудовищную строку с амляутами (которую я не вспомню как откопировать с консоли в текст, чтобы вам показать), более того, строка длиной 77 символов (UNICODE, UTF-8, однако)...

То есть, реализация `printk()` в ядре: а). выводит диагностику не через системный журнал, а параллельно демону журналирования; б). пытается как-то интерпретировать поток UNICODE символов, пытаясь преобразовывать их побайтно в ASCII, в). тем самым узувечив символы UNICODE.

Итог этих опытов может быть кратко сформулирован так: добавьте к множеству инструментов программирования, доступных в ядре, набор **всех** системных вызовов POSIX API пространства пользователя. Естественно, толкование результатов некоторых из таких системных вызовов в контексте ядра может быть весьма двусмысленным, а иногда такие результаты и просто бессмысленны. Но сами вызовы — выполнимые!

Подмена системных вызовов

Системные вызовы из пользовательских процессов, как это детально обсуждалось ранее, **все** проходят через таблицу с именем `sys_call_table` (это своего рода case-селектор, который передаёт управление на обработчик требуемого запроса). Индекс для адреса обработчика каждого системного вызова в этом массиве и определяется номером системного вызова:

```
$ cat /usr/include/asm/unistd_32.h
...
#define __NR_restart_syscall   0
#define __NR_exit              1
#define __NR_fork              2
#define __NR_read              3
#define __NR_write             4
#define __NR_open              5
#define __NR_close             6
#define __NR_waitpid           7
#define __NR_creat             8
#define __NR_link              9
#define __NR_unlink           10
#define __NR_execve           11
...
```

Это и есть таблица входов для связи системных вызовов пространства пользователя с обработчиками этих вызовов в пространстве ядра (естественно, для 64-бит это будет `unistd_64.h`).

Иногда хотелось бы подменить или добавить позицию (адрес обработчика) в таблице (это техника, благополучно известная программистам ещё со времён MS-DOS, и применяется в различных операционных системах). Такая модификация бывает нужна, например (этим перечень возможностей далеко не исчерпывается):

- Для мониторинга и накопления статистики по какому-либо существующему системному вызову;
- Для добавления **собственного** обработчика нового системного вызова, который затем используется прикладными программами пространства пользователя целевого пакета;
- Так делают программы-вирусы или недоброжелательные программы, пытающиеся перехватить контроль над компьютером;

Из сказанного уже понятно, что намерение модификации таблицы системных вызовов представляется заманчивым, и, в общем, лежит на поверхности. Но реализовать это (с некоторых пор) становится не так просто! Добавить новый системный вызов можно, в принципе, **двумя** основными способами (всё остальное будет какая-то их взаимная комбинация):

- **Статически**, добавив свой файл реализации `arch/i386/kernel/new_calls.c` в дереве исходных

кодов Linux, добавив запись в таблицу системных вызовов `arch/i386/kernel/syscall_table.S`, и включив свою реализацию в сборку ядра, дописав в `arch/i386/kernel/Makefile`, среди многих подобных, строку вида:

```
obj-y += new_calls.o
```

После этого мы собираем новое ядро, как это рассказано в приложении в конце текста, и получаем новое ядро системы, в котором реализован требуемый новый системный вызов в пространстве ядра. Весь процесс детально описан в [25]. Мы не будем обращаться к этому способу, просто потому, что это не входит в круг наших интересов.

– **Динамически**, во время выполнения дополнив таблицу `sys_call_table[]` ядра ссылкой на код собственного модуля, который и реализует новый системный вызов (и сделать это действие в пространстве ядра может, естественно, только код модуля ядра).

До определённого времени (ранее версии 2.6) ядро экспортировало адрес таблицы системных вызовов `sys_call_table[]`. На сейчас, этот символ может присутствовать в таблице имён ядра (`/proc/kallsyms`), но не экспортируется для использования модулями:

```
$ cat /proc/kallsyms | grep 'sys_call'
c052476b t proc_sys_call_handler
c07ab3d8 R sys_call_table
```

Тем не менее, ядро всегда импортирует символ `sys_close`[27], находящийся в начальных позициях таблицы `sys_call_table[]`, который экспортируется ядром:

```
$ cat /proc/kallsyms | grep sys_close
c04dea99 T sys_close
```

Некоторые изощрённые программы, во множестве варьируемые в форумных обсуждениях, разыскивают это известное значение в сегменте кода ядра, определяют по нему местоположение таблицы `sys_call_table[]`, после чего могут динамически добавлять новые, или подменять существующие системные вызовы. Но вам не нужна никакая такая изощрённость, если вы детально разберётесь с тем, как ядро экспортирует символы для использования их из кода модулей, и как можно оперировать с символами, не экспортируемыми ядром — то чем мы занимались выше.

Итак, мы уже собрали выше весь арсенал достаточных средств, чтобы это сделать. Для экспериментов выберем системный вызов `sys_write`. Нам предстоит только определить адрес обработчика этого системного вызова, как мы делали это ранее, и заменить его на свою функцию обработчика. Но загрузка так написанного модуля закончится серьёзной неудачей:

```
$ sudo insmod mod_wrchg_1.ko
...
Message from syslogd@notebook at Dec 31 01:56:41 ...
 kernel:CR2: 00000000c07ab3e8
$ dmesg | tail -n100 | grep -v audit
! адрес sys_call_table = c07ab3d8
! адрес в позиции 4[__NR_write] = c04e12fc
! адрес sys_write = c04e12fc
! адрес нового sys_write = fe1f1024
! CR0 = 8005003b
BUG: unable to handle kernel paging request at c07ab3e8
IP: [<fe1f40b6>] wrchg_init+0xb6/0xd4 [mod_wrchg_1]
*pdpt = 0000000000a8c001 *pde = 0000000036881063 *pte = 00000000007ab161
Oops: 0003 [#1] SMP
...
```

А в результате мы получим аварийно установленный модуль, который невозможно будет даже удалить, не прибегая к перезагрузке системы:

```
$ lsmod | grep mod_
mod_wrchg             2732  1
```

27 На это обращают внимание многие пишущие на эту тему, причину такой избирательности я не могу объяснить, мы ещё вернёмся детально к рассмотрению этого системного вызова далее.

Неудача связана с тем, что таблица адресов системных вызовов находится в сегменте **только для чтений**, и попытка записи в неё приводит к ошибке защиты памяти (обращаем внимание на символ R):

```
$ cat /proc/kallsyms | grep sys_call_table
c07ab3d8 R sys_call_table
```

Этому делу можно помочь — голь на выдумки хитра: мы ведь выполняем код модуля в режиме **супервизора**, в нулевом кольце защиты процессора x86, где всё допустимо! Мы просто на время перезаписи точки входа таблицы `sys_call_table` отменим контроль записи в сегмент, объявленный доступным только для чтения (а затем так же его восстановим). Заметьте, что этот пример работает только для архитектуры i686 (или i386, i486), но и защиту записи мы здесь наблюдаем в i686! На другой платформе будут найдены свои аналогичные решения. В архитектуре i686 за контроль записи отвечает 16-й бит в управляющем регистре процессора CR0 (называемый WP бит), в архитектуре x86_64 картина будет отличаться, я ещё вернусь к этому в два слова.

Итак, рассмотрим код примера (архив `new_sys.tgz`), выполняющего требуемое нами действие. Но используемая здесь некоторая функциональность понадобится нам и далее, в других рассматриваемых модулях, поэтому вынесем её в отдельные включаемые файлы, и рассмотрим их внимательнее:

CR0.c :

```
// 16 бит WP:          |
//                     V
//   3    2    2    1    1    1    0    0  0
//   1    7    3    9    5    1    7    3  0
//   0000 0000 0000 0001 0000 0000 0000 0000 => 0x00010000
//   1111 1111 1111 1110 1111 1111 1111 1111 => 0xfffeffff

// показать управляющий регистр CR0
#define show_cr0()                          \
{  register unsigned r_eax asm ( "eax" ); \
   asm( "pushl %eax" );                     \
   asm( "movl %cr0, %eax" );                \
   printk( "! CR0 = %x\n", r_eax );         \
   asm( "popl %eax");                       \
}

//код выключения защиты записи:
#define rw_enable()              \
asm( "pushl %eax \n"             \
     "movl %cr0, %eax \n"        \
     "andl $0xfffeffff, %eax \n" \
     "movl %eax, %cr0 \n"        \
     "popl %eax" );

//код включения защиты записи:
#define rw_disable()             \
asm( "pushl %eax \n"             \
     "movl %cr0, %eax \n"        \
     "orl $0x00010000, %eax \n"  \
     "movl %eax, %cr0 \n"        \
     "popl %eax" );
```

Здесь показан (комментарием) формат управляющего регистра `cr0` для 32-разрядных процессоров Intel, и несколько макросов, оперирующих с этим регистром. Макросы `rw_enable()` (разрешающий запись в сегмент для чтения) и `rw_disable()` (восстанавливающий контроль записи), реализованы мной как инлайновые ассемблерные вставки, причём **без параметров**, а поэтому регистры в них можно указать и как `%eax` и `%cr0`, (с одним префиксом %, а не двойным).

В таком виде это работает только на 32-разрядной платформе. Для 64-разрядной платформы всё принципиально остаётся так же, но а). должны использоваться 64-разрядные операции (суффикс `q` в записи мнемоник команд AT&T), б). вместо регистра `%eax` должен определяться регистр `%rax`, в). другой бит CR0 ответственный за защиту записи, утверждается, что это должно быть (я это не проверял):

```
asm( "andq $0xfffffffffffeffff, %rax" );
...
asm( "orq $0x0000000000001000, %rax" );
```

Другой включаемый файл (тоже только из соображений повторяемости) содержит реализацию функции `find_sym()`, осуществляющий поиск заказанного символа ядра (параметр функции), и возвращает найденный адрес (или не возвращает, если такого символа в ядре не существует).

find.c :

```
static void* find_sym( const char *sym ) {
   static unsigned long faddr = 0; // static!!!
   // ----------- вложенная функция - расширение GCC ---------
   int symb_fn( void* data, const char* sym, struct module* mod, unsigned long addr ) {
      if( 0 == strcmp( (char*)data, sym ) ) {
         faddr = addr;
         return 1;
      }
      else return 0;
   };
   // ------------------------------------------------------
   kallsyms_on_each_symbol( symb_fn, (void*)sym );
   return (void*)faddr;
}
```

В коде функции `find_sym()` использовано такое синтаксическое расширение `gcc` (не допускаемое ANSI стандартами языка C), как определение вложенной функции `symb_fn()`, локальной по отношению к вызывающей (такие вещи очень характерны, например, для языков группы PASCAL Н.Вирта). Такой приём можно счесть и за трюкачество, но он позволило описать возврат адреса любого имени `sym` из таблицы `/proc/kallsyms`, не прибегая ни к каким глобальным переменным для общего использования.

Примечание: Напомню лишний раз, что вы всё-равно не скомпилируете модуль другим компилятором кроме `gcc`, который в любых версиях допускает такое расширение.

Теперь мы готовы рассмотреть код самого модуля, который выглядит так:

mod_wrchg.c :

```
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/uaccess.h>
#include <linux/unistd.h>

#include "../find.c"
#include "../CR0.c"

asmlinkage long (*old_sys_write) (
       unsigned int fd, const char __user *buf, size_t count );

asmlinkage long new_sys_write (
       unsigned int fd, const char __user *buf, size_t count ) {
   int n;
   if( 1 == fd ) {
      static const char prefix[] = ":-) ";
      mm_segment_t fs = get_fs();
      set_fs( get_ds() );
```

```
      n = old_sys_write( 1, prefix, strlen( prefix ) );
      set_fs(fs);
   }
   n = old_sys_write( fd, buf, count );
   return n;
};
EXPORT_SYMBOL( new_sys_write );

static void **taddr; // адрес таблицы sys_call_table

static int __init wrchg_init( void ) {
   void *waddr;
   if( ( taddr = find_sym( "sys_call_table" ) ) != NULL )
      printk( "! адрес sys_call_table = %p\n", taddr );
   else {
      printk( "! sys_call_table не найден\n" );
      return -EINVAL;
   }
   old_sys_write = (void*)taddr[ __NR_write ];
   printk( "! адрес в позиции %d[__NR_write] = %p\n", __NR_write, old_sys_write );
   if( ( waddr = find_sym( "sys_write" ) ) != NULL )
      printk( "! адрес sys_write = %p\n", waddr );
   else {
      printk( "! sys_write не найден\n" );
      return -EINVAL;
   }
   if( old_sys_write != waddr ) {
      printk( "! непонятно! : адреса не совпадают\n" );
      return -EINVAL;
   }
   printk( "! адрес нового sys_write = %p\n", &new_sys_write );
   show_cr0();
   rw_enable();
   taddr[ __NR_write ] = new_sys_write;
   show_cr0();
   rw_disable();
   show_cr0();
   return 0;
}

static void __exit wrchg_exit( void ) {
   printk( "! адрес sys_write при выгрузке = %p\n", (void*)taddr[ __NR_write ] );
   rw_enable();
   taddr[ __NR_write ] = old_sys_write;
   rw_disable();
   return;
}

module_init( wrchg_init );
module_exit( wrchg_exit );

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

Это самый большой и сложный пример, из всех которые мы рассматривали до сих пор. Но он стоит детального рассмотрения. Да, здесь достаточно много элементов, которые можно отнести к трюкачеству, из которых нужно отметить:

- новый обработчик `new_sys_write()` системного вызова `sys_write` при выводе на `SYSOUT` делает **предшествующий** выводимой строке вывод собственного префикса (строки `":-) "`), причём для этого ему необходимо сначала использовать сегмент данных в адресном пространстве ядра (взять данные строки вывода из области ядра), после чего, восстановить контроль принадлежности адреса сегменту данных пространства пользователя (для вывода оригинальной переданной строки);
- при выгрузке модуля, он должен **обязательно** восстановить прежнюю функцию обработчик `old_sys_write()`;
- в таком восстановлении скрыта потенциальная угроза критической для ядра ошибки, в том случае (крайне редком), если некоторый другой модуль подменит адрес обработчика **после** нашего — то, что показано, годится только как иллюстрационный упрощённый вариант;

В конечном счёте, почти все эти элементы встречались ранее, теперь только объединяем их вместе. А вот как выглядит протокол выполнения этого примера:

- работа в ядре (а уж тем более с таблицей системных вызовов) крайне рискованное занятие (как у сапёра), поэтому в коде делается двойная перепроверка: адрес обработчика вызова, найденный как символ ядра `sys_write`, сравнивается с адресом в позиции `__NR_write` в таблице `sys_call_table`;

```
$ sudo insmod mod_wrchg.ko
:-) $ :-) e:-) c:-) h:-) o:-)  :-) c:-) т:-) р:-) о:-) к:-) а:-)
:-) строка
:-) $ lsmod | head -n4
:-) :-) Module                  Size  Used by
:-) mod_wrchg               1382  0
:-) fuse                   48375  2
:-) ip6table_filter         2227  0
:-) $ sudo rmmod mod_wrchg
$
```

Мы подменили один из самых используемых при работе с терминалом системных вызовов Linux, поэтому приготовьтесь, что объясняться с системой в командной строке станет очень непросто, даже всего лишь для того, чтобы удалить установленный модуль. Но восстанавливается система после удаления корректно...

Теперь можно посмотреть и на то, как это происходило и с точки зрения системного журнала:

```
$ dmesg | tail -n120 | grep -v audit
! адрес sys_call_table = c07ab3d8
! адрес в позиции 4[__NR_write] = c04e12fc
! адрес sys_write = c04e12fc
! адрес нового sys_write = fd8ae024
! CR0 = 8005003b
! CR0 = 8004003b
! CR0 = 8005003b
! адрес sys_write при выгрузке = fd8ae024
```

В этом примере специально показывался отладочный вывод содержимого управляющего регистра `cr0` процессора.

Добавление новых системных вызовов

Эта задача очень похожа на предыдущую, причём, её практическое значение может оказаться существенно выше, например, для организации некоторой собственной функциональности в рамках **крупного прикладного** проекта (или сугубо проприетарного проекта, не переносимого между экземплярами установки). Один из способов реализации такой возможности упоминался ранее — это добавления кода в ядро с его последующей пересборкой, то, что было названо статической модификацией таблицы системных вызовов. Слабая сторона такого решения состоит в его плохой переносимости: проект сложно устанавливать на систему заказчика, ядро которой должно быть модифицировано. Больший интерес должна представлять возможность проделать это динамически.

В отличие от обсуждавшегося выше примера подмены системного вызова, эта задача, при всём её сходстве, имеет некоторые отягчающие особенности:

- Размер оригинальной таблицы системных вызовов `sys_call_table` произвольно увеличивается от версии к версии ядра (и существенно зависит от конкретной процессорной платформы).
- Константа, задающая размерность таблицы (известная в ядре как `__NR_syscall_max`), является препроцессорной константой периода компиляции, и неизвестна на периоде выполнения (по крайней мере, мне неизвестна).
- Пытаясь добавить собственный системный вызов мы не имеем права выйти за пределы этой существующей таблицы.

Размер таблицы `sys_call_table` достаточно велик, и меняется от версии к версии ядра:

```
$ cat /proc/kallsyms | grep ' sys_' | grep T | wc -l
345
```

Примечание: Это достаточно грубая оценка только **порядка** величины: некоторые обработчики в современных версиях **подменены** на другие их формы, показательным примером того является обработчик (`ptregs_fork`) вызова `fork()` в одной из начальных (`__NR_fork = 2`) позиций `sys_call_table` :

```
$ cat /proc/kallsyms | grep ptregs_fork
c040929c t ptregs_fork
$ cat /proc/kallsyms | grep sys_fork
c040ee13 T sys_fork
```

Смягчает выше перечисленные ограничивающие обстоятельства то, что таблица системных вызовов **не плотная**, в ней есть не использующиеся позиции (оставшиеся от устаревших системных вызовов, или зарезервированные на будущее). Все такие позиции заполнены одни адресом — указателем на функцию обработчика нереализованных вызовов `sys_ni_syscall`:

```
$ cat /proc/kallsyms | grep sys_ni_syscall
c045b9a8 T sys_ni_syscall
```

Следуя таким путём, мы можем добавить свой новый обработчик системного вызова в **любую** неиспользуемую позицию таблицы `sys_call_table`. Текстуально (в исходном коде, **статически**) структуру таблицы можем детально рассмотреть, **для интересующей нас платформы**, в дереве исходных кодов ядра. Для образца используем дерево ядра 3.0.9 (в листинге показаны только неиспользуемые позиции, а комментарии вида `__NR_#` в конце строк добавлены мною) :

```
$ cat /usr/src/linux-3.0.9/arch/x86/kernel /syscall_table_32.S
ENTRY(sys_call_table)
        .long sys_restart_syscall     /* 0 - old "setup()" system call, used for restarting */
        .long sys_exit
        .long ptregs_fork
...
        .long sys_ni_syscall    /* old break syscall holder */       //17
        .long sys_ni_syscall    /* old stty syscall holder */        //31
        .long sys_ni_syscall    /* old gtty syscall holder */        //32
        .long sys_ni_syscall    /* 35 - old ftime syscall holder */  //35
        .long sys_ni_syscall    /* old prof syscall holder */        //44
        .long sys_ni_syscall    /* old lock syscall holder */        //53
        .long sys_ni_syscall    /* old mpx syscall holder */         //56
        .long sys_ni_syscall    /* old ulimit syscall holder */      //58
        .long sys_ni_syscall    /* old profil syscall holder */      //98
        .long sys_ni_syscall    /* old "idle" system call */         //112
        .long sys_ni_syscall    /* old "create_module" */            //127
        .long sys_ni_syscall    /* 130: old "get_kernel_syms" */     //130
        .long sys_ni_syscall    /* reserved for afs_syscall */       //137
        .long sys_ni_syscall    /* Old sys_query_module */           //167
        .long sys_ni_syscall    /* reserved for streams1 */          //188
        .long sys_ni_syscall    /* reserved for streams2 */          //189
        .long sys_ni_syscall    /* reserved for TUX */               //222
```

```
        .long sys_ni_syscall                                    //223
        .long sys_ni_syscall                                    //251
        .long sys_ni_syscall    /* sys_vserver */               //273
        .long sys_ni_syscall    /* 285 */ /* available */       //285
...
        .long sys_setns                                         // 346
```

Видим, что для этой версии ядра таблица имеет 347 позиций, из которых 21 не задействованы (и никогда не могут быть задействованы, потому, что назначить новый системный вызов устаревшему старому — это слишком рискованный ход: никто не гарантирует систему от выполнения весьма старых программ, а последствия этого были бы непредсказуемы для ядра).

Анализу неиспользуемых позиций (в динамике) и будет посвящён первый рассматриваемый модуль (архив `add_sys.tgz`, он разделён с предыдущим `new_sys.tgz` только для того, чтобы не загромождать проект, и использует совместно используемые включаемые файлы, рассмотренные ранее):

ni-test.c :

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

static unsigned long asct = 0, anif = 0;

int symb_fn( void* data, const char* sym, struct module* mod, unsigned long addr ) {
   if( 0 == strcmp( "sys_call_table", sym ) )
      asct = addr;
   else if( 0 == strcmp( "sys_ni_syscall", sym ) )
      anif = addr;
   return 0;
}

#define SYS_NR_MAX_OLD 340
// - этот размер таблицы взят довольно произвольно из ядра 2.6.37
static void show_entries( void ) {
   int i, ni = 0;
   char buf[ 200 ] = "";
   for( i = 0; i <= SYS_NR_MAX_OLD; i++ ) {
      unsigned long *taddr = ((unsigned long*)asct) + i;
      if( *taddr == anif ) {
         ni++;
         sprintf( buf + strlen( buf ), "%03d, ", i );
      }
   }
   printk( "! найдено %d входов: %s\n", ni, buf );
}

static int __init init_driver( void ) {
   kallsyms_on_each_symbol( symb_fn, NULL );
   printk( "! адрес таблицы системных вызовов = %lx\n", asct );
   printk( "! адрес не реализованных вызовов  = %lx\n", anif );
   if( 0 == asct || 0 == anif ) {
      printk( "! не найдены символы ядра\n" );
      return -EFAULT;
   }
   show_entries();
   return -EPERM;
}
module_init( init_driver );

MODULE_DESCRIPTION( "test unused entries" );
```

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

Код достаточно простой, не вдаваясь в обсуждения деталей, смотрим, что покажет он (выполнение в ядре 2.6.32):

```
$ sudo insmod ni-test.ko
insmod: error inserting 'ni-test.ko': -1 Operation not permitted
$ dmesg | tail -n 18 | grep !
! найдено 26 входов: 017, 031, 032, 035, 044, 053, 056, 058, 098, 112, 127, 130, 137, 167, 188,
189, 222, 223, 251, 273, 274, 275, 276, 285, 294, 317,
```

Резюме:

- почти 10% размера таблицы системных вызовов не используются;
- стабильность этого списка очень высока (сознательно для анализа кода была выбрана версия 3.0.9, а для исполнения в динамике версия 2.6.32, отстоящие друг от друга более, чем на 2 года выпуска);

Теперь мы готовы реализовать модуль, реализующий новый системный вызов, и программу пользовательского пространства (процесс), использующий такой вызов. Номер нового вызова определён в общем заголовочном файле, для согласованности использования модулем и программой:

syscall.h :

```
// номер нового системного вызова
#define __NR_own 223
// может быть взят любой, полученный при загрузке модуля ni-test.ko
// для ядра 2.6.32 был получен ряд:
// 017, 031, 032, 035, 044, 053, 056, 058, 098, 112, 127, 130, 137,
// 167, 188, 189, 222, 223, 251, 273, 274, 275, 276, 285, 294, 317,
```

Прежде всего создадим тестовую программу, использующую новый системный вызов, например так:

syscall.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "syscall.h"

static void do_own_call( char *str ) {
   int n = syscall( __NR_own, str, strlen( str ) );
   if( n >= 0 )
      printf( "syscall return %d\n", n );
   else
      printf( "syscall error %d : %s\n", n, strerror( -n ) );
}

int main( int argc, char *argv[] ) {
   char str[] = "DEFAULT STRING";
   if( 1 == argc ) do_own_call( str );
   else
      while( --argc > 0 ) do_own_call( argv[ argc ] );
   return EXIT_SUCCESS;
};
```

Программа может делать один или серию (если использовать несколько параметров в командной строке) системных вызовов, передаёт символьный параметр в вызов (подобно тому, как это делает, например `sys_write`) для того, чтобы в реализационной части системного вызова показать как эта строка копируется из пространства пользователя в пространство ядра (или могла бы возвращаться обратно). Но главным интересом здесь есть код возврата: успех или неудача выполнения системного вызова.

А вот и модуль, подхватывающий выполнение этого системного вызова уже в пространстве ядра:

add-sys.c :

```
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/uaccess.h>
#include <linux/unistd.h>

#include "../find.c"
#include "../CR0.c"
#include "syscall.h"

// системный вызов с двумя параметрами:
asmlinkage long (*old_sys_addr) ( const char __user *buf, size_t count );

asmlinkage long new_sys_call ( const char __user *buf, size_t count ) {
   static char buf_msg[ 80 ];
   int res = copy_from_user( buf_msg, (void*)buf, count );
   buf_msg[ count ] = '\0';
   printk( "! передано %d байт: %s\n", count, buf_msg );
   return res;
};
EXPORT_SYMBOL( new_sys_call );

static void **taddr; // адрес таблицы sys_call_table

static int __init new_sys_init( void ) {
   void *waddr;
   if( ( taddr = find_sym( "sys_call_table" ) ) != NULL )
      printk( "! адрес sys_call_table = %p\n", taddr );
   else {
      printk( "! sys_call_table не найден\n" );
      return -EINVAL;
   }
   old_sys_addr = (void*)taddr[ __NR_own ];
   printk( "! адрес в позиции %d[__NR_own] = %p\n", __NR_own, old_sys_addr );
   if( ( waddr = find_sym( "sys_ni_syscall" ) ) != NULL )
      printk( "! адрес sys_ni_syscall = %p\n", waddr );
   else {
      printk( "! sys_ni_syscall не найден\n" );
      return -EINVAL;
   }
   if( old_sys_addr != waddr ) {
      printk( "! непонятно! : адреса не совпадают\n" );
      return -EINVAL;
   }
   printk( "! адрес нового sys_call = %p\n", &new_sys_call );
   rw_enable();
   taddr[ __NR_own ] = new_sys_call;
   rw_disable();
   return 0;
}

static void __exit new_sys_exit( void ) {
   printk( "! адрес sys_call при выгрузке = %p\n", (void*)taddr[ __NR_own ] );
   rw_enable();
   taddr[ __NR_own ] = old_sys_addr;
   rw_disable();
   return;
```

```
}

module_init( new_sys_init );
module_exit( new_sys_exit );

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

Здесь также делается двойная проверка (перестраховка) соответствия адреса в заданной (`__NR_own`) позиции таблицы `sys_call_table` адресу `sys_ni_syscall`.

И теперь оцениваем то, что у нас получилось:

```
$ ./syscall
syscall return 38 : Function not implemented
```

Это было до загрузки модуля, реализующего требуемый программе системный вызов. Загружаем такой модуль:

```
$ sudo insmod add-sys.ko
$ lsmod | grep 'sys'
add_sys                 1432  0
$ dmesg | tail -n 30 | grep !
! адрес sys_call_table = c07ab3d8
! адрес в позиции 223[__NR_own] = c045b9a8
! адрес sys_ni_syscall = c045b9a8
! адрес нового sys_call = fdd7c024
$ ./syscall новые аргументы
syscall return 0 : Success
syscall return 0 : Success
```

Программа сделала 2 системных вызова:

```
$ dmesg | tail -n 30 | grep !
! адрес sys_call_table = c07ab3d8
! адрес в позиции 223[__NR_own] = c045b9a8
! адрес sys_ni_syscall = c045b9a8
! адрес нового sys_call = fdd7c024
! передано 18 байт: аргументы
! передано 10 байт: новые
```

Выгружаем реализующий модуль:

```
$ sudo rmmod add-sys
$ dmesg | tail -n 50 | grep !
! адрес sys_call_table = c07ab3d8
! адрес в позиции 223[__NR_own] = c045b9a8
! адрес sys_ni_syscall = c045b9a8
! адрес нового sys_call = fdd7c024
! передано 18 байт: аргументы
! передано 10 байт: новые
! адрес sys_call при выгрузке = fdd7c024
```

И повторяем выполнение только-что успешно выполнявшейся программы::

```
$ ./syscall 1 2 3
syscall return 38 : Function not implemented
syscall return 38 : Function not implemented
syscall return 38 : Function not implemented
```

Ядро не в состоянии поддержать более выполнение требуемого программе системного вызова!

По образу и подобию показанного может быть установлен произвольный новый обработчик системного вызова в системе.

Скрытый обработчик системного вызова[28]

Интересен сам по себе вопрос: может ли модуль в принципе установить обработчик (подменив существующий, или добавив новый) таким образом, чтобы ядро системы «не знало» об этом, не выявляя такого модуля средствами диагностики `lsmod` или в файловой системе `/proc`. Интерес этот чисто прагматический, ответ на него определяет может ли вредоносные программы произвести подобные изменения. И ответ на него — положительный! Для этого такой модуль должен:

- выделить динамически блок памяти для кода функции обработчика нового системного вызова (а, возможно, и отдельный блок для собственных данных);
- переместить код функции обработчика в такую динамическую область, возможно, установив **признак выполнимости** (в 64-бит или PAE архитектуре) для страниц этой динамической памяти;
- установить в таблице `sys_call_table` адрес обработчика на этот **перемещённый** код;
- завершиться функцию инициализации модуля с кодом отличным от нуля, тем самым модуль фактически не устанавливается и не фиксируется ядром;

Идея здесь состоит в том, что блоки памяти, выделяемые динамически запросами `kmalloc()` и другими (мы подробно рассматривали это ранее) продолжают существовать, даже если модуль, их создавший, прекратил существование. В этом случае возникают объекты в памяти, к которым потеряны все пути доступа, которые не могут быть никаким образом уничтожены. Подобные механизмы, если они присутствуют (сознательно, или по ошибке), ведут к постепенной деградации операционной системы. Такие примеры своим использованием проясняют подобные вещи лучше, чем десятки увещеваний «на словах».

А теперь рассмотрим всё это на примере (архив `hidden.tgz`), который во многом аналогичен предыдущему, он использует те же включаемый файлы определений, а также очень подобный тестовый процесс пространства пользователя:

`hidden.c :`

```
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/uaccess.h>
#include <linux/unistd.h>
#include <../arch/x86/include/asm/cacheflush.h>

#include "../find.c"
#include "../CR0.c"
#include "syscall.h"

// описания функций обработчиков для разных вариантов:
// asmlinkage long new_sys_call( const char __user *buf, size_t count )
#define VARNUM 5
#ifndef VARIANT
#define VARIANT 0
#endif
#if VARIANT>VARNUM
#undef VARIANT
#define VARIANT 0
#endif

static long shift; // величина сдвига тела функции системного обработчика
#if VARIANT == 0
   #include "null.c"
#elif VARIANT == 1
   #include "local.c"
#elif VARIANT == 2
   #include "getpid.c"
```

28 Эту главу можете пропустить без всякого ущерба для всего последующего изложения.

```
#elif VARIANT == 3
   #include "strlen_1.c"
#elif VARIANT == 4
   #include "strlen_2.c"
#elif VARIANT == 5
   #include "write.c"
#else
   #include "null.c"
#endif

static int __init new_sys_init( void ) {
   void *waddr, *move_sys_call,
       **taddr;                  // адрес таблицы sys_call_table
   size_t sys_size;
   printk( "!... SYSCALL=%d, VARIANT=%d\n", NR, VARIANT );
   if( ( taddr = find_sym( "sys_call_table" ) ) != NULL )
      printk( "! адрес sys_call_table = %p\n", taddr );
   else
      return -EINVAL | printk( "! sys_call_table не найден\n" );
   if( NULL == ( waddr = find_sym( "sys_ni_syscall" ) ) )
      return -EINVAL | printk( "! sys_ni_syscall не найден\n" );
   if( taddr[ NR ] != waddr )
      return -EINVAL | printk( "! системный вызов %d занят\n", NR );
   {  unsigned long end;
      asm( "movl $L2, %%eax \n"
           :"=a"(end)::
         );
      sys_size = end - (long)new_sys_call;
      printk( "! статическая функция: начало= %p, конец=%lx, длина=%d \n",
              new_sys_call, end, sys_size );
   }
   // выделяем блок памяти под функцию обработчик
   move_sys_call = kmalloc( sys_size, GFP_KERNEL );
   if( !move_sys_call ) return -EINVAL | printk( "! memory allocation error!\n" );
   printk( "! выделен блок %d байт с адреса %p\n", sys_size, move_sys_call );
   // копируем резидентный код нового обработчика в выделенный блок памяти
   memcpy( move_sys_call, new_sys_call, sys_size );
   shift = move_sys_call - (void*)new_sys_call;
   printk( "! сдвиг кода = %lx байт\n", shift );
   // снять бит NX-защиты с страницы
   //int set_memory_x(unsigned long addr, int numpages);
   set_memory_x( (long unsigned)move_sys_call, sys_size / PAGE_SIZE + 1 );
   printk( "! адрес нового sys_call = %p\n", move_sys_call );
   rw_enable();
   taddr[ NR ] = move_sys_call;
   rw_disable();
   return -EPERM;
}
module_init( new_sys_init );

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

Вот, собственно, и всё. Но рассмотрения разных для вариантов сборки код включает в себя файлы (`null.c`, `local.c` и им подобные), все которые содержат разные реализации функции обработчика нового системного вызова с единым именем и прототипом:

```
asmlinkage long new_sys_call( const char __user *buf, size_t count );
```

Прототип этого нового системного вызова выбран достаточно произвольно, так же, как и все существующие

системные вызовы различаются параметрами и их числом. Простейший обработчик (`null.c`) показывает только принципиальную возможность, поэтому устанавливаемый ним обработчик системного вызова с номером `NR` (определяется параметром сборки `SYSCALL` и по умолчанию 223) ничего осознанного не делает, но только возвращает признак нормального завершения:

null.c :

```
asmlinkage long new_sys_call( const char __user *buf, size_t count ) {
   asm( "movl $0, %%eax\n"  // эквивалент return 0;
        "popl %%ebp    \n"
        "ret           \n"
        "L2: nop       \n"  // нам нужна метка L2 после return
        ::: "%eax"
      );
   return 0;                // только для синтаксиса
};
```

Поскольку код такого модуля портит таблицу `sys_call_table` и не может её восстанавливать, а при многократном выполнении будет оставлять после себя блоки потерянной (навсегда!) памяти, то нам нужно обзавестись дуальным к нему модулем, который восстанавливает первичное состояние таблицы:

restore.c :

```
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/uaccess.h>
#include <linux/unistd.h>

#include "../find.c"
#include "../CR0.c"
#include "syscall.h"

static int __init new_sys_init( void ) {
   void **taddr;                   // адрес таблицы sys_call_table
   void *waddr, *old_sys_addr;
   if( ( taddr = find_sym( "sys_call_table" ) ) != NULL )
      printk( "! адрес sys_call_table = %p\n", taddr );
   else return -EINVAL | printk( "! sys_call_table не найден\n" );
   if( ( waddr = find_sym( "sys_ni_syscall" ) ) != NULL )
      printk( "! адрес sys_ni_syscall = %p\n", waddr );
   else return -EINVAL | printk( "! sys_ni_syscall не найден\n" );
   old_sys_addr = (void*)taddr[ NR ];
   printk( "! адрес в позиции %d = %p\n", NR, old_sys_addr );
  if( old_sys_addr != waddr ) {
      kfree( old_sys_addr );
      rw_enable();
      taddr[ NR ] = waddr;  // восстановить sys_ni_syscall
      rw_disable();
      printk( "! итоговый адрес обработчика %p\n", taddr[ NR ] );
   }
   else
      printk( "! итоговый адрес обработчика не изменяется\n" );
   return -EPERM;
}
module_init( new_sys_init );

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

Теперь у нас есть всё, чтобы проверить как это работает:

```
$ make VARIANT=0
```

```
...
$ ./syscall
syscall return -38 [fffffffda], reason: Function not implemented
$ sudo insmod hidden.ko
insmod: error inserting 'hidden.ko': -1 Operation not permitted
$ lsmod | grep hid
$
$ ./syscall 1 23 456
syscall return 0 [00000000], reason: Success
syscall return 0 [00000000], reason: Success
syscall return 0 [00000000], reason: Success
$ dmesg | tail -n60 | grep !
!... SYSCALL=223, VARIANT=0
! адрес sys_call_table = c07ab3d8
! статическая функция: начало= f7ecd000, конец=f7ecd00f, длина=15
! выделен блок 15 байт с адреса d9322030
! сдвиг кода = e1455030 байт
! адрес нового sys_call = d9322030
$ sudo insmod restore.ko
insmod: error inserting 'restore.ko': -1 Operation not permitted
$ ./syscall
syscall return -38 [fffffffda], reason: Function not implemented
```

Как легко видеть: модуля `hidden.ko` в системе не установлено, но новый системный вызов с номером `NR` замечательно отрабатывается.

Единственное место в коде `hidden.c`, с подобным которому мы до сих пор не встречались, и которое требует минимальных комментариев, это строка, в которой производится вызов:

```
int set_memory_x( unsigned long addr, int numpages );
```

В старших моделях x86, работающих в 64-бит или расширенном режиме PAE, введен NX-бит защиты страницы памяти от выполнения (старший бит в записи таблицы страниц). В ядре Linux этот аппаратный механизм защиты применяется, начиная с версии 2.6.30. Вызов `set_memory_x()` и снимает ограничение выполнения для `numpages` последовательных страниц (размером `PAGESIZE` каждая), начиная со страницы адреса `addr`. Аналогично, вызов восстанавливает защиту страницы от выполнения. Для ядра 32-бит x86 (не PAE!) этот механизм, как уже было сказано, не используется.

Ассемблерная вставка в функции обработчика представляет полный эквивалент оператора `return 0;` (в полном соответствии с соглашениями языка C, с выталкиванием регистра `%ebp` из стека...). Понадобился такой эквивалент **только** потому, что нам нужна метка `L2` (её адрес) после оператора `return`, а компилятор `gcc` такие метки после завершения кода «оптимизирует». Это место не должно смущать. Весь целевой код обработчика должен предшествовать этой вставке.

Сложность написания кода для таких обработчиков очень высока, но, в принципе, всё это реализуемо. Фактически, при этом предстоит вручную, без помощи компилятора `gcc` (опция `-fPIC`) реализовать нечто подобное PIC-кодированию (Position Independed Code), позиционно независимому (в памяти) кодирование. Такое написание обработчиков само по себе любопытно, и некоторым его вопросам будет посвящено всё остальное изложение до конца раздела. Если вас не интересуют такие детали, вы можете безболезненно опустить всю эту оставшуюся часть.

Сложности перемещаемой функции обработчика связаны с тем, что:

1. Функция не может использовать никакие внешние переменные, описанные вне её тела: после завершения функции инициализации модуля все такие области памяти будут освобождены. Могут использоваться только локальные переменные в стеке.

2. Точно то же самое относится и к другим (локальным) функциям описанным в модуле. Здесь приходит на помощь такое расширение `gcc` как **вложенные описания функций**, которые будут перемещены вместе с телом объемлющей их функции обработчика.

Пример сказанного — следующий обработчик:

local.c :

```
asmlinkage long new_sys_call( const char __user *buf, size_t count ) {
   long res = 1000;
   int inc( int in ) {           // вложенное описание функции
      return ++in;
   }
   res = res + inc( count );
   res = inc( count );
   asm( "movl %%ebx, %%eax\n\t"  // эквивалент return res;
        "leave            \n\t"
        "ret              \n\t"
        "L2: nop          \n\t"  // нам нужна метка L2 после return
        ::"b"(res):"%eax"
      );
   return 0;                     // только для синтаксиса
};
```

И результат использования такого обработчика:

```
$ make VARIANT=1
...
$ sudo insmod hidden.ko
insmod: error inserting 'hidden.ko': -1 Operation not permitted
$ lsmod | grep hid
$ ./syscall
syscall return 16 [00000010], reason: Success
$ ./syscall 123
syscall return 5 [00000005], reason: Success
$ ./syscall 1 22 333 4444
syscall return 6 [00000006], reason: Success
syscall return 5 [00000005], reason: Success
syscall return 4 [00000004], reason: Success
syscall return 3 [00000003], reason: Success
```

3. Следующая сложность будет состоять в том, что функции API ядра, экспортируемые или не экспортируемые ядром (`strlen(), printk(), sys_getpid(), sys_write()` ...), в этом коде нельзя вызывать в привычном виде, записав просто вызов функции по её имени. Адрес такого вызова разрешится в момент **статической** линковки (смещение адреса запишется в поле команды), а при вызове приведёт к вызову со смещением и краху выполнения. Примитивный пример того, как это может быть сделано работоспособно показан ниже:

getpid.c :

```
//c044e690 T sys_getpid
asmlinkage long new_sys_call( const char __user *buf, size_t count ) {
   long res = 0;
   long (*own_getpid)( void );
   own_getpid = 0xc044e690UL;
   res = own_getpid();
   asm( "leave            \n"
        "ret              \n"  // эквивалент return res;
        "L2: nop          \n"  // нам нужна метка L2 после return
        ::"a"(res):
      );
   return 0;                   // только для синтаксиса
};
```

Адрес (не экспортируемый) функции `sys_getpid()` в данном случае для простоты взят **статически** из `/proc/kallsym`, но он может находиться и более сложными способами динамически как это обсуждалось

ранее.

```
$ sudo insmod hidden.ko
insmod: error inserting 'hidden.ko': -1 Operation not permitted
$ ./syscall
syscall return 19783 [00004d47], reason: Success
$ ./syscall 12 13
syscall return 19793 [00004d51], reason: Success
syscall return 19793 [00004d51], reason: Success
$ ps -A | tail -n3
19792 ?        00:00:00 sleep
19796 pts/12   00:00:00 ps
19797 pts/12   00:00:00 tail
```

Другой, более реалистичный пример того же рода:

strlen_2.c :

```
asmlinkage long new_sys_call( const char __user *buf, size_t count ) {
   long res = 0;
   size_t own_strlen( const char* ps ) { // вложенная функция
      long res = 0;
      asm(
        "movl    8(%%ebp), %%eax \n"     // ps -> call parameter
        "movl    %%eax, (%%esp)  \n"
        "call   *%%ebx           \n"
        :"=a"(res):"b"((long)(&strlen)): // strlen() из API ядра
      );
      return res;
   }
   res = own_strlen( buf );
   asm( "leave            \n"
        "ret              \n"           // эквивалент return res;
        "L2: nop          \n"           // нам нужна метка L2 после return
        ::"a"(res):
      );
   return 0;                            // только для синтаксиса
};
```

Этот же пример демонстрирует как функция использует строчную переменную **из пространства пользователя**, адрес которой передан ей в системном вызове:

```
$ make VARIANT=4
...
$ sudo insmod hidden.ko
insmod: error inserting 'hidden.ko': -1 Operation not permitted
$ lsmod | grep hid
$
$ ./syscall 1 23 456 'новая строка'
syscall return 24 [00000018], reason: Success
syscall return 4 [00000004], reason: Success
syscall return 3 [00000003], reason: Success
syscall return 2 [00000002], reason: Success
$ dmesg | tail -n60 | grep !
!... SYSCALL=223, VARIANT=4
! адрес sys_call_table = c07ab3d8
! статическая функция: начало= f7ede000, конец=f7ede018, длина=24
! выделен блок 24 байт с адреса d9085940
! сдвиг кода = e11a7940 байт
! адрес нового sys_call = d9085940
```

Этот же пример вскрывает очередную неприятность при написании подобных перемещаемых функций:

4. Мы успешно использовали выше **скалярные** (`int, long` ...) локальные переменные объявленные внутри функции. Но это не относится к массивам, в частности, к строкам, размещённым как локальные переменные в теле функции.

Модифицирем вызов в предыдущем пример так:

```
res = own_strlen( "1234567" );
```

Или вот так:

```
char str[ 80 ] = "1234567";
res = own_strlen( str );
```

Этим мы переведём его в неработоспособное состояние: хотя указатель строки и размещён в стеке и доступен, но указываемая строка размещается где-то отдельно (в сегменте данных), и значение её указателя оказывается некорректным.

5. Ещё один любопытный и работоспособный пример:

write.c :

```
//c04e12fc T sys_write
asmlinkage long new_sys_call( const char __user *buf, size_t count ) {
   long res = 0;
   asmlinkage size_t (*own_write)( int fd, const char* s, size_t l );
   own_write = (void*)0xc04e12fc;
   res = own_write( 1, buf, count );
   asm( "leave            \n"
        "ret              \n"  // эквивалент return res;
        "L2: nop          \n"  // нам нужна метка L2 после return
        ::"a"(res):
      );
   return 0;                   // только для синтаксиса
};

$ make VARIANT=5
...
$ sudo insmod hidden.ko
insmod: error inserting 'hidden.ko': -1 Operation not permitted
$ lsmod | grep hid
$
$ ./syscall 1 23 456 'новая строка'
новая строка
syscall return 24 [00000018], reason: Success
456
syscall return 4 [00000004], reason: Success
23
syscall return 3 [00000003], reason: Success
1
syscall return 2 [00000002], reason: Success
```

Здесь системный вызов не только корректно принимает переданную ему строку (подобно тому, как это делает `sys_write`), но и выводит параметр своего вызова, эту строку, не в системный журнал, а **на терминал**.

Ещё ряд дополнительных вариантов в написании такого перемещаемого обработчика приведены для рассмотрения в архиве `hidden.tgz`.

Динамическая загрузка модулей

До сих пор мы устанавливали собранные модули выполнением команды `insmod`, когда это происходило на этапе разработки, или `modprobe`, когда модуль отлажен и инсталлирован с системе. Симметрично, удаляли вы все установленные модули командой `rmmod`. Во всех этих случаях операции над модулями производятся **статически**. Возникает вопрос: а можно ли модули устанавливать (и выгружать) **динамически**, то есть по требованию, из собственного программного кода? Это вызывает встречный вопрос: а зачем это надо? Нужно это в самых разнообразных случаях и по разным причинам:

1. Программы утилиты `insmod`, `modprobe` и `rmmod`, сами по себе, являются программами пользовательского пространства, и любопытно, как ними в коде выполняются подобные действия.

2. В некоторых случаях от системы требуется некая функциональность, которая на текущий момент не предоставляется, и которая обеспечивается подгружаемым модулем ядра. В таких случаях было бы в высшей степени удобно подгрузить такой модуль непосредственно по требованию его использования. Классическим и общеизвестным примером такой ситуации есть команда монтирования такой файловой подсистемы (типа `qnx4, minix` и др.), которая не подгружена в системе по умолчанию, например:

```
$ lsmod | grep minix
$ sudo mount -t minix /dev/sda5 /mnt/sda5
$ ls /mnt/sda5
bin  boot  dev  etc  home  mnt  proc  root  sbin  tmp  usr  var
$ lsmod | grep minix
minix                  19212  1
```

В этом случае модуль (`minix.ko`) подгружается также по запросу из **пользовательской** утилиты `mount`.

3. Разработчики определённого класса оборудования могли бы иметь родовой (generic) **модуль** драйвера, который подгружал бы по необходимости видовой модуль драйвера, под конкретную модель оборудования в этом классе. Классическими примерами такого случая (возможно, и решаемых в каждом случае другими средствами) являются целые семейства драйверов (по **типу** плат) в таких интерфейсах к платам **класса** E1/T1 в IP-телефонии, как интерфейс DAHDI (компании Digium), или интерфейс компании Sangoma. В этом случае типовые модули динамически подгружаются из среды родового **модуля ядра**.

4. Развитием предыдущего подхода может быть создана техника построения плагинов: возможность добавления функциональности к сложной системе посредством добавление новых специфических частей, но не затрагивая переделками код существующих частей системы. В области пространства пользователя такая возможность реализуется за счёт использования разделяемых библиотек. В области **модулей ядра** такая возможность могла бы реализоваться посредством динамической загрузки модулей.

Этими случаями покрываются далеко не все области, где пригодилась бы динамическая загрузки модулей.

... из процесса пользователя

Для загрузки модуля из пространства пользовательского процесса (как это делает `insmod`, `modprobe` и `rmmod`) существует системный вызов `sys_init_module()` :

```
asmlinkage long sys_init_module( void __user *umod, unsigned long len, const char __user *uargs );
```

Для выгрузки модуля, соответственно, системный вызов `sys_delete_module()`:

```
asmlinkage long sys_delete_module( const char __user *name, unsigned int flags );
```

Но... Всё, что касается операций динамических загрузки и выгрузки модулей, покрыто изрядным мраком, man-описания и существующие в интернете ссылки описывают какие-то устаревшие реализации, относящиеся к реализациям на границе версий ядра 2.4 и 2.6. Так что здесь придётся изрядно поэкспериментировать, и пособирать воедино оговорки и намёки, разбросанные по исходным текста ядра Linux.

Для всех примеров (архив umaster.tgz) соберём тестовый подопытный модуль, который и будет динамически загружаться в разных окружениях:

slave.c :

```
#include "../common.c"

static char* parm1 = "";
module_param( parm1, charp, 0 );

static char* parm2 = "";
module_param( parm2, charp, 0 );

static char this_mod_file[ 40 ];

static int __init mod_init( void ) {
   set_mod_name( this_mod_file, __FILE__ );
   printk( "+ модуль %s загружен: parm1=%s, parm2=%s\n", this_mod_file, parm1, parm2 );
   return 0;
}

static void __exit mod_exit( void ) {
   printk( "+ модуль %s выгружен\n", this_mod_file );
}
```

Здесь, и во всех последующих примерах модулей (для их укорочения), использован общий включаемый файл:

common.c :

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

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

static int __init mod_init( void );
static void __exit mod_exit( void );

module_init( mod_init );
module_exit( mod_exit );

inline void __init set_mod_name( char *res, char *path ) {
   char *pb = strrchr( path, '/' ) + 1,
        *pe = strrchr( path, '.' );
   strncpy( res, pb, pe - pb );
   sprintf( res + ( pe - pb ), "%s", ".ko" );
};
```

Поработав с модулем автономно, наблюдаем, на будущее, как внешне выглядит именно его работа:

```
$ sudo insmod slave.ko
$ dmesg | tail -n30 | grep +
+ module slave.ko loaded: parm1=, parm2=
$ sudo rmmod slave
$ dmesg | tail -n30 | grep +
+ module slave.ko unloaded
```

При необходимости, модуль готов принять до двух символьных параметров:

```
$ sudo ./inst1 slave.ko parm1='строка1' parm2='строка2'
загрузка модуля: slave.ko parm1=строка1 parm2=строка2
размер файла модуля slave.ko = 94800 байт
модуль slave.ko успешно загружен!
$ dmesg | tail -n30 | grep +
+ модуль slave.ko загружен: parm1=строка1, parm2=строка2
$ lsmod | grep slave
```

```
slave                   1009  0
```

Вот и всё, что от него требуется. Важно то, что, если модулю передать не те параметры, или не в том формате их записи, то он нещадно ругается, и не станет загружаться.

Для загрузки пробного модуля соберём **приложение**:

inst1.c :

```
#include <sys/stat.h>
#include <errno.h>
#include "common.h"

// asmlinkage long sys_init_module          // системный вызов sys_init_module()
//                 ( void __user *umod, unsigned long len, const char __user *uargs );

int main( int argc, char *argv[] ) {
   char parms[ 80 ] = "", file[ 80 ] = SLAVE_FILE;
   void *buff = NULL;
   int fd, res;
   off_t fsize;         /* общий размер в байтах */
   if( argc > 1 ) {
      strcpy( file, argv[ 1 ] );
      if( argc > 2 ) {
         int i;
         for( i = 2; i < argc; i++ ) {
            strcat( parms, argv[ i ] );
            strcat( parms, " " );
         }
      }
   }
   printf( "загрузка модуля: %s %s\n", file, parms );
   fd = open( file, O_RDONLY );
   if( fd < 0 ) {
      printf( "ошибка open: %m\n" );
      return errno;
   }
   {  struct stat fst;
      if( fstat( fd, &fst ) < 0 ) {
         printf( "ошибка stat: %m\n" );
         close( fd );
         return errno;
      }
      if( !S_ISREG( fst.st_mode ) ) {
         printf( "ошибка: %s не файл\n", file );
         close( fd );
         return EXIT_FAILURE;
      }
      fsize = fst.st_size;
   }
   printf( "размер файла модуля %s = %ld байт\n", file, fsize );
   buff = malloc( fsize );
   if( NULL == buff ) {
      printf( "ошибка malloc: %m\n" );
      close( fd );
      return errno;
   }
   if( fsize != read( fd, buff, fsize ) ) {
      printf( "ошибка read: %m\n" );
      free( buff );
      close( fd );
```

```
      return errno;
   }
   close( fd );
   res = syscall( __NR_init_module, buff, fsize, parms );
   free( buff );
   if( res < 0 ) printf( "ошибка загрузки: %m\n" );
   else printf( "модуль %s успешно загружен!\n", file );
   return res;
};
```

Приложение (это и другие) включает файл `common.h` (не `common.c`!):

`common.h :`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/syscall.h>
#include <fcntl.h>

#define SLAVE_FILE "./slave.ko";
```

Здесь, главным образом, определяется имя файла загружаемого модуля (`./slave.ko`), чтобы его не вводить каждый раз при тестировании в виде параметра командной строки. Приложение громоздкое, но логика его крайне проста:

- Ищется файл откомпилированного модуля ядра с указанным именем (переменная `file`);
- Определяется полный размер этого файла (переменная `fsize`);
- В точности под этот размер динамически выделяется буфер чтения `buff`;
- В буфер читается **образ** загружаемого модуля (в формате объектного файла);
- Системному вызову `sys_init_module( void* umod, unsigned long len, const char* uargs )` передаются адрес образа модуля в памяти (`buff`, 1-й параметр) и его длина (`fsize`, 2-й параметр);
- Если необходимо, в качестве 3-го параметра `sys_init_module()` передаётся строка параметров загрузки модуля, если параметры не используются, 3-м параметром указывается пустая строка, но ни в коем случае не NULL.

Сразу же изготовим симметричное приложение для выгрузки модулей (эквивалент `rmmod`):

`rem1.c :`

```
#include "common.h"

//  asmlinkage long sys_delete_module        // системный вызов sys_delete_module()
//                  ( const char __user *name, unsigned int flags );
//  flags: O_TRUNC, O_NONBLOCK

int main( int argc, char *argv[] ) {
   char file[ 80 ]  = SLAVE_FILE;
   int res;

   if( argc > 1 ) strcpy( file, argv[ 1 ] );
   char *slave_mod = strrchr( file, '/' ) != NULL ?
                     strrchr( file, '/' ) + 1 :
                     file;
   if( strrchr( file, '.' ) != NULL )
      *strrchr( file, '.' ) = '\0';
   printf( "выгружается модуль %s\n", slave_mod );
   res = syscall( __NR_delete_module, slave_mod, O_TRUNC );
```

```
   if( res < 0 ) printf( "ошибка выгрузки: %m\n" );
   else printf( "модуль %s успешно загружен!\n", slave_mod );
   return res;
};
```

А теперь — как это всё работает:

```
$ sudo ./inst1 slave.ko parm1='строка1' parm2='строка2'
загрузка модуля: slave.ko parm1=строка1 parm2=строка2
размер файла модуля slave.ko = 94800 байт
модуль slave.ko успешно загружен!
$ dmesg | tail -n30 | grep +
+ модуль slave.ko загружен: parm1=строка1, parm2=строка2
$ lsmod | grep slave
slave                   1009  0
$ sudo ./rem1
выгружается модуль slave
$ dmesg | tail -n30 | grep +
+ модуль slave.ko выгружен
$ lsmod | head -n3
Module                  Size  Used by
fuse                   48375  2
ip6table_filter         2227  0
```

Если кому-то кажется не совсем корректным использование непрямых системных вызовов `syscall()`, то их можно, в конечном итоге, заменить вызовами стандартной системной библиотеки для этих `syscall()`, заменив в листингах всего по одной строке:

inst2.c :

```
...
   res = init_module( buff, fsize, parms ); // вызов sys_init_module()
...
```

rem2.c :

```
...
   res = delete_module( slave_mod, O_TRUNC );  // flags: O_TRUNC, O_NONBLOCK
...
```

Почему я сразу не показал листинги с `init_module()` и `delete_module()`, и зачем морочу вам голову с `syscall()`? Да по очень простой причине: **нигде**, ни в литературе, ни в справочных руководствах Linux, ни в интернет до последнего времени не предоставлялось ни образцов корректного использования `init_module()` и `delete_module()`, ни даже их корректных прототипов вызова. Напротив, все источники полнились просто синтаксически некорректными примерами, соответствующими устаревшим версиями. Поэтому восстанавливать примеры их использования пришлось именно обратным реинжинирингом через `syscall()`. Но в сведих версиях мы находим справку по `init_module()` и не менее интересному вызову `finit_module()`:

```
$ man init_module
SYNOPSIS
       int init_module( void *module_image, unsigned long len, const char *param_values );
       int finit_module( int fd, const char *param_values, int flags );
...
```

... из модуля ядра

В предыдущей части мы загружали (и выгружали) модуль `slave.ko` динамически из кода приложения. Теперь нам предстоит сделать то же самое, но уже из кода **вызывающего** модуля (`master.ko`). Вот образец такого модуля (архив `master.tgz`):

master.c :

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

```
#include <linux/vmalloc.h>
#include "../common.c"
#include "../find.c"

static char* file = "./slave.ko";
module_param( file, charp, 0 );

static char this_mod_file[ 40 ],               // имя файла master-модуля
            slave_name[ 80 ];                  // имя файла slave-модуля
static int __init mod_init( void ) {
   void *waddr;
   long res = 0;
   long len;
   struct file *f;
   void *buff;
   size_t n;
   asmlinkage long (*sys_init_module)          // системный вызов sys_init_module()
              ( void __user *umod, unsigned long len, const char __user *uargs );

   set_mod_name( this_mod_file, __FILE__ );
   if( ( waddr = find_sym( "sys_init_module" ) ) == NULL ) {
      printk( "! sys_init_module не найден\n" );
      res = -EINVAL;
      goto end;
   }
   printk( "+ адрес sys_init_module = %p\n", waddr );
   sys_init_module = waddr;
   strcpy( slave_name, file );
   f = filp_open( slave_name, O_RDONLY, 0 );
   if( IS_ERR( f ) ) {
      printk( "+ ошибка открытия файла %s\n", slave_name );
      res = -ENOENT;
      goto end;
   }
   len = vfs_llseek( f, 0L, 2 ); // 2 - means SEEK_END
   if( len <= 0 ) {
      printk( "+ ошибка lseek\n" );
      res = -EINVAL;
      goto close;
   }
   printk( "+ длина файла модуля = %d байт\n", (int)len );
   if( NULL == ( buff = vmalloc( len ) ) ) {
      res = -ENOMEM;
      goto close;
   };
   printk( "+ адрес буфера чтения = %p\n", buff );
   vfs_llseek( f, 0L, 0 );       // 0 - means SEEK_SET
   n = kernel_read( f, 0, buff, len );
   printk( "+ считано из файла %s %d байт\n", slave_name, n );
   if( n != len ) {
      printk( "+ ошибка чтения\n" );
      res = -EIO;
      goto free;
   }
   {  mm_segment_t fs = get_fs();
      set_fs( get_ds() );
      res = sys_init_module( buff, len, "" );
      set_fs( fs );
      if( res < 0 ) goto insmod;
```

```
   }
   printk( "+ модуль %s загружен: file=%s\n", this_mod_file, file );
insmod:
free:
   vfree( buff );
close:
   filp_close( f, NULL );
end:
   return res;
}

static void __exit mod_exit( void ) {
   asmlinkage long (*sys_delete_module)        // системный вызов sys_delete_module()
               ( const char __user *name, unsigned int flags );
// flags: O_TRUNC, O_NONBLOCK
   void *waddr;
   char *slave_mod = strrchr( slave_name, '/' ) != NULL ?
                     strrchr( slave_name, '/' ) + 1 :
                     slave_name;
   *strrchr( slave_mod, '.' ) = '\0';
   printk( "+ выгружается модуль %s\n", slave_mod );
   if( ( waddr = find_sym( "sys_delete_module" ) ) == NULL ) {
      printk( "! sys_delete_module не найден\n" );
      return;
   }
   printk( "+ адрес sys_delete_module = %p\n", waddr );
   sys_delete_module = waddr;
   {  long res = 0;
      mm_segment_t fs = get_fs();
      set_fs( get_ds() );
      res = sys_delete_module( slave_mod, 0 );
      set_fs( fs );
      if( res < 0 )
         printk( "+ ошибка выгрузки модуля %s\n", slave_mod );
   }
   printk( "+ модуль %s выгружен\n", this_mod_file );
}
```

Логика модуля в точности повторяет логику рассматриваемого раньше пользовательского приложения, только здесь, как всегда в коде ядра, всё делается гораздо осторожнее, и использован другой инструментарий. На завершающем этапе нам неизвестны адреса системных обработчиков `sys_init_module()` и `sys_delete_module()`, они не экспортируются ядром. Поэтому финальные шаги делаются примерно так:

- Адреса символов ядра `sys_init_module` и `sys_delete_module` разыскиваются функцией `find_sym()` (мы подобное рассматривали раньше);
- Выполняется косвенный вызов по указателям функций `sys_init_module` и `sys_delete_module`;
- Эти адреса присваиваются функциональным переменным: `(*sys_init_module)(...)` и `(*sys_delete_module)(...)` (вообще то, имена этих могли бы быть **произвольными**, и совпали с именами системных вызовов ... случайно);
- Это и есть требуемые обработчики системных вызовов;

А теперь то, как всё это выглядит при выполнении:

```
$ sudo insmod master.ko
$ dmesg | tail -n30 | grep +
+ адрес sys_init_module = c0470f50
+ длина файла модуля = 94692 байт
+ адрес буфера чтения = f9d51000
+ считано из файла ./slave.ko 94692 байт
```

```
+ модуль slave.ko загружен: parm1=, parm2=
+ модуль master.ko загружен: file=./slave.ko
```

Предпоследняя строка системного журнала выведена из совсем другого другого модуля, чем обрамляющие её сверху и снизу строки.

```
$ lsmod | head -n4
Module                  Size  Used by
slave                   1001  0
master                  1785  0
fuse                   48375  2
```

Модуль `slave` загружен позже, чем модуль `master`, и между ними нет никаких зависимостей, что правильно.

```
$ sudo rmmod master
$ dmesg | tail -n30 | grep +
+ выгружается модуль slave
+ адрес sys_delete_module = c046f4e8
+ модуль slave.ko выгружен
+ модуль master.ko выгружен
$ lsmod | head -n4
Module                  Size  Used by
fuse                   48375  2
ip6table_filter         2227  0
ip6_tables              9409  1 ip6table_filter
```

Подключаемые плагины

Теперь у нас есть всё для того, чтобы создать макет использования отдельных, подключаемых к основному проекту в качестве плагинов, модулей. Создадим сознательно утрированный пример (архив `plugin.tgz`), но он будет работать с динамическими модулями энергичнее, чем при любой реальной потребности:

1. Добавим новый системный вызов `__NR_str_trans` (мы это уже легко умеем делать):

syscall.h :

```
// номер нового системного вызова
#define __NR_str_trans 223
```

2. Пользовательский процесс передаёт корневому модулю (`master.ko`) строку, изображающую некоторое численное значение (в различных системах счисления: 8, 10, 16), и ожидает получить обратно вычисленное числовое значение:

syscall.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "syscall.h"

static void do_own_call( char *str ) {
   int n = syscall( __NR_str_trans, str, strlen( str ) );
   if( n >= 0 )
      printf( "syscall return %d\n", n );
   else
      printf( "syscall error %d : %s\n", n, strerror( -n ) );
}

int main( int argc, char *argv[] ) {
   if( 1 == argc ) do_own_call( "9876" );
   else {
      int i;
```

```
      for( i = 1; i < argc; i++ )
         do_own_call( argv[ i ] );
   }
   return EXIT_SUCCESS;
};
```

3. Корневой модуль (`master.ko`) устанавливает в таблицу системных вызовов новый системный обработчик (`__NR_str_trans`), но не занимается непосредственно переводом полученной строки в численное значение, в зависимости от синтаксиса записи этой строки-параметра (восьмеричное, десятичное, или шестнадцатеричное значение) он выбирает и загружает соответствующий модуль (`oct.ko`, `dec.ko`, или `hex.ko`) для вычислений, и **загружает** его;

4. Все модули-плагины экспортируют **одно и то же имя** точки входа (`str_translate`), поэтому никакие два из обрабатывающих модулей-плагинов **не могут быть загружены одновременно**;

5. Загрузив модуль-плагин корневой модуль должен найти экспортируемое тем имя `str_translate()`, и передать ему строку на обработку, полученный результат и будет итогом работы системного нового вызова;

6. После обработки полученного запроса модуль-плагин должен быть тут же динамически выгружен, во избежания конфликта экспортируемых имён.

7. Модули обработчики, как легко понятно — совершенно однотипные. Для ещё большого сокращения объёма, все они используют общее включение:

`slave.c :`

```
#include "../common.c"

static char this_mod_file[ 40 ];

long str_translate( const char *buf );
EXPORT_SYMBOL( str_translate );
```

Файл `common.c` мы уже встречали раньше, а сами три плагина-обработчика имеют вид:

`oct.c :`

```
#include "slave.c"

static const char dig[] = "01234567";

long str_translate( const char *buf ) {
   long res = 0;
   const char *p = buf;
   printk( "+ %s : запрос : %s\n", this_mod_file, buf );
   while( *p != '\0' ) {
      char *s = strchr( dig, *p );
      if( s == NULL ) return -EINVAL;
      res = res * 8 + ( s - dig );
      p++;
   }
   return res;
};

static int __init mod_init( void ) {
   set_mod_name( this_mod_file, __FILE__ );
   printk( "+ модуль %s загружен\n", this_mod_file );
   return 0;
}

static void __exit mod_exit( void ) {
```

```
   printk( "+ модуль %s выгружен\n", this_mod_file );
}
```

dec.c :

```
...
static const char dig[] = "0123456789";

long str_translate( const char *buf ) {
   long res = 0;
   const char *p = buf;
   printk( "+ %s : запрос : %s\n", this_mod_file, buf );
   while( *p != '\0' ) {
      char *s = strchr( dig, *p );
      if( s == NULL ) return -EINVAL;
      res = res * 10 + ( s - dig );
      p++;
   }
   return res;
};
...
```

hex.c :

```
...
static const char digh[] = "0123456789ABCDEF",
                  digl[] = "0123456789abcdef";

long str_translate( const char *buf ) {
   long res = 0;
   const char *p = buf;
   printk( "+ %s : запрос : %s\n", this_mod_file, buf );
   while( *p != '\0' ) {
      char *s;
      int val;
      s = strchr( digh, *p );
      if( s != NULL )
         val = s - digh;
      else {
         s = strchr( digl, *p );
         if( s == NULL ) return -EINVAL;
         val = s - digl;
      }
      res = res * 16 + val;
      p++;
   }
   return res;
};
...
```

Такая примитивная однотипность очень помогает отследить суть происходящего. Мы получили на этом шаге три идентичных модуля (`dec.ko`, `hex.ko`, `oct.ko`,):

```
$ ls -l *.ko
-rw-rw-r-- 1 olej olej  95223 Фев 12 15:13 dec.ko
-rw-rw-r-- 1 olej olej  95553 Фев 12 15:13 hex.ko
-rw-rw-r-- 1 olej olej 126348 Фев 12 15:13 master.ko
-rw-rw-r-- 1 olej olej  95223 Фев 12 15:13 oct.ko
```

Каждый из этих трёх модулей экспортирует **одно и то же** самое имя точки входа `str_translate()`. Поэтому любые два из таких модулей **не могут** быть загружены одновременно, в чём легко убедиться:

```
$ sudo insmod hex.ko
$ dmesg | tail -n30 | grep +
+ модуль hex.ko загружен
```

```
$ cat /proc/kallsyms | grep T | grep translate
c0579604 T isofs_name_translate
c063f888 T set_translate
c063f8a4 T inverse_translate
f8ab2000 T str_translate     [hex]
$ sudo insmod oct.ko
insmod: error inserting 'oct.ko': -1 Invalid module format
$ dmesg | tail -n30 | grep oct
oct: exports duplicate symbol str_translate (owned by hex)
```

Это интересный эксперимент вообще относительно экспортируемых символов ядра. Но невозможность загрузки таких модулей **одновременно** не означает невозможность их использования вообще. Нам нужно просто некоторое внешнее программное обрамление, которое сможет загружать каждый из этих модулей по требованию.

8. Функции такого программного обрамления и выполняет корневой модуль, загружающий модули-плагины и замыкающий всю конфигурацию программных компонент:

master.c :

```
#include <linux/fs.h>
#include <linux/vmalloc.h>
#include "syscall.h"
#include "../common.c"
#include "../find.c"
#include "CR0.c"

static char this_mod_file[ 40 ];      // имя файла master-модуля

static void **taddr,                  // адрес таблицы sys_call_table
            *old_sys_addr;            // адрес старого обработчика (sys_ni_syscall)

asmlinkage long (*sys_init_module)    // системный вызов sys_init_module()
                ( void __user *umod, unsigned long len, const char __user *uargs );
asmlinkage long (*sys_delete_module)  // системный вызов sys_delete_module()
                ( const char __user *name, unsigned int flags );

static long load_slave( const char* fname ) {
   long res = 0;
   struct file *f;
   long len;
   void *buff;
   size_t n;
   f = filp_open( fname, O_RDONLY, 0 );
   if( IS_ERR( f ) ) {
      printk( "+ ошибка открытия файла %s\n", fname );
      return -ENOENT;
   }
   len = vfs_llseek( f, 0L, 2 ); // 2 - means SEEK_END
   if( len <= 0 ) {
      printk( "+ ошибка lseek\n" );
      return -EINVAL;
   }
   printk( "+ длина файла модуля = %d байт\n", (int)len );
   if( NULL == ( buff = vmalloc( len ) ) ) {
      filp_close( f, NULL );
      return -ENOMEM;
   };
   printk( "+ адрес буфера чтения = %p\n", buff );
   vfs_llseek( f, 0L, 0 );       // 0 - means SEEK_SET
   n = kernel_read( f, 0, buff, len );
```

```
   printk( "+ считано из файла %s %d байт\n", fname, n );
   if( n != len ) {
      printk( "+ ошибка чтения\n" );
      vfree( buff );
      filp_close( f, NULL );
      return -EIO;
   }
   filp_close( f, NULL );
   { mm_segment_t fs = get_fs();
     set_fs( get_ds() );
     res = sys_init_module( buff, len, "" );
     set_fs( fs );
   }
   vfree( buff );
   if( res < 0 )
      printk( "+ ошибка загрузки модуля %s : %ld\n", fname, res );
   return res;
}

static long unload_slave( const char* fname ) {
   long res = 0;
   mm_segment_t fs = get_fs();
   set_fs( get_ds() );
   if( strrchr( fname, '.' ) != NULL )
      *strrchr( fname, '.' ) = '\0';
   res = sys_delete_module( fname, 0 );
   set_fs( fs );
   if( res < 0 )
      printk( "+ ошибка выгрузки модуля %s\n", fname );
   return res;
}

// новый системный вызов
asmlinkage long sys_str_translate( const char __user *buf, size_t count ) {
   static const char* slave_name[] =              // имена файлов slave-модулей
                      { "dec.ko", "oct.ko", "hex.ko" };
   static char buf_msg[ 80 ], mod_file[ 40 ], *par1;
   int res = copy_from_user( buf_msg, (void*)buf, count ), ind, trs;
   buf_msg[ count ] = '\0';
   long (*loaded_str_translate)( const char *buf );
   printk( "+ системный запрос %d байт: %s\n", count, buf_msg );
   if( buf_msg[ 0 ] == '0' ) {
      if( buf_msg[ 1 ] == 'x' ) ind = 2; // hex
      else ind = 1;                      // oct
   }
   else if( strchr( "123456789", buf_msg[ 0 ] ) != 0 )
      ind = 0;                           //dec
   else return -EINVAL;
   strcpy( mod_file, slave_name[ ind ] );
   par1 = buf_msg + ind;
   if( ( res = load_slave( mod_file ) ) < 0 ) return res;
   if( ( loaded_str_translate = find_sym( "str_translate" ) ) != NULL )
      printk( "+ адрес обработчика = %p\n", loaded_str_translate );
   else {
      printk( "+ str_translate не найден\n" );
      return -EINVAL;
   }
   if( ( trs = loaded_str_translate( par1 ) ) < 0 )
      return trs;
```

```
   printk( "+ вычислено значение %d\n", trs );
   res = unload_slave( mod_file );
   if( res < 0 ) return res;
   else return trs;
};

static int __init mod_init( void ) {
   long res = 0;
   void *waddr;
   set_mod_name( this_mod_file, __FILE__ );
   if( ( taddr = find_sym( "sys_call_table" ) ) != NULL )
      printk( "+ адрес sys_call_table = %p\n", taddr );
   else {
      printk( "+ sys_call_table не найден\n" );
      return -EINVAL;
   }
   old_sys_addr = (void*)taddr[ __NR_str_trans ];
   printk( "+ адрес в позиции %d[__NR_str_trans] = %p\n", __NR_str_trans, old_sys_addr );
   if( ( waddr = find_sym( "sys_ni_syscall" ) ) != NULL )
      printk( "+ адрес sys_ni_syscall = %p\n", waddr );
   else {
      printk( "+ sys_ni_syscall не найден\n" );
      return -EINVAL;
   }
   if( old_sys_addr != waddr ) {
      printk( "+ непонятно! : адреса не совпадают\n" );
      return -EINVAL;
   }
   printk( "+ адрес нового sys_call = %p\n", &sys_str_translate );
   if( ( waddr = find_sym( "sys_init_module" ) ) == NULL ) {
      printk( "+ sys_init_module не найден\n" );
      return -EINVAL;
   }
   printk( "+ адрес sys_init_module = %p\n", waddr );
   sys_init_module = waddr;
   if( ( waddr = find_sym( "sys_delete_module" ) ) == NULL ) {
      printk( "+ sys_delete_module не найден\n" );
      return -EINVAL;
   }
   printk( "+ адрес sys_delete_module = %p\n", waddr );
   sys_delete_module = waddr;
   rw_enable();
   taddr[ __NR_str_trans ] = sys_str_translate;
   rw_disable();
   printk( "+ модуль %s загружен\n", this_mod_file );
   return res;
}

static void __exit mod_exit( void ) {
   printk( "+ адрес syscall при выгрузке = %p\n", (void*)taddr[ __NR_str_trans ] );
   rw_enable();
   taddr[ __NR_str_trans ] = old_sys_addr;
   rw_disable();
   printk( "+ восстановлен адрес syscall = %p\n", old_sys_addr );
   printk( "+ модуль %s выгружен\n", this_mod_file );
   return;
}
```

И вот как выглядит работа модуля:

```
$ sudo insmod master.ko
$ ./syscall 0x77
syscall error -1 : Operation not permitted
$ dmesg | tail -n30 | grep +
+ адрес sys_call_table = c07ab3d8
+ адрес в позиции 223[__NR_str_trans] = c045b9a8
+ адрес sys_ni_syscall = c045b9a8
+ адрес нового sys_call = f99db024
+ адрес sys_init_module = c0470f50
+ адрес sys_delete_module = c046f4e8
+ модуль master.ko загружен
$ sudo ./syscall 077
syscall return 63
$ dmesg | tail -n30 | grep +
+ системный запрос 3 байт: 077
+ длина файла модуля = 95223 байт
+ адрес буфера чтения = f9c09000
+ считано из файла oct.ko 95223 байт
+ модуль oct.ko загружен
+ адрес обработчика = f9b83000
+ oct.ko : запрос : 77
+ вычислено значение 63
+ модуль oct.ko выгружен
$ sudo ./syscall 77
syscall return 77
$ dmesg | tail -n30 | grep +
+ системный запрос 2 байт: 77
+ длина файла модуля = 95223 байт
+ адрес буфера чтения = f9c41000
+ считано из файла dec.ko 95223 байт
+ модуль dec.ko загружен
+ адрес обработчика = f9c75000
+ dec.ko : запрос : 77
+ вычислено значение 77
+ модуль dec.ko выгружен
$ sudo ./syscall 0x77
syscall return 119
$ dmesg | tail -n30 | grep +
+ системный запрос 4 байт: 0x77
+ длина файла модуля = 95553 байт
+ адрес буфера чтения = f9c7b000
+ считано из файла hex.ko 95553 байт
+ модуль hex.ko загружен
+ адрес обработчика = f9caf000
+ hex.ko : запрос : 77
+ вычислено значение 119
+ модуль hex.ko выгружен
$ sudo ./syscall z77
syscall error -1 : Operation not permitted
$ sudo rmmod master
$ lsmod | head -n4
Module                  Size  Used by
minix                  19212  1
fuse                   48375  2
ip6table_filter         2227  0
$ dmesg | tail -n37 | grep +
+ адрес syscall при выгрузке = f99db024
+ восстановлен адрес syscall = c045b9a8
+ модуль master.ko выгружен
```

```
$ sudo ./syscall 0x77
syscall error -1 : Operation not permitted
```

В этом примере показан вывод такого количества промежуточных результатов, что дополнительно комментировать его работу нет необходимости. Может возникнуть закономерный последний вопрос: а где же здесь возможность дополнять в проект модули-плагины, не затрагивая код корневого модуля? Её здесь в явном виде нет. Но стоит вынести соответствие критерия выбора (переменная `ind`) используемого плагина к **имени файла** модуля (массив `slave_name[]`) в текстовый конфигурационный файл, а читать такие файлы мы научились несколькими разделами ранее — и вы получаете динамически расширяемую систему. Это элементарно просто, а не сделано это в и так предельно громоздком примере только чтобы его не усложнять дополнительно.

Обсуждение

Весь этот раздел о нетривиальных возможностях модулей ядра написан, конечно, не в намерении поощрить и развить хакерские наклонности читателей в написании вирусов или другого вредоносного программного обеспечения — в этом вам воспрепятствует, в первую очередь, требование наличия привилегий `root` для операций с модулями и защищённость самой операционной системы (это вам не Windows!). Показаны эти возможности, и выделены в отдельный, финальный раздел, чтобы подвести итоги нашему рассмотрению, и прийти к пониманию того, что:

1. В пространстве ядра можно выполнить **практически всё**! Модули являются полноценной составной частью ядра, поэтому сказанное относится и к ним в полной мере. (Представьте себе: как могли бы существовать в пользовательском пространстве возможности, недостижимые в ядре, если **все** такие возможности тому же пользовательскому пространству предоставляет ядро.)

2. Код модуля выполняется в привилегированном режиме (режим супервизора, кольцо защиты 0 для x86 архитектуры), поэтому ему доступны любые операции, которые недоступны пользовательскому коду: привилегированные операции, работа с управляющими регистрами процессора (`cr0` — `cr4` для x86), работа с таблицами страниц, со структурами MMU и многое другое.

3. Раз существует такая дуальность возможностей для процессов и для ядра, то и API для использования таких возможностей столь же дуальны. Для пользовательских процессов это POSIX API, а для ядра — API ядра. То и другое отличаются по форме (имена вызовов и структур данных, прототипы вызовов, число параметров и др.), но подобны друг другу. Если при написании модулей у вас возникают затруднения — ищите аналогии в POSIX API! (Это особенно хорошо было видно на примерах чтения файлов из ядра.)

4. И, конечно, дотошное изучение заголовочных файлов в `ls /lib/modules/`uname -r`/build/include` и обширных документальных заметок в подкаталоге `Documentation` дерева исходных кодов вашего ядра. Это одно уже должно быть достаточным мотивом для скачивания исходных кодов своего ядра, даже если вы вовсе не собираетесь его пересобирать.

5. Наконец, одну и ту же функциональность в коде можно реализовать, обычно, несколькими совершенно разными способами. Для реализаций ваших фантазий в пространстве ядра существует **больше** альтернатив, чем в пространстве пользователя. Хорошо показателен в этом смысле обсуждавшийся ранее пример открытия и чтения именованного файла. Эта задача, как пример, может быть реализована, как минимум, 4-мя различными способами (мы сейчас не обсуждаем эффективность и предпочтительность любого из них):

– открытие `filp_open()` с последующим чтением с помощью `kernel_read()`;

– открытие `filp_open()` с последующим чтением с помощью `vfs_read()`;

– открытие системным вызовом `sys_open()` с последующим чтением с помощью системного вызова

`sys_read()`, при том, что системные вызовы осуществляются через команду `int 0x80`, или её эквиваленты;

– открытие вызовом функции обработчика системным вызова `sys_open()` с последующим чтением также с помощью вызовом функции обработчика системного вызова `sys_read()`, при том, что адреса функций обработчиков находятся как не экспортируемые символы ядра;

И даже эти предложенные варианты — это далеко не всё, что можно применить как альтернативные способы реализации абсолютно одной и той же функциональности в ядре.

6. На этом примере уместно, наверное, обратить внимание, что постоянно повторяемую из одной публикации по ядру в другую фразу о том, что в коде (модулей) ядра недоступны для использования библиотеки (разделяемые, `.so`), в частности, и стандартная библиотека языка C (POSIX), следует понимать именно **узко технологически**: библиотеки недоступны **как формат** представления данных. Но функциональность **кода**, реализующего библиотечные вызовы, вполне доступны и для кода модуля ядра, как это показывалось выше. Таким образом, **весь** API системных вызовов Linux также доступен, при некоторой изобретательности, в коде модуля, в тех случаях, конечно, когда эти вызовы обладают каким-то смыслом в контексте ядра: вызов `sys_getpid()` можно выполнить из ядра, но, в большинстве случаев, возвращаемое им значение будет сложно интерпретировать, часто это будет просто «мусор».

Задачи

1. Сделайте модуль, который читает свои конфигурационные значения из файла. Пусть модуль читает при загрузке из конфигурационного файла произвольное число каналов данных, каждый из которых записан конфигурационной строкой вида `chan=dev123`, где `dev123` — это будет имя устройства `/dev/dev123`, в которое можно писать данные произвольной длины, а потом читать оттуда записанное.

2. Проанализируйте, в каких случаях (достаточно редких) модулю может понадобиться а). читать из файла, б). писать в файл.

3. Как рассмотреть все имена API, экспортируемые непосредственно ядром (не учитывая символов, экспортируемых другими динамически загруженными модулями)? Подсчитайте число таких имён в вашей системе.

4. Сделайте реакцию на аппаратное прерывание (IRQ), обрабатываемую в пространстве пользователя: по IRQ модуль посылает сигнал приложению, который позже осуществляет реакцию.

5. То же, что и в предыдущем случае, но ядро должно уведомлять приложение через широковещательное сообщение протокола `netlink`.

6. Замените (или расширьте своим кодом, что разумнее) один из системных вызовов ядра Linux.

7. Добавьте свой собственный системный вызов к работающему ядру (динамически).

8. Сделайте свои собственные приложения, загружающее и выгружающее, соответственно, указанный модуль ядра (так как это делают `insmod` и `rmmod`).

9. Сделайте модуль с динамически подключаемыми **плагинами**, на манер того, как это делает `mount` для разнообразных файловых систем.

13. Отладка в ядре

Процесс отладки модулей ядра намного сложнее отладки пользовательских приложений. Это обусловлено целым рядом особенностей и окружения работы модулей ядра:

– Код ядра представляет собой набор функциональных возможностей, не связанных ни с каким конкретным процессом, многие из этих возможностей выполняются параллельно и в независимых потоках от наблюдаемого (в модуле).

– Код модуля не может в полной мере быть выполнен под отладчиком, не может легко трассироваться; многие ядерные механизмы принципиально существуют только во временных зависимостях и не могут быть приостановлены или заторможены.

– Даже при использовании интерактивных отладчиков (об этом детально далее), становится возможен динамический **контроль** значений и состояний (диагностика), но практически никогда невозможно **изменение** значений для наблюдения их поведения, как это практикуется в пользовательском пространстве с использованием `gdb`; эта особенность обуславливается не технологическими сложностями отладчиков, а уровнем последствий для операционной системы в результате таких вмешательств.

– Ошибки, возникающие в коде ядра может оказаться чрезвычайно трудно **воспроизвести**, повторить ситуацию для анализа и наблюдения.

– Поиском ошибок ядра можно легко сломать всю систему, и тем самым уничтожить и большую часть данных, которые и использовались для их поиска.

Ещё одна сложность отладки в пространстве ядра, на этот раз уже не технического свойства, состоит в том, что команда разработчиков ядра Linux крайне негативно относится вообще к идее интерактивных отладчиков для их ядра. Это их принципиальная философия. Мотивируется это тем, что при наличии и использования развитых интерактивных отладчиков для ядра будет возрастать «лёгкость» в отношении решений, принимаемых к ядру, и это приведёт к накоплению ошибок в ядре. В любом случае, существовало и существует целый ряд проектов интерактивных отладчиков для их ядра, но ни один из них не признан как «официальный», многие из них появляются и через некоторое время затухают.

В итоге: отладка кода ядра — это, скорее, может быть набор эмпирических трюков и рекомендаций, но не слаженная технология. Некоторый минимальный набор таких трюков и рекомендаций мы и рассмотрим далее.

Отладочная печать

Как бы этого, возможно, кому-то бы и не хотелось признать, основным способом отладки модулей ядра было и остаётся использование вызова отладочного вывода `printk()`. Использование `printk()` — это самый универсальный способ работы по отладке. Детали использования `printk()` и настройки демонов системного журнала - рассматривались ранее. Тексты сообщений не должны использовать символы вне таблицы ASCII, в частности, недопустимо использовать русские буквы в любой кодировке.

Если не проявлять известную осторожность, можно получить тысяч сообщений, созданные выполнением `printk()`, переполняющие текстовую консоль, или файл системного журнала, или даже полностью диск журналирования — в этом нет ничего страшного, но такой обширный вывод не подлежит никакому анализу и является совершенно бессмысленной тратой времени.

Интерактивные отладчики

Во-первых, для отладочных целей в ядре можно использовать общеизвестный отладчик gdb, но только для целей **наблюдения**. Но даже это является непростой в организации задачей, если мы собираемся динамически исследовать внутренности своего подгружаемого модуля, а не вообще копаться в коде самого ядра (что вообще не затрагивается по ходу всего нашего рассмотрения). Для запуска gdb используем команду:

```
# gdb /usr/src/linux/vmlinux /proc/kcore
...
```

Здесь первый параметр указывает пересобранный образ ядра (несжатый, а загружаемый образ вашей системы, находящийся, например, по имени /boot /vmlinuz — это сжатый образ), а второй параметр — это имя файла ядра, формируемомого динамически. Но для работы с модулем этого мало: отладчик ничего не знает о модуле! Мы можем получить **статически** информацию о **текущей** загрузке модуля, и предоставить её gdb. Сделаем это так:

```
$ sudo insmod ./hello_printk.ko
```

- ядро должно быть собрано с опцией CONFIG_DEBUG_INFO ...

- при этом в каталоге /sys/module/hello_printk/sections находятся файлы .text, .bss, .data, содержащие адреса начала загрузки секций кода, инициализированных и неинициализированных данных, соответственно.

- используя считанные из них значения, выполним команду в оболочке gdb (запущенной как показано было выше):

```
(gdb) add-symbol-file ./hello_printk.ko 0xd0832000 -s .bss 0xd0837100 -s .data 0xd0836be0
add symbol table from file "hello_printk.ko" at
    .text_addr = 0xd0832000
    .bss_addr = 0xd0837100
    .data_addr = 0xd0836be0
(y or n) y
Reading symbols from scull.ko...done.
...
```

Вот после столь хлопотных действий мы имеем в gdb информацию о нашем модуле и получаем возможность наблюдения за переменными — как я могу оценивать, в меру своих предпочтений, возможности отнюдь не адекватные затраченным усилиям...

Помимо gdb, существует целый ряд независимых проектов, ставящих своей целью отладку для ядра. Но, как уже было сказано: а). все такие проекты носят «инициативный» характер, и б). все они имеют изрядные ограничения в своих возможностях (что связано вообще с принципиальной сложностью отладки в ядре ..., но все эти проекты активно развиваются). Только коротко перечислим такого рода инструменты, детальное их использование оставим для энтузиастов на самостоятельную проработку:

- Встроенный отладчик ядра kdb, являющийся неофициальным патчем к ядру (доступен по адресу http://oss.sgi.com - Silicon Graphics International Corp.). Для использования kdb необходимо взять патч, в версии, в точности соответствующей версии отлаживаемого ядра, применить его и пересобрать и переустановить ядро. В настоящее время существует только для архитектуры IA-32 (x86).

- Патч kgdb, находящийся даже в дереве исходных кодов ядра; эта технология поддерживает удалённую отладку с другого хоста, соединённого с отлаживаемым последовательной линией, или через сеть Ethernet; в кодах ядра можно найти некоторые описания: Documentation/i386/kgdb.

- Независимый проект под тем же именем продукта kgdb (доступен по адресу http://kgdb.linsyssoft.com), эта версия не поддерживает удалённую отладку по сети.

Нужно иметь в виду, что оба названных выше продукта `kgdb` имеют очень ограниченный спектр поддерживаемых процессорных платформ, из числа тех, на которых работает Linux, реально это x86 и PPC. Ряд самых интересных на сегодня платформ никак не затрагиваются этими средствами.

Отладка в виртуальной машине

Весьма продуктивной оказывается отладка модулей в среде виртуальной машины (VM). В этом направлении у автора есть изрядный положительный опыт, полученный с использованием таких динамично развивающихся проектов виртуальных машин QEMU (свободный проект: http://wiki.qemu.org) и VirtualBox (также основанный на коде виртуализации из QEMU проект от Sun Microsystems, ныне от Oracle: http://www.oracle.com/technetwork/server-storage/virtualbox/downloads/index.html или независимый ресурс проекта https://www.virtualbox.org/wiki/Downloads). Хотя, естественно, может использоваться и любой другой проект для создания виртуальных сред выполнения: XEN, Bochs и т.д.— эта область продуктов очень быстро изменяется и прогрессирует.

Отладка в среде виртуальной машины (естественно, с учётом всех минусов, привносимых любым моделированием) создаёт целый ряд дополнительных преимуществ:

- Отработка модуля ядра производится в изолированном окружении, нет риска разрушения базовой операционной системы и необходимости постоянных перезагрузок, что сильно замедляет темп проекта.

- Простота связи (загрузка модуля, наблюдение результатов) со средой разработки по внутренней TCP/IP виртуальной сети на основе тунельного интерфейса Linux. Для проектов сетевых модулей всегда может создаваться несколько **виртуальных** сетевых интерфейсов — при разрушении отлаживаемого интерфейса всегда сохраняется возможность доступа по резервным, и восстановления работоспособности среды.

- Возможность использования отладчика `gdb` в базовой (хостовой) системе, для наблюдения «извне» за процессами, происходящими в виртуальной машине. Особенно гибко это реализовано и используется в среде QEMU.

- Возможность ведения разработки для иных процессорных архитектур (ARM, PPC, MIPS) на развитой рабочей станции x86 с наличием обширного инструментария (эта возможность — только для QEMU, VirtualBox поддерживает только x86 архитектуру).

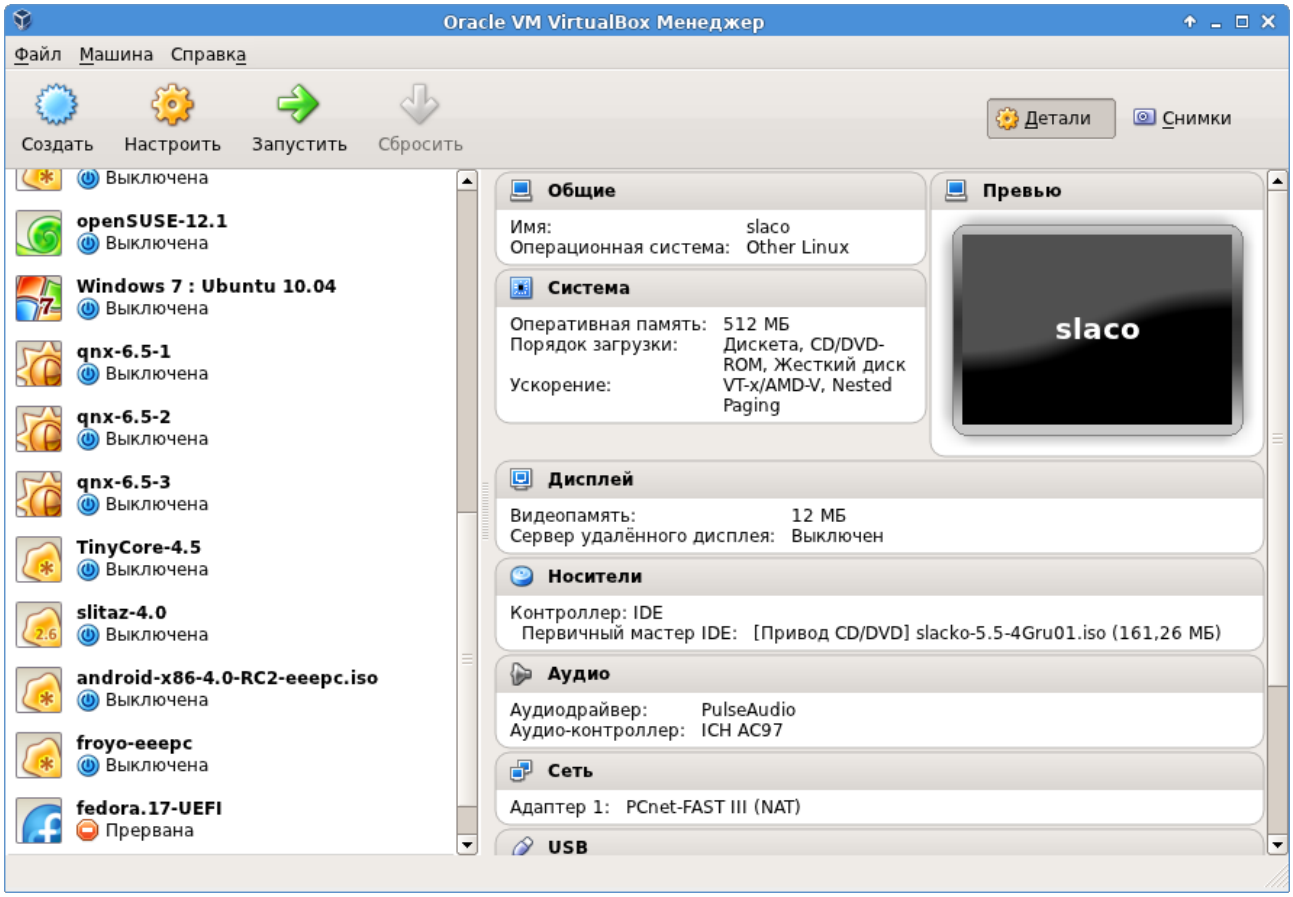
- В единой среде виртуализации может быть создано много виртуальных машин с разными инсталлированными версиями (сигнатурами) ядра. Это позволяет проводить тестирование и обкатку промышленного проекта (накануне сдачи) одновременно для широкого спектра версий ядра. (Интересно отметить, что на хостовой машине с объёмом RAM, скажем, 4Gb, одновременно могут **выполняться**, например, 10-12 VM, **каждой** из которых отведено по 1Gb RAM, при этом хостовая машина всё ещё будет достаточно активно реагировать на интерфейс работы с пользователем.)

- В среде виртуализации может быть установлено много различных дистрибутивов Linux со всеми свойственными им различиями. Помимо этого, там же могут быть установлены VM и отличающихся операционных систем, как это и имеет место на показанном рисунке ниже: не POSIX операционные системы Windows, или POSIX но не Linux микроядерная операционная система QNX. Это позволяет вести отработку платформенно независимых компонент проекта.

Хотелось бы специально акцентировать предпоследний пункт, не совсем очевидный, приведенного перечисления. Вспомним, что модуль скомпилированный в одном ядре, неработоспособен в отличающемся ядре (даже просто **по написанию** сигнатуры, имени ядра в команде: `uname -r`). Но, из-за изменчивости API ядра, о которой уже много сказано, ваш модуль может вообще даже не компилироваться в следующем по номеру ядре (изменение состава функций API, изменения их прототипов, типов параметров и многое другое). А вот как-раз подготовленный заранее тестовый набор виртуальных машин для последовательного набора ядер, позволяет

оттестировать и отработать разрабатываемый модуль, подготовить наиболее гибкую его промышленную поставку (за счёт, например, использования препроцессорных `#defined` относительно версий ядер в коде модуля, что и показывалось в некоторых приводившихся ранее примерах).

Из названных двух близких систем виртуализации: QEMU является более гибким и универсальным инструментом, но VirtualBox имеет более дружественные инструменты конфигурирования и управления виртуальными машинами. О технике отладки в виртуальной среде, особенно на кроссовых платформах, можно и должно сказать очень много, но это уже предмет отдельного большого разговора.



Следует напомнить, что в силу **сетевой природы** графического протокола X11 в UNIX, любое графическое приложение Linux может быть запущено на удалённом сетевом хосте, с отображением GUI приложения на локальном хосте. Не является исключением и VirtualBox, что открывает возможности для групповой работы над проектом в единой среде удалённой виртуальной машины — это создаёт существенно новые возможности. Добиться удалённого запуска X11 приложения можно, например (это не единственный способ) посредством тунелирования X11 протокола над протоколом SSH (опция `-X`). Так был запущен и сеанс, показанный выше на рисунке:

```
$ ssh -X olej@192.168.1.9
olej@192.168.1.9's password:
Last login: Thu Nov 14 15:56:30 2013 from 192.168.1.20
$ VirtualBox &
[1] 2549
```

Отдельные отладочные приёмы и трюки

Здесь мы перечислим некоторые мелкие приёмы применяемые в процессе отладки, которые сложились и показали свою продуктивность в процессе работ над реальными разработками в области модулей ядра.

Модуль исполняемый как разовая задача

Один из продуктивных трюков, который уже неоднократно применялся по ходу всего рассмотрения ранее, есть сознательное написание модуля, возвращающего ненулевое значение из инициализирующей функции, который вовсе и «не собирается» загружаться. Такой модуль выполняется однократно, подобно пользовательскому процессу (начинающемуся с `main()`), но отличаясь тем, что делает он это в супервизорном режиме (с полными привилегиями) и в адресном пространстве ядра. Пример такого простейшего модуля приводится в архиве `simple-debug.tgz`):

md.c :

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

static int __init hello_init( void ) {
   extern int sys_close( int fd );
   void* Addr;
   Addr = (void*)sys_close;
   printk( KERN_INFO "sys_close address: %p\n", Addr );
   return -1;
}

module_init( hello_init );
```

Такой модуль в принципе не может загрузиться, так как он возвращает -1 (или точнее: не 0). В этой связи у модуля даже нет процедуры завершения (она ему не нужна):

```
$ sudo /sbin/insmod ./md.ko
insmod: error inserting './md.ko': -1 Operation not permitted
```

Но такой модуль начинает выполняться (`hello_init()`), выполняться в контексте ядра, и производит диагностический вывод:

```
$ dmesg | tail -n2
md: module license 'unspecified' taints kernel.
sys_close address: c047047a
```

И в таком качестве подобный модуль (который не загрузится, но и не навредит) становится интересным средством отладки, особенно на начальных этапах отработки, когда можно проверить все инициализированные значения модуля и используемых им экспортируемых переменных ядра.

Тестирующий модуль

При организации модульного тестирования (unit testing) разработчик может столкнуться с недоумением, с тем как оформлять тесты, ведь создаваемый код модуля не может быть скомпилирован для работы в пользовательском режиме. Но в этом случае в коде проектируемого модуля могут быть созданы **экспортируемые** точки входа вида `test_01(), test_02(), ... test_MN()`, а для последовательного вызова тестовых входов создан отдельный тестирующий модуль, использующий показанный ранее трюк (разовое исполнение), весь код которого умещается в единственную функцию инициализации... Пример такой реализации упрощённой до предела показан в том же архиве `simple-debug.tgz`:

md1.h :

```
#include <linux/module.h>
MODULE_LICENSE( "GPL" );
```

```
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
extern char* test_01( void );
extern char* test_02( void );
static int __init init( void );
module_init( init );
```

md1.c :

```
#include "md1.h"
static char retpref[] = "this string returned from ";

char* test_01( void ) {
   static char res[ 80 ];
   strcpy( res, retpref );
   strcat( res, __FUNCTION__ );
   return res;
};
EXPORT_SYMBOL( test_01 );

char* test_02( void ) {
   static char res[ 80 ];
   strcpy( res, retpref );
   strcat( res, __FUNCTION__ );
   return res;
};
EXPORT_SYMBOL( test_02 );

static int __init init( void ) {
   return 0;
}
static void __exit exit( void ) {}
module_exit( exit );
```

А это — полный код тестирующего модуля, который мы пишем, как описывали выше, для однократного выполнения:

mt1.c :

```
#include "md1.h"
static int __init init( void ) {
   printk( "%s\n", test_01() );
   printk( "%s\n", test_02() );
   return -1;
}
```

И вот как выглядит выполнение последовательности тестов проектируемого модуля:

```
$ sudo insmod md1.ko
$ sudo insmod mt1.ko
insmod: error inserting 'mt1.ko': -1 Operation not permitted
$ dmesg | tail -n2
this string returned fron test_01
this string returned fron test_02
```

Интерфейсы пространства пользователя к модулю

Для контроля значений ключевых переменных (и даже их изменений) внутри модуля — их можно отобразить в псевдофайловые системы `/proc`, а ещё лучше `/sys`. Это часто делается, например, для счётчика обработанных в драйвере прерываний, как это показано в примере ниже (попутно показано, что таким способом можно контролировать переменные даже внутри обработчиков аппаратных прерываний):

mdsys.c :

```
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/interrupt.h>
#include <linux/version.h>

#define SHARED_IRQ 16                // my eth0 interrupt line
static int irq = SHARED_IRQ;
module_param( irq, int, S_IRUGO );    // may be change

static unsigned int irq_counter = 0;
static irqreturn_t mdsys_interrupt( int irq, void *dev_id ) {
   irq_counter++;
   return IRQ_NONE;
}

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t show( struct class *class, struct class_attribute *attr, char *buf ) {
#else
static ssize_t show( struct class *class, char *buf ) {
#endif
   sprintf( buf, "%d\n", irq_counter );
   return strlen( buf );
}

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t store( struct class *class, struct class_attribute *attr, const char *buf, size_t c
#else
static ssize_t store( struct class *class, const char *buf, size_t count ) {
#endif
   int i, res = 0;
   const char dig[] = "0123456789";
   for( i = 0; i < count; i++ ) {
      char *p = strchr( dig, (int)buf[ i ] );
      if( NULL == p ) break;
      res = res * 10 + ( p - dig );
   }
   irq_counter = res;
   return count;
}

CLASS_ATTR( mds, 0666, &show, &store ); // => struct class_attribute class_attr_mds
static struct class *mds_class;
static int my_dev_id;

int __init init( void ) {
   int res = 0;
   mds_class = class_create( THIS_MODULE, "mds-class" );
   if( IS_ERR( mds_class ) ) printk( KERN_ERR "bad class create\n" );
   res = class_create_file( mds_class, &class_attr_mds );
   if( res != 0 ) printk( KERN_ERR "bad class create file\n" );
   if( request_irq( irq, mdsys_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id ) )
      res = -1;
   return res;
}

void cleanup( void ) {
   synchronize_irq( irq );
   free_irq( irq, &my_dev_id );
   class_remove_file( mds_class, &class_attr_mds );
```

```
    class_destroy( mds_class );
    return;
}

module_init( init );
module_exit( cleanup );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_DESCRIPTION( "module in debug" );
MODULE_LICENSE( "GPL v2" );
```

Этот модуль получился прямой комбинацией нескольких примеров, которые мы написали раньше, так что все механизмы нам знакомы. Обработка ошибок при установке модуля практически отсутствует, чтобы не загромождать текст.

Для проверки того как это работает, загрузим модуль для контроля линии IRQ, например, сетевого адаптера (хотя это с таким же успехом могла бы быть и линия системного таймера):

```
$ cat /proc/interrupts | grep eth
  16:      34985          0   IO-APIC-fasteoi   i915, eth0
$ sudo insmod mdsys.ko irq=16
$ cat /sys/class/mds-class/mds
280
$ cat /sys/class/mds-class/mds
301
$ cat /sys/class/mds-class/mds
353
```

Здесь мы контролируем нарастающее значение счётчика сработавших прерываний. Изменим начальное значение этого счётчика, от которого происходит инкремент:

```
$ echo 10 > /sys/class/mds-class/mds
$ cat /sys/class/mds-class/mds
29
$ sudo rmmod mdsys
```

Подобным образом мы можем «вытащить» в наружу модуля сколь угодно много переменных для диагностики и управления.

Комплементарный отладочный модуль

Весьма часто техника создания интерфейсов в пространство `/proc` или `/sys`, как это описано выше, является совершенно приемлемой, но после завершения работ было бы нежелательно оставлять конечному пользователю доступ к диагностическим и управляющим переменным, хотя бы из тех соображений, что таким образом сохраняется возможность очень просто разрушить нормальную работу изделия[29]. Но переписывать код модуля перед его сдачей — это тоже мало приемлемый вариант, так как всякой редактурой можно внести существенные ошибки в код модуля. В этом случае для проектируемого модуля на период отладки может быть создан парный ему (комплементарный) модуль:

- проектируемый модуль теперь не выносит критические переменные в качестве органов диагностики в файловые системы, а только объявляет их экспортируемыми:
- комплементарный отладочный модуль динамически устанавливает связь с этими переменными (импортирует) при своей загрузке...
- и создаёт для них интерфейсы (в `/proc`, `/sys`) по диагностическим (читаемым) и управляющим (записываемым) переменным;

29 И создаём себе реальную возможность нарваться на рекламацию.

– после завершения отладки отладочный модуль просто изымается из проекта.

Чтобы увидеть в деталях о чём речь, трансформируем в эту схему пример, описанный в предыдущем разделе... Причём сделаем это без всяких изменений и улучшений — полный эквивалент, чтобы мы могли сравнить исходники по принципу: что было и что стало?

Файл общих определений:

mdsys2.h :

```
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/interrupt.h>
#include <linux/version.h>

extern unsigned int irq_counter;
int __init init( void );
void __exit cleanup( void );
module_init( init );
module_exit( cleanup );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_DESCRIPTION( "module in debug" );
MODULE_LICENSE( "GPL v2" );
```

Собственно проектируемый (отлаживаемый) модуль:

mdsys2.c :

```
#include "mdsys2.h"

#define SHARED_IRQ 16               // my eth0 interrupt
static int irq = SHARED_IRQ;
module_param( irq, int, S_IRUGO );   // may be change

unsigned int irq_counter = 0;
EXPORT_SYMBOL( irq_counter );
static irqreturn_t mdsys_interrupt( int irq, void *dev_id ) {
   irq_counter++;
   return IRQ_NONE;
}

static int my_dev_id;
int __init init( void ) {
   if( request_irq( irq, mdsys_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id ) )
      return -1;
   else
      return 0;
}

void cleanup( void ) {
   synchronize_irq( irq );
   free_irq( irq, &my_dev_id );
   return;
}
```

И модуль, создающий для него отладочный интерфейс:

mdsysc.h :

```
#include "mdsys2.h"

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
```

```
static ssize_t show( struct class *class, struct class_attribute *attr, char *buf ) {
#else
static ssize_t show( struct class *class, char *buf ) {
#endif
   sprintf( buf, "%d\n", irq_counter );
   return strlen( buf );
}

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t store( struct class *class, struct class_attribute *attr, const char *buf, size_t c
#else
static ssize_t store( struct class *class, const char *buf, size_t count ) {
#endif
   int i, res = 0;
   const char dig[] = "0123456789";
   for( i = 0; i < count; i++ ) {
      char *p = strchr( dig, (int)buf[ i ] );
      if( NULL == p ) break;
      res = res * 10 + ( p - dig );
   }
   irq_counter = res;
   return count;
}

CLASS_ATTR( mds, 0666, &show, &store ); // => struct class_attribute class_attr_mds
static struct class *mds_class;

int __init init( void ) {
   int res = 0;
   mds_class = class_create( THIS_MODULE, "mds-class" );
   if( IS_ERR( mds_class ) ) printk( KERN_ERR "bad class create\n" );
   res = class_create_file( mds_class, &class_attr_mds );
   if( res != 0 ) printk( KERN_ERR "bad class create file\n" );
   return res;
}

void cleanup( void ) {
   class_remove_file( mds_class, &class_attr_mds );
   class_destroy( mds_class );
   return;
}
```

Теперь отладочный модуль не знает ничего ни о прерываниях, ни о структуре отлаживаемого модуля — он знает только ограниченный набор экспортируемых переменных (или, как вариант, экспортируемых точек входа), по именам и по типам. Опробуем то, что у нас получилось, и сравним с примером предыдущего раздела:

```
$ sudo insmod mdsys2.ko
$ sudo insmod mdsysc.ko
$ lsmod | head -n3
Module                  Size  Used by
mdsysc                   934  0
mdsys2                   844  1 mdsysc
$ cat /sys/class/mds-class/mds
784
$ cat /sys/class/mds-class/mds
825
$ echo 0 > /sys/class/mds-class/mds
$ cat /sys/class/mds-class/mds
21
```

Теперь мы удалим отладочный модуль:

```
$ sudo rmmod mdsysc
```

Отлаживаемый модуль замечательно продолжает работать, но все отладочные интерфейсы к нему исчезли:

```
$ lsmod | head -n3
Module                  Size  Used by
mdsys2                   844  0
lp                      6794  0
$ cat /sys/class/mds-class/mds
cat: /sys/class/mds-class/mds: Нет такого файла или каталога
$ sudo rmmod mdsys2
```

Пишите в файлы протоколов...

У вас всегда остаётся возможность писать отладочные сообщения из модуля ядра в собственный **файл протокола** выполнения, который позже доступен для детального анализа. Имеется в виду свой **собственный** файл **данных**, который создаёт и пишет модуль. При этом вы ничем не ограничены в степени детализации сообщений, направляемый в файл протокола. Саму технику такой записи в файл мы рассмотрели ранее, при рассмотрении работы модуля с файлами данных (архив `file.tgz` в примерах).

Некоторые мелкие советы в завершение

Чаще перезагружайте систему!

Отладка модулей ядра отличается от отладки пользовательского пространства тем, что очередное аварийное завершение теста модуля может оставлять «следы» в ядре, создавая тем малозаметные (или поздно обнаруживаемые) аномалии в поведении системы. Особенно часто это наблюдается, например, при отработке интерфейсов драйвера в файловую систему `/proc`.

Побочные эффекты от накопленных ошибок в ядре системы могут доходить до того состояния, что при дальнейших улучшениях отрабатываемого кода, даже компилятор `gcc` станет сообщать вам о каких-то загадочных внутренних ошибках компилятора... Это уже явная народная примета того, что ... пришла пора перезагружаться!

Для того, чтобы избежать десятков часов **бездарно** потерянного времени, при работе над модулями, перезагружайте время от времени ваш инструментальный Linux, даже если вам кажется, что он совершенно нормально работает. После перезагрузки результаты повторения только-что выполненного теста могут радикально поменяться!

Используйте естественные POSIX тестеры

Здесь я имею в виду, что при отработке модуля всегда, прежде, чем начинать более жёсткое тестирование драйвера, проверьте его реакцию по чтению и запись на естественные POSIX тестеры: `cat` для чтения, `echo` для записи и подобные ним. В этом качестве могут быть полезны и другие стандартные утилиты Linux, например `cp`. Возможно, для обеспечения совместимости функционирования совместно с POSIX командами, вам потребуется добавить к драйверу дополнительную функциональность (например, отработка ситуации `EOF`), которая и не требуется конечными спецификациями на продукт. Но получение полной POSIX совместимости для вашего продукта стоит затраченного дополнительного труда!

Тестируйте чтение сериями

Выполняя проверку операций `read()`, не ограничивайтесь одиночной операцией тестирования. Вместо этого проверяйте серию последовательных операций тестирования. Этим вы страхуетесь, что ваш драйвер не

только нормально отрабатывает операцию, но и нормально восстанавливается после операции и готов к выполнению следующей. Другими словами, вместо одиночной операции `cat` (в простейшем случае) делайте несколько последовательных, сверяя их идентичность:

```
$ cat /dev/xxx
REZULT
$ cat /dev/xxx
REZULT
$ cat /dev/xxx
REZULT
```

Подобное можно было не раз видеть на протяжении предыдущих показанных тестов. То же имеет место и в отношении к операциям записи, но в значительно меньшей степени.

Задачи

1. Сделайте отладочный (комплементарный) модуль к разрабатываемому (любому), который отображает внутренние переменные в `/sys` и позволяет записывать в них новые значения.

Заключение

«Нельзя объять необъятное»
Козьма Прутков.

Есть ещё множество механизмов, API и трюков, которые используются в программировании ядра. Их просто нет возможности описать в любом издании обозримого объёма — для сравнения обратитесь к POSIX API пользовательского пространства, которое описывают тысячи и тысячи страниц публикаций... А API ядра должно иметь и имеет аналоги практически всех механизмов, предоставляемых в пространстве пользователя.

Существует некоторое предубеждение, что программирование в ядре, и в частности модулей ядра, требуют особого аскетизма в выборе используемых возможностей, и вообще весьма ограничено в том, что вам при этом доступно. Целью моих примеров было показать: при программировании в технике модулей ядра вам доступны **все возможности**, о которых вы слышали из POSIX ... плюс ещё изрядное количество сверх того (например доступ к управляющим регистрам процессора и привилегированным командам). Некоторые ограничения на этом пути составляет отсутствие внятных описания относительно API ядра, но это ограничение преодолевается дотошным изучением («верить никому нельзя») и изобретательным экспериментированием.

Основную (а временами и единственную) помощь в поиске адекватных нашим намерениям API ядра даёт рассмотрение открытых исходных кодов ядра, и, главным образом, **заголовочных файлов** определений кода ядра (с чего и нужно начинать рассмотрение).

Приложения

Приложение А: Краткая справка по утилите make

При модульном программировании работать с утилитой make приходится постоянно. Более того, «работать» это сильно мягко сказано: приходится постоянно переписывать сценарный файл Makefile, причём для довольно изощрённых случаев. Детальное описание make доступно [22]. Здесь же приведём только самую краткую справку (главным образом для напоминания о умалчиваемых значениях переменных make).

Утилита make существует в разных ОС, из-за особенностей выполнения, наряду с «родной» реализацией во многих ОС присутствует GNU реализация gmake, и поведение этих реализаций может достаточно существенно отличаться, поэтому совсем не лишним бывает проверить с чем мы имеем дело:

```
$ make --version
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
...
```

Утилита make автоматически определяет какие части большой программы должны быть перекомпилированы в зависимости от произошедших изменений, и выполняет необходимые для этого действия. Изменения (обновления) фиксируются исключительно по датам последних модификаций файлов. На самом деле, область применения make не ограничивается только сборкой программ. Её можно использовать ее для решения любых задач, где одни файлы должны автоматически обновляться при изменении других файлов.

Многократно выполняемая сборка приложений проекта, с учётом зависимостей и обновлений, делается утилитой make, которая использует оформленный сценарий сборки. По умолчанию имя файла сценария сборки - Makefile. Утилита make обеспечивает полную сборку одной указанной **цели** в сценарии сборки, например:

```
$ make
$ make clean
```

Если цель не указывается, то выполняется **первая последовательная** цель в файле сценария (почему-то существует суеверие, что собирается цель с именем all — просто цель all ставится в файле часто выше всех остальных, вот она по умолчанию и собирается). Может использоваться и любой другой сценарный файл сборки, тогда он указывается так:

```
$ make -f Makefile.my
```

Сценарий Makefile состоит из синтаксических конструкций всего двух типов: целей и макроопределений. Описание цели состоит из трех частей: а). имени цели, б). списка зависимостей и в). списка команд интерпретатора shell, требуемых для построения цели. Имя цели — непустой список имён файлов, которые предполагается создать. Список зависимостей — список имён файлов, в зависимости от которых строится цель. Имя цели и список зависимостей составляют заголовок цели, записываются в одну строку и разделяются двоеточием (':'). Список команд записывается со следующей строки, причем все команды начинаются **с обязательного символа табуляции**. Любая строка в последовательности списка команд, не начинающаяся с табуляции (ещё одна, следующая команда) или '#' (комментарий) — считается завершением текущей цели и началом новой.

Утилита make имеет множество умалчиваемых значений (переменных, суффиксов, ...), важнейшими из которых являются правила обработки суффиксов, а также определения внутренних переменных окружения. Эти данные называются базой данных make и могут быть рассмотрены (объём вывода очень велик, поэтому смотрим его через файл):

```
$ make -p >make.suffix
make: *** Не заданы цели и не найден make-файл.  Останов.
$ cat make.suffix
# GNU Make 3.81
```

```
# Copyright (C) 2006  Free Software Foundation, Inc.
...
# База данных Make, напечатана Thu Apr 14 14:48:51 2011
...
CC = cc
LD = ld
AR = ar
CXX = g++
COMPILE.cc = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
COMPILE.C = $(COMPILE.cc)
...
SUFFIXES := .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l .s .S .mod .sym .def .h .info .dvi
.tex .texinfo .texi .txinfo .w .ch...
# Implicit Rules
...
%.o: %.c
#  команды, которые следует выполнить (встроенные):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
...
```

Все эти значения (переменных: `CC, LD, AR, EXTRA_CFLAGS, ...`) могут использоваться файлом сценария как неявные определения с значениями по умолчанию. Кроме этого, вы можете определить и свои правила обработки по умолчанию для указанных вами суффиксов (расширений файловых имён), как это показано на примере выше для исходных файлов кода на языке C: `*.c`.

Как существенно ускорить сборку make

На сегодня, когда практически не осталось в обиходе (или выходят из обращения) однопроцессорных (одноядерных) настольных компьютеров, сборку многих проектов можно значительно (в разы) ускорить, используя умение `make` запускать несколько заданий в параллель (ключ `-j`):

```
$ man make
...
    -j [jobs], --jobs[=jobs]
     Specifies the number of jobs (commands) to run simultaneously.  If there is more than one -j
     last  one is effective.  If the -j option is given without an argument, make will not limit
     the number of jobs that can run simultaneously.
```

Проверим как это работает. В качестве эталона для сборки возьмём проект NTP-сервера (выбран проект, который собирается не очень долго, но и не слишком быстро):

```
$ pwd
/usr/src/ntp-4.2.6p3
```

Вот как это происходит на 4-х ядерном процессоре Atom (не очень быстрая модель, частота 1.66Ghz) но с очень быстрым твердотельным SDD:

```
$ cat /proc/cpuinfo | head -n10
processor      : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 28
model name     : Intel(R) Atom(TM) CPU  330   @ 1.60GHz
stepping       : 2
cpu MHz        : 1596.331
cache size     : 512 KB
$ make clean
$ time make -j1
...
real    2m7.698s
user    1m56.279s
sys     0m12.665s
```

```
$ make clean
$ time make -j2
...
real    1m16.018s
user    1m58.883s
sys     0m12.733s
$ make clean
$ time make -j3
...
real    1m9.751s
user    2m23.385s
sys     0m15.229s
$ make clean
$ time make -j4
...
real    1m5.023s
user    2m40.270s
sys     0m16.809s
$ make clean
$ time make
...
real    2m6.534s
user    1m56.119s
sys     0m12.193s
$ make clean
$ time make -j
...
real    1m5.708s
user    2m43.230s
sys     0m16.301s
```

Это работает! А вот та же компиляция на гораздо более быстром 2-х ядерном процессоре, но с типовым HDD:

```
$ cat /proc/cpuinfo | head -n10
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Pentium(R) Dual-Core CPU E6600 @ 3.06GHz
stepping        : 10
cpu MHz         : 3066.000
cache size      : 2048 KB
...
$ pwd
/usr/src/ntp-4.2.6p3
$ time make
...
real    0m31.591s
user    0m21.794s
sys     0m4.303s
$ time make -j2
...
real    0m23.629s
user    0m21.013s
sys     0m3.278s
```

Итоговая скорость здесь в 3-4 раза лучше, но улучшение от числа процессоров только порядка 20%, и это потому, что тормозящим звеном здесь является накопитель, при записи большого числа `.obj` файлов. Но мы можем перенести файлы проекта в `tmpfs` (о чём говорилось при рассмотрении компиляции ядра):

```
$ pwd
/dev/shm/ntp-4.2.6p3
$ make -j
...
```

```
real    0m4.081s
user    0m1.710s
sys     0m1.149s
```

Здесь улучшение относительно исходной компиляции достигает почти порядка!

Резюме этого экскурса: тщательно оптимизируйте условия сборки вашего проекта под оборудование, на котором это производится, и, учитывая, что в процессе отладки сборка выполняется сотни раз — вы сэкономите множество времени!

Приложение Б: Тесты распределителя памяти

Возможности динамического выделения памяти детально обсуждались ранее. Но в литературе и обсуждениях фигурируют самые разнообразные и противоречивые цифры и рекомендации по использованию (или не использованию) механизмов `kmalloc()`, `vmalloc()`, `__get_free_pages()`. Проделаем некоторые грубые оценки на различных компьютерах, с различными объёмами реальной RAM и с установленными Linux различных версий ядра. Для этого используем подготовленные тесты (архив `mtest.tgz`):

memmax.c :

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

static int mode = 0; // выделение памяти: 0 - kmalloc(), 1 - __get_free_pages(), 2 - vmalloc()
module_param( mode, int, S_IRUGO );

char *mfun[] = { "kmalloc", "__get_free_pages", "vmalloc" };

static int __init init( void ) {
   static char *kbuf;
   static unsigned long order, size;
   if( mode < 0 || mode > 2 ) {
      printk( KERN_ERR "illegal mode value\n" );
      return -1;
   }
   for( size = PAGE_SIZE, order = 0; ; order++, size *= 2 ) {
      char msg[ 120 ];
      sprintf( msg, "order=%2ld, pages=%6ld, size=%9ld - %s ",
               order, size / PAGE_SIZE, size, mfun[ mode ] );
      switch( mode ) {
         case 0:
            kbuf = (char *)kmalloc( (size_t)size, GFP_KERNEL );
            break;
         case 1:
            kbuf = (char *)__get_free_pages( GFP_KERNEL, order );
            break;
         case 2:
            kbuf = (char *)vmalloc( size );
            break;
      }
      strcat( msg, kbuf ? "OK\n" : "failed\n" );
      printk( KERN_INFO "%s", msg );
      if( !kbuf ) break;
      switch( mode ) {
         case 0:
            kfree( kbuf );
```

```
            break;
         case 1:
            free_pages( (unsigned long)kbuf, order );
            break;
         case 2:
            vfree( kbuf );
            break;
      }
   }
   return -1;
}
module_init( init );

MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_DESCRIPTION( "memory allocation size test" );
MODULE_LICENSE( "GPL v2" );
```

По 3-м экземплярам компьютеров с Linux указываются ниже перед результатами тестирования: а). версия ядра, б). объём установленной оперативной памяти.

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
$ cat /proc/meminfo | grep MemTotal
MemTotal:        2053828 kB
$ sudo insmod memmax.ko mode=0
insmod: error inserting 'memmax.ko': -1 Operation not permitted
$ dmesg | tail -n100 | grep order
order= 0, pages=     1, size=     4096 - kmalloc OK
order= 1, pages=     2, size=     8192 - kmalloc OK
order= 2, pages=     4, size=    16384 - kmalloc OK
order= 3, pages=     8, size=    32768 - kmalloc OK
order= 4, pages=    16, size=    65536 - kmalloc OK
order= 5, pages=    32, size=   131072 - kmalloc OK
order= 6, pages=    64, size=   262144 - kmalloc OK
order= 7, pages=   128, size=   524288 - kmalloc OK
order= 8, pages=   256, size=  1048576 - kmalloc OK
order= 9, pages=   512, size=  2097152 - kmalloc OK
order=10, pages=  1024, size=  4194304 - kmalloc OK
order=11, pages=  2048, size=  8388608 - kmalloc failed
$ sudo insmod memmax.ko mode=1
insmod: error inserting 'memmax.ko': -1 Operation not permitted
$ dmesg | tail -n100 | grep order
order= 0, pages=     1, size=     4096 - __get_free_pages OK
order= 1, pages=     2, size=     8192 - __get_free_pages OK
order= 2, pages=     4, size=    16384 - __get_free_pages OK
order= 3, pages=     8, size=    32768 - __get_free_pages OK
order= 4, pages=    16, size=    65536 - __get_free_pages OK
order= 5, pages=    32, size=   131072 - __get_free_pages OK
order= 6, pages=    64, size=   262144 - __get_free_pages OK
order= 7, pages=   128, size=   524288 - __get_free_pages OK
order= 8, pages=   256, size=  1048576 - __get_free_pages OK
order= 9, pages=   512, size=  2097152 - __get_free_pages OK
order=10, pages=  1024, size=  4194304 - __get_free_pages OK
order=11, pages=  2048, size=  8388608 - __get_free_pages failed
$ sudo insmod memmax.ko mode=2
insmod: error inserting 'memmax.ko': -1 Operation not permitted
$ dmesg | tail -n100 | grep order
order= 0, pages=     1, size=     4096 - vmalloc OK
order= 1, pages=     2, size=     8192 - vmalloc OK
order= 2, pages=     4, size=    16384 - vmalloc OK
```

```
order= 3, pages=     8, size=    32768 - vmalloc OK
order= 4, pages=    16, size=    65536 - vmalloc OK
order= 5, pages=    32, size=   131072 - vmalloc OK
order= 6, pages=    64, size=   262144 - vmalloc OK
order= 7, pages=   128, size=   524288 - vmalloc OK
order= 8, pages=   256, size=  1048576 - vmalloc OK
order= 9, pages=   512, size=  2097152 - vmalloc OK
order=10, pages=  1024, size=  4194304 - vmalloc OK
order=11, pages=  2048, size=  8388608 - vmalloc OK
order=12, pages=  4096, size= 16777216 - vmalloc OK
order=13, pages=  8192, size= 33554432 - vmalloc OK
order=14, pages= 16384, size= 67108864 - vmalloc failed

$ uname -r
2.6.18-92.el5
$ cat /proc/meminfo | grep MemTotal
MemTotal:       255600 kB
$ sudo /sbin/insmod memmax.ko mode=0
insmod: error inserting 'memmax.ko': -1 Operation not permitted
$ dmesg | tail -n100 | grep order
EXT3-fs: mounted filesystem with ordered data mode.
order= 0, pages=     1, size=     4096 - kmalloc OK
order= 1, pages=     2, size=     8192 - kmalloc OK
order= 2, pages=     4, size=    16384 - kmalloc OK
order= 3, pages=     8, size=    32768 - kmalloc OK
order= 4, pages=    16, size=    65536 - kmalloc OK
order= 5, pages=    32, size=   131072 - kmalloc OK
order= 6, pages=    64, size=   262144 - kmalloc failed
$ sudo /sbin/insmod memmax.ko mode=1
insmod: error inserting 'memmax.ko': -1 Operation not permitted
$ dmesg | tail -n100 | grep order
order= 0, pages=     1, size=     4096 - __get_free_pages OK
order= 1, pages=     2, size=     8192 - __get_free_pages OK
order= 2, pages=     4, size=    16384 - __get_free_pages OK
order= 3, pages=     8, size=    32768 - __get_free_pages OK
order= 4, pages=    16, size=    65536 - __get_free_pages OK
order= 5, pages=    32, size=   131072 - __get_free_pages OK
order= 6, pages=    64, size=   262144 - __get_free_pages OK
order= 7, pages=   128, size=   524288 - __get_free_pages OK
order= 8, pages=   256, size=  1048576 - __get_free_pages OK
order= 9, pages=   512, size=  2097152 - __get_free_pages OK
order=10, pages=  1024, size=  4194304 - __get_free_pages OK
order=11, pages=  2048, size=  8388608 - __get_free_pages failed
$ sudo /sbin/insmod memmax.ko mode=2
insmod: error inserting 'memmax.ko': -1 Operation not permitted
$ dmesg | tail -n100 | grep order
order= 0, pages=     1, size=     4096 - vmalloc OK
order= 1, pages=     2, size=     8192 - vmalloc OK
order= 2, pages=     4, size=    16384 - vmalloc OK
order= 3, pages=     8, size=    32768 - vmalloc OK
order= 4, pages=    16, size=    65536 - vmalloc OK
order= 5, pages=    32, size=   131072 - vmalloc OK
order= 6, pages=    64, size=   262144 - vmalloc OK
order= 7, pages=   128, size=   524288 - vmalloc OK
order= 8, pages=   256, size=  1048576 - vmalloc OK
order= 9, pages=   512, size=  2097152 - vmalloc OK
order=10, pages=  1024, size=  4194304 - vmalloc OK
order=11, pages=  2048, size=  8388608 - vmalloc OK
order=12, pages=  4096, size= 16777216 - vmalloc OK
```

```
order=13, pages=  8192, size= 33554432 - vmalloc OK
order=14, pages= 16384, size= 67108864 - vmalloc OK
order=15, pages= 32768, size=134217728 - vmalloc OK
order=16, pages= 65536, size=268435456 - vmalloc failed

$ uname -r
2.6.35.13-92.fc14.x86_64
$ cat /proc/meminfo | grep MemTotal
MemTotal:        4047192 kB
$ sudo /sbin/insmod memmax.ko mode=0
insmod: error inserting 'memmax.ko': -1 Operation not permitted
$ dmesg | tail -n100 | grep order
[1747955.216447] order= 0, pages=     1, size=     4096 - kmalloc OK
[1747955.216452] order= 1, pages=     2, size=     8192 - kmalloc OK
[1747955.216456] order= 2, pages=     4, size=    16384 - kmalloc OK
[1747955.216460] order= 3, pages=     8, size=    32768 - kmalloc OK
[1747955.216465] order= 4, pages=    16, size=    65536 - kmalloc OK
[1747955.216469] order= 5, pages=    32, size=   131072 - kmalloc OK
[1747955.216475] order= 6, pages=    64, size=   262144 - kmalloc OK
[1747955.216481] order= 7, pages=   128, size=   524288 - kmalloc OK
[1747955.216495] order= 8, pages=   256, size=  1048576 - kmalloc OK
[1747955.216519] order= 9, pages=   512, size=  2097152 - kmalloc OK
[1747955.325561] order=10, pages=  1024, size=  4194304 - kmalloc OK
[1747955.325695] order=11, pages=  2048, size=  8388608 - kmalloc failed
$ sudo /sbin/insmod memmax.ko mode=1
insmod: error inserting 'memmax.ko': -1 Operation not permitted
$ dmesg | tail -n100 | grep order
[1748395.522702] order= 0, pages=     1, size=     4096 - __get_free_pages OK
[1748395.522708] order= 1, pages=     2, size=     8192 - __get_free_pages OK
[1748395.522712] order= 2, pages=     4, size=    16384 - __get_free_pages OK
[1748395.522716] order= 3, pages=     8, size=    32768 - __get_free_pages OK
[1748395.522720] order= 4, pages=    16, size=    65536 - __get_free_pages OK
[1748395.522725] order= 5, pages=    32, size=   131072 - __get_free_pages OK
[1748395.522730] order= 6, pages=    64, size=   262144 - __get_free_pages OK
[1748395.522737] order= 7, pages=   128, size=   524288 - __get_free_pages OK
[1748395.522745] order= 8, pages=   256, size=  1048576 - __get_free_pages OK
[1748395.522759] order= 9, pages=   512, size=  2097152 - __get_free_pages OK
[1748395.522777] order=10, pages=  1024, size=  4194304 - __get_free_pages OK
[1748395.522788] order=11, pages=  2048, size=  8388608 - __get_free_pages failed
$ sudo /sbin/insmod memmax.ko mode=2
insmod: error inserting 'memmax.ko': -1 Operation not permitted
$ dmesg | tail -n100 | grep order
[1747830.678358] order= 0, pages=     1, size=     4096 - vmalloc OK
[1747830.678445] order= 1, pages=     2, size=     8192 - vmalloc OK
[1747830.678496] order= 2, pages=     4, size=    16384 - vmalloc OK
[1747830.678552] order= 3, pages=     8, size=    32768 - vmalloc OK
[1747830.678607] order= 4, pages=    16, size=    65536 - vmalloc OK
[1747830.678667] order= 5, pages=    32, size=   131072 - vmalloc OK
[1747830.678745] order= 6, pages=    64, size=   262144 - vmalloc OK
[1747830.678848] order= 7, pages=   128, size=   524288 - vmalloc OK
[1747830.679015] order= 8, pages=   256, size=  1048576 - vmalloc OK
[1747830.679312] order= 9, pages=   512, size=  2097152 - vmalloc OK
[1747830.679932] order=10, pages=  1024, size=  4194304 - vmalloc OK
[1747830.681139] order=11, pages=  2048, size=  8388608 - vmalloc OK
[1747830.683463] order=12, pages=  4096, size= 16777216 - vmalloc OK
[1747830.688677] order=13, pages=  8192, size= 33554432 - vmalloc OK
[1747830.697957] order=14, pages= 16384, size= 67108864 - vmalloc OK
[1747830.712238] order=15, pages= 32768, size=134217728 - vmalloc OK
[1747830.742639] order=16, pages= 65536, size=268435456 - vmalloc OK
```

```
[1747830.810859] order=17, pages=131072, size=536870912 - vmalloc OK
[1747831.040146] order=18, pages=262144, size=1073741824 - vmalloc OK
[1747831.636957] order=19, pages=524288, size=2147483648 - vmalloc OK
[1747831.784385] order=20, pages=1048576, size=4294967296 - vmalloc failed
```

Обратите внимание!: тест показывает не максимально возможный размер блока, который тот или иной механизм выделения памяти способен разместить (и такой тест несложно соорудить из показанного), а грубо оценивает блок, который уже нельзя разместить.

Следующая вещь, которая явно требует оценивания — это порядок временных затрат на выделение блока при использовании того или иного механизма. Код такого модуля-теста показан ниже:

memtim.c :

```
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/vmalloc.h>
#include <asm/msr.h>
#include <linux/sched.h>

static long size = 1000;
module_param( size, long, 0 );

#define CYCLES 1024                       // число циклов накопления

static int __init init( void ) {
   int i;
   unsigned long order = 1, psize;
   unsigned long long calibr = 0;
   const char *mfun[] = { "kmalloc", "__get_free_pages", "vmalloc" };
   for( psize = PAGE_SIZE; psize < size; order++, psize *= 2 );
   printk( KERN_INFO "size = %ld order = %ld(%ld)\n", size, order, psize );
   for( i = 0; i < CYCLES; i++ ) {       // калибровка времени выполнения rdtscll()
      unsigned long long t1, t2;
      schedule();                        // обеспечивает лучшую повторяемость
      rdtscll( t1 );
      rdtscll( t2 );
      calibr += ( t2 - t1 );
   }
   calibr = calibr / CYCLES;
   printk( KERN_INFO "calibr=%lld\n", calibr  );
   for( i = 0; i < sizeof( mfun ) / sizeof( mfun[ 0 ] ); i++ ) {
      char *kbuf;
      char msg[ 120 ];
      int j;
      unsigned long long suma = 0;
      sprintf( msg, "proc. cycles for allocate %s : ", mfun[ i ] );
      for( j = 0; j < CYCLES; j++ ) {    // циклы накопления измерений
         unsigned long long t1, t2;
         schedule();                     // обеспечивает лучшую повторяемость
         rdtscll( t1 );
         switch( i ) {
            case 0:
               kbuf = (char *)kmalloc( (size_t)size, GFP_KERNEL );
               break;
            case 1:
               kbuf = (char *)__get_free_pages( GFP_KERNEL, order );
               break;
            case 2:
               kbuf = (char *)vmalloc( size );
```

```
                break;
            }
            if( !kbuf ) break;
            rdtscll( t2 );
            suma += ( t2 - t1 - calibr );
            switch( i ) {
               case 0:
                  kfree( kbuf );
                  break;
               case 1:
                  free_pages( (unsigned long)kbuf, order );
                  break;
               case 2:
                  vfree( kbuf );
                  break;
            }
         }
         if( kbuf )
            sprintf( ( msg + strlen( msg ) ), "%lld", ( suma / CYCLES ) );
         else
            strcat( msg, "failed" );
         printk( KERN_INFO "%s\n", msg );
      }
      return -1;
   }
   module_init( init );
   MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
   MODULE_DESCRIPTION( "memory allocation speed test" );
   MODULE_LICENSE( "GPL v2" );
```

Результаты этого теста я приведу только для одной системы, из-за их объёмности и громоздкости. Вы их можете повторить для своего компьютера и своей версии ядра:

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
$ sudo insmod ./memtim.ko
insmod: error inserting './memtim.ko': -1 Operation not permitted
$ dmesg | tail -n4
size = 1000 order = 1(4096)
proc. cycles for allocate kmalloc : 146
proc. cycles for allocate __get_free_pages : 438
proc. cycles for allocate vmalloc : 210210

$ sudo insmod ./memtim.ko size=4096
insmod: error inserting './memtim.ko': -1 Operation not permitted
$ dmesg | tail -n4
size = 4096 order = 1(4096)
proc. cycles for allocate kmalloc : 181
proc. cycles for allocate __get_free_pages : 877
proc. cycles for allocate vmalloc : 59626

$ sudo insmod ./memtim.ko size=65536
insmod: error inserting './memtim.ko': -1 Operation not permitted
$ dmesg | tail -n4
size = 65536 order = 5(65536)
proc. cycles for allocate kmalloc : 1157
proc. cycles for allocate __get_free_pages : 940
proc. cycles for allocate vmalloc : 84129

$ sudo insmod ./memtim.ko size=262144
```

```
insmod: error inserting './memtim.ko': -1 Operation not permitted
$ dmesg | tail -n4
size = 262144 order = 7(262144)
proc. cycles for allocate kmalloc : 2151
proc. cycles for allocate __get_free_pages : 2382
proc. cycles for allocate vmalloc : 52026
```

В последнем нашем эксперименте сделаем блок не кратным размеру страницы MMU (чуть-чуть урежем значение из предыдущего запуска):

```
$ sudo insmod ./memtim.ko size=262000
insmod: error inserting './memtim.ko': -1 Operation not permitted
$ dmesg | tail -n4
size = 262000 order = 7(262144)
proc. cycles for allocate kmalloc : 8674
proc. cycles for allocate __get_free_pages : 4730
proc. cycles for allocate vmalloc : 55612
```

- видно, как `__get_free_pages()` и `kmalloc()` (что странно для последнего) «впадают в задумчивость», и в разы теряют производительность; практически не замечает этого изменения.

Можно заметить следующее:

- При распределении малых блоков разница `kmalloc()` и `vmalloc()` разительная, и составляет до 3-х порядков:

```
$ sudo insmod ./memtim.ko size=5
insmod: error inserting './memtim.ko': -1 Operation not permitted
$ dmesg | tail -n30 | grep -v audit
size = 5 order = 1(4096)
proc. cycles for allocate kmalloc : 143
proc. cycles for allocate __get_free_pages : 890
proc. cycles for allocate vmalloc : 152552
```

- При увеличении размеров запрашиваемого блока различия нивелируются, и на больших объёмах не превышают порядка.

- В этих различиях нет ничего страшного, учитывая ту гибкость и диапазон, которые обеспечивает как раз `vmalloc()`, если только речь не идёт о быстром получении-удалении малых блоков в динамике.

Источники информации

[1]. Peter Jay Salzman, Michael Burian, Ori Pomerantz: «The Linux Kernel Module Programming Guide», 2001.

Перевод Андрей Киселёв: «Руководство по программированию модулей ядра Linux», 2004:

http://citforum.univ.kiev.ua/operating_systems/linux/lkmpg/

[2]. Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman: «Linux Device Drivers», (3rd Edition), 2005, 2001, 1998, O'Reilly Media, Inc., ISBN: 0-596-00590-3.

Перевод: «Драйверы Устройств Linux, Третья Редакция»...

- для онлайн чтения: http://dmilvdv.narod.ru/Translate/LDD3/index.html?linux_device_drivers.html
- для скачивания в PDF формате: http://dmilvdv.narod.ru/Translate/LDD3/Linux_Device_Drivers_3_ru.pdf

[3]. Robert Love: «Linux Kernel Development», (3rd Edition), 2010.

Русское 2-е издание: Р. Лав: «Разработка ядра Linux», М.: «Изд. Дом Вильямс», 2006, стр. 448.

[4]. Wolfgang Mauerer : «Professional Linux Kernel Architecture (Wrox Programmer to Programmer)», Wiley Publishing Inc., 2008, p.1335.

[5]. Sreekrishnan Venkateswaran, «Essential Linux Device Drivers», Prentice Hall, 2008, p.714.

Сайт книги: http://elinuxdd.com

Архив кодов примеров: http://elinuxdd.com/~elinuxdd/elinuxdd.docs/listings/

[6]. Jerry Cooperstein, «Writing Linux Device Drivers», 2009,

том 1: «A guide with exercises», стр. 372

том 2: «Lab Solutions», стр. 259

Авторский сайт: http://coopj.com/

Архив кодов примеров: http://coopj.com/LDD/

[7]. Клаудия Зальзберг Родригес, Гордон Фишер, Стивен Смолски: «Linux. Азбука ядра», Пер. с англ., М.: «Кудиц-образ», 2007, стр. 577.

[8]. А. Гриффитс, «GCC. Полное руководство. Platinum Edition», Пер. с англ., М.: «ДиаСофт», 2004, ISBN 966-7992-33-0, стр. 624.

[9]. Олег Цилюрик, Егор Горошко: «QNX/UNIX: анатомия параллелизма», СПб.: «Символ-Плюс», 2005, ISBN 5-93286-088-X, стр. 288. Книга по многим URL в Интернет представлена для скачивания, например, здесь: http://bookfi.org/?q=Цилюрик&ft=on#s

[10]. Бовет Д., Чезати М.: «Ядро Linux, 3-е издание», Пер. с англ., СПб.: «БХВ-Петербург», 2007, ISBN 978-5-94157-957-0, стр. 1104. Книга может быть скачана:

http://proxy.bookfi.org/genesis/49000/7e38ee9e1d14e03708699ea5ea2b4f88/_as/%5BBovet_D.,_CHezati_M.%5D_YAdro_Linux(BookFi.org).djvu

[11]. Крищенко В. А., Рязанова Н. Ю.: «Основы программирования в ядре операционной системы GNU/Linux», сдано в издательство МГТУ в 2008 году.

Текст статьи: http://sevik.ru/syslinux/pdf/sys_linux.pdf

Примеры кода к статье: http://sevik.ru/syslinux/samples/syslinux_samples.tar.gz

[12]. Greg Kroab-Hartman: «Linux Kernel in a Nutshell», O'Reilly Vtdia, Inc., 2007, ISBN-10: 0-596-10079-5, стр. 184.

http://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/index.html

[13]. «The Linux Kernel API» :

http://www.kernel.org/doc/htmldocs/kernel-api/

[14]. Роб Кёртен: «Введение в QNX Neutrino. Руководство для разработчиков приложений реального времени», Пер. с англ., СПб.: BHV-СПб, 2011, ISBN 978-5-9775-0681-6, 368 стр.

[15]. Клаус Вейрле, Фронк Пэльке, Хартмут Риттер, Даниэль Мюллер, Марк Бехлер: «Linux: сетевая архитектура. Структура и реализация сетевых протоколов в ядре», Пер. с англ., М.: «КУДИЦ-ОБРАЗ», 2006, ISBN 5-9579-0094-X, стр. 656.

[16]. David Mosberger, Stephane Eranian: «IA-64 Linux Kernel», Hewlet-Packard Company, Prentice Hall PTR, 2002, стр. 522

[17]. Rajaram Regupathy: «Bootstrap Yourself with Linux-USB Stack: Design, Develop, Debug, and Validate Embedded USB», Course Technology, a part of Cengage Lernining, 2012, ISBN-10: 1-4354-5786-2, стр. 302.

[18]. У. Р. Стивенс, «UNIX: взаимодействие процессов», СПб.: «Питер», 2003, ISBN 5-318-00534-9, стр. 576.

[19]. У. Ричард Стивенс, Стивен А. Раго: «UNIX. Профессиональное программирование», второе издание, СПб.: «Символ-Плюс», 2007, ISBN 5-93286-089-8, стр. 1040. Полный архив примеров кодов к этой книге может быть взят здесь: http://www.kohala.com/start/apue.linux.tar.Z

[20]. W. Richard Stevens' Home Page (ресурс полного собрания книг и публикаций У. Р. Стивенса):

http://www.kohala.com/start/

[21]. Tigran Aivazian (tigran@veritas.com): «Внутреннее устройство Ядра Linux 2.4», 21 October 2001, Перевод: Андрей Киселев.

http://doc.agro.net.ua/lib.profi.net.ua/opennet/docs/RUS/lki/lki.html#toc2

[22]. «GNU Make. Программа управления компиляцией. GNU `make` Версия 3.79. Апрель 2000», авторы: Richard M. Stallman и Roland McGrath, перевод: Владимир Игнатов, 2000.

http://linux.yaroslavl.ru/docs/prog/gnu_make_3-79_russian_manual.html

[23] «Отладчик GNU уровня исходного кода. Восьмая Редакция, для GDB версии 5.0. Март 2000», авторы: Ричард Столмен, Роланд Пеш, Стан Шебс и др.».

http://linux.yaroslavl.ru/docs/altlinux/doc-gnu/gdb/gdb.html

[24]. Cristian Benvenuti: «Understanding Linux Network Internals», O'Reilly Media, Inc., 2006, ISBN: 978-0-596-00255-8, стр.1035.

[25]. А. Соловьев: «Разработка модулей ядра ОС Linux (Kernel newbie's manual)»:

http://rus-linux.net/MyLDP/BOOKS/knm.pdf

[26]. Зубков С.В.: «Assembler для DOS, Windows, UNIX», М.:"ДМК Пресс", 2000, ISBN 5-94074-003-0, стр.608.

[27]. Dmitri Gribenko: «Ассемблер в Linux для программистов С», 06.06.2008 :

http://wasm.ru/article.php?article=asm_linux_for_c

[28]. «Building a custom kernel», http://fedoraproject.org/wiki/Docs/CustomKernel

Есть перевод этой публикации: http://forum.russianfedora.ru/viewtopic.php?f=14&t=1367&start=0)

[29]. М. Тим Джонс, «Анатомия распределителя памяти slab в Linux» :

http://www.ibm.com/developerworks/ru/library/l-linux-slab-allocator/index.html?S_TACT=105AGX99&S_CMP=GR01

[30]. «GCC-Inline-Assembly-HOWTO», автор перевода мне неизвестен :

http://www.iakovlev.org/index.html?p=1483&m=1

[31]. Павел Курочкин, «Разработка драйверов для USB-устройств под Linux», 19 Nov 2006:

http://www.opennet.ru/base/dev/write_linux_driver.txt.html

[32]. «Реализация низкоуровневой поддержки шины PCI в ядре Linux», 5 Aug 2004:

http://www.opennet.ru/base/dev/pci_linux_kernel.txt.html

[33]. Олег Цилюрик: «Модули ядра Linux» - серия из 77 статей, опубликованных на сайте IBM Developer Works: 1-я статья цикла - https://www.ibm.com/developerworks/ru/library/l-linux_kernel_01/, последняя статья цикла - https://www.ibm.com/developerworks/ru/library/l-linux_kernel_77/

[34]. Lennart Poettering : «systemd для администраторов», перевод Сергей Пташник — периодически обновляемое руководство по systemd, последняя редакция 3 марта 2014 г., 98 стр. - http://www2.kangran.su/~nnz/pub/s4a/s4a_latest.pdf

[35]. Rami Rosen : «Linux Kernel Networking: Implementation and Theory», Apress, 650 pages, 2014, ISBN-13: 978-1-4302-6196-4

http://www.amazon

.com/Linux-Kernel-Networking-Implementation-Theory-ebook/dp/B00FL16XUO/ref=sr_1_3?s=books&ie=UTF8&qid=1392043891&sr=1-3&keywords=Linux+kernel

Ссылка для скачивания: http://www.foxebook.net/linux-kernel-networking-implementation-and-theory/

[36]. Lixiang Yang : «The Art of Linux Kernel Design: Illustrating the Operating System Design Principle and Implementation»,

http://www.amazon.com/Art-Linux-Kernel-Design-Implementation/dp/1466518030/ref=sr_1_9?s=books&ie=UTF8&qid=1392043891&sr=1-9&keywords=Linux+kernel

[37]. Скотт Максвелл : «Ядро Linux в комментариях», К: «ДиаСофт», 2000. - 488 стр., ISBN 966-7393-46-1

[38]. Олег Цилюрик:

«Сопоставление: 10 языков программирования», http://mylinuxprog.blogspot.com/2014/02/10.html

«Ещё несколько экзотических ЯП», http://mylinuxprog.blogspot.com/2014/03/blog-post.html

[39]. Harald Kipp : «ARM GCC Inline Assembler Cookbook»

http://www.ethernut.de/en/documents/arm-inline-asm.html

[40]. GCC online documentation, these are manuals for the latest full releases

http://gcc.gnu.org/onlinedocs/

[41]. libusb проект http://www.libusb.org/wiki/WikiStart

[42]. libusb Developers Guide http://libusb.sourceforge.net/doc/

[43]. libusb-1.0 API Reference http://libusb.sourceforge.net/api-1.0/

[44]. The Linux Kernel API http://oss.org.cn/ossdocs/gnu_linux/kernel-api/

[45]. LXR (Linux Cross-Referencer) ресурсы перекрёстного анализа исходных кодов ядра Linux:

http://lxr.free-electrons.com/source/

http://lxr.linux.no/

http://lxr.missinglinkelectronics.com/linux

http://lxr.oss.org.cn/

[46]. Страница проекта FUSE - Filesystem in Userspace:

http://fuse.sourceforge.net/