Программирование модулей ядра Linux

Проект книги

Олег Цилюрик,

редакция 3.159

23.04.2012г.

Оглавление

Содержание

Памяти моих родителей, которые научили меня, что одним из немногих достойных занятий, которыми занимаются люди, является написание книг.

Предисловие от автора

«Omne tulit punctum qui miscuit utile dulci, lectorem delectando pariterque monenendo» :

«Всеобщего одобрения заслуживает тот, кто соединил приятное с полезным».

Гораций, «Ars Poetica».

Введение

Эта книга появилась как итог подготовки и проведения курса тренингов, которые мне предложила организовать компания Global Logic (http://www.globallogic.com/) для сотрудников украинских отделений (http://globallogic.com.ua) компании (на Украине работают свыше 2000 сотрудников этой компании). Первоначальный курс, начитанный в тренинговом[1] цикле весны-лета 2011 года в Харькове и составил базовую часть текста. Второй «подход» к курсу состоялся в тренинговом курсе лета-осени 2011 года, проводимом для Львовского отделения компании, специализирующемся более на разработках для встраиваемого оборудования — это потребовало существенной детализации материала. К завершению подготовки курса стало ясно, что большую проделанную работу, главным образом по написанию и отладке **примеров кода**, жаль терять бессмысленно, только как иллюстративный материал к тренингам. Более того, несколько моих коллег прошлых лет, на протяжении этой работы, обращались с просьбой переслать им материал в том «сыром» виде как он находился в процессе (as is), и утверждали позже, что он им заметно помог. Всё это подвигло на намерение довести лекционный материал до печатного издания. Исходные тексты были значительно дополнены и переработаны, итогом чего и получилась эта книга, которую вы держите в руках.

Литература по программированию модулей ядра Linux хоть и малочисленна, но она есть. В конце книги приведено достаточно много обстоятельных источников информации по этому предмету: они достаточно хороши, а отдельные из них — так просто замечательные... Но актуальность (по моему мнению) дополнительной систематизации информации, попытка которой сделана в этой книге, на момент её написания подталкивается ещё и двумя дополнительными обстоятельствами:

- Всплеск интереса к операционным системам, базирующихся на ядре Linux, для самых различных классов мобильных устройств. Примерами того есть в высшей степени динамично развивающаяся система Android, или анонсированная к ближайшему завершению система Chrome OS. И в этих тенденциях прослеживается такая особенность, что инструментарий развития прикладных приложений (Java слой) предоставляется и афишируется в максимальной мере, в то время, как средства «натягивания» ядра операционной системы на специфическое оборудование заметно (или сознательно?) вуалируются (лицензия GPL обязывает, но разработчики не особенно торопятся...).

- Тенденция роста числа процессоров в единице оборудования: на сегодня уже не являются экзотикой компьютеры SMP с 2-4 ядрами, или в комбинации: 4 процессора по 4 ядра (пусть это пока и в производительных серверах). Плюс каждое ядро может быть дополнено гипертрейдингом. Но и это только начало: большую активность приобрёл поиск технологий параллельного использования десятков, сотен, а то и тысяч параллельно работающих процессоров — в эту сторону обратим внимание на модель программирования CUDA от компании NVIDIA. Все эти архитектуры используются эффективно только в том случае, если SMP адекватно поддерживается со стороны ядра.

- Очень серьёзное расширение (в последние 5-7 лет) аппаратных платформ, **на практике** используемых IT рынком. Если ещё 10 лет многочисленные процессорные платформы: SPARK, ARM, MIPS, PPC, ...

1 Так процесс обучения принято называть в компании, поэтому я придерживаюсь принятой терминологии ... на то, что мы проводили. Хотя я, скорее называл бы это: практический семинар, когда специалисты, все с изрядным опытом завершённых проектов, обмениваются мнениями и складывают единую точку зрения на вещи.

— существовали как маргинальные линии развития в стороне от доминирующей линии Intel x86, то на сегодня картина разительно изменилась. И все эти архитектуры поддерживаются системой Linux (здесь вам не альянс Intel & Microsoft). И более того, на многих аппаратных платформах, «покувыркавшись» со своими оригинальными проприетарными системами программного обеспечения, начинают рассматривать Linux как **основную** программную платформу.

Эти наметившиеся тенденции, если и не подвигают к немедленному написанию собственных компонент ядра (что, возможно, и совершенно не обязательно), то, по крайней мере, подталкивают интерес к более точному пониманию и анализу тех процессов, которые происходят в ядре.

Материалы данной книги (сам текст, сопутствующие его примеры, файлы содержащие эти примеры), как и предмет её рассмотрения — задумывались и являются свободно распространяемыми, и могут передаваться и/или изменяться в соответствии с условиями GNU (General Public License), опубликованными Free Software Foundation, версии 2 или более поздней.

Кому адресована эта книга

Книга рассчитана на опытных разработчиков системного программного обеспечения. Предполагается, возможно, отсутствие у читателя богатого опыта в программировании именно для ядра Linux, или даже вообще в программировании для этой системы - но предполагается какой-то опыт в системном программировании для других операционных систем, который будет базой для построения аналогий. В высшей степени плодотворно любое знакомство с одной или несколькими POSIX системами: Open Solaris, QNX, FreeBSD, NetBSD, MINIX3... - с любой из них в равной степени.

Совершенно естественно, что от читателя требуется квалифицированное знание языка C — единственного необходимого и достаточного языка системного программирования (из числа компилирующих) в Linux. Это необходимо для самостоятельного анализа и понимания приводимых примеров — примеры приводятся, но код их детально не обсуждается. Очень продуктивно в дополнение к этому (для работы с многочисленными приводимыми примерами, а ещё больше - их последующей модификации и сравнений) хотя бы минимальные познания в языках скриптового программирования UNIX (и лучше нескольких), что-то из числа: `bash`, `perl`, `awk`, `python`... В высшей степени безусловным подспорьем будет знание и опыт прикладного программирования в стандартах POSIX: обладающий таким опытом найдёт в нём прямые аналогии API и механизмам в ядре Linux.

Естественно, я предполагаю, что вы «на дружеской ноге» с UNIX/POSIX консольными утилитами, такими, как: `ls`, `rm`, `grep`, `tar` и другие. 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, я думаю, не столь значительны, чтобы вызвать какие-либо затруднения.

Структура книги

Исходя из целевого предназначения, построена и структура книги. Начинаем мы, без всяких предварительных объяснений, с интуитивно понятного всякому программисту примера написания простейшего модуля ядра. Только после этого мы возвращаемся к детальному рассмотрению того, чем же является модуль, и какое место он занимает в общей архитектуре Linux, и как он соотносится с ядром системы и приложениями пространства пользователя. Далее будет **очень беглый** обзор того инструмента, который мы «имеем в руках» для ежедневного использования при разработке модулей ядра.

Всё, что описывается далее — это разные стороны техники написания модулей ядра: управление памятью, взаимопонимание со службой времени, обработка прерываний, программные интерфейсы символьных и сетевых устройств ... и многое другое. Но укрупнённо эти вопросы отсортированы примерно по

следующему порядку, несколько отличающемуся от часто принятого:

- Беглый обзор простейших возможностей, как уже было сказано...
- Вопросы архитектуры системы;
- Краткий обзор доступных инструментов прграммирования;
- Обзор API и механизмов, представляющих интерфейсы модуля во внешний мир, в пространство пользователя;
- Обзор API и механизмов, но реализующих «внутреннюю механику ядра» - это инструментарий программиста для внутренней реализации алгоритмики модуля;
- Отдельные тонкие вопросы и возможности, не так часто требующиеся при отработке драйверов, но представляющих интерес.

Текст этой книги (как и предшествовавший ему курс тренингов) ориентировался, в первую очередь, не столько для чтения или разъяснений, сколько в качестве справочника при последующей работе по программированию в этой области. Это накладывает отпечаток на текст (и обязательства на автора):

- 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.
- Подготовлено максимальное (в меру моих сил возможное) число **примеров**, иллюстрирующих изложение. Это не примеры гипотетические, описывающие фрагментарно как должно быть, все 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 этот пример и что он должен показать).
- С другой стороны, эти примеры максимально упрощены, исключено всё лишнее, что могло бы усложнять их ясность. Хотелось бы предполагать, что такие примеры могут быть не только иллюстрацией, но **стартовыми шаблонами**, от которых можно оттолкнуться при написании своего кода того же целевого предназначения: берём такой шаблон, и начинаем редактированием наращивать его спецификой развиваемого проекта...
- Именно с этой целью (возможность последующего справочного использования) в тексте показаны и протоколы выполняемых команд: возможно, с излишней детализацией опций командной строки и показанного результата выполнения — для того, чтобы такие команды можно было **воспроизводить** и повторять, не запоминая их детали.

Некоторые стороны программирования модулей я сознательно опускаю или минимизирую. Это те вопросы, которые достаточно редко могут возникнуть у разработчиков программных проектов общего назначения: примером такой оставленной за пределами рассмотрения техники является программирование драйверов блочных устройств — эта техника вряд ли понадобится кому либо, кроме самой команды разработчиков новой модели устройства прямого доступа, а они, нужно надеяться, являются уже профессионалами такого класса, что успешно сконструируют драйвер для своего устройства. Максимально сокращены и те разделы, по которым трудно (или мне не удалось) показать действующие характерные программные **примеры** минимально разумного объёма. Рассказа на качественном уровне, «на пальцах» - я старался всячески избегать: те аспекты, которые я не могу показать в коде, я просто грубо опускаю (возможно, в более поздних редакциях они найдут своё место).

Во многих местах по тексту разбросаны абзацы, предваряемые выделенным словом: «**Примечание:**». Иногда это одна фраза, иногда пол-страницы и более... Это: необязательные уточняющие детали, обсуждение (или точнее указание) **непонятных мне** на сегодня деталей, где-то это «лирическое отступление» в сторону от основной линии изложения, но навеянное попутно... Если кого-то такие отступления станут утомлять — их

можно безболезненно опускать при чтении.

В завершение всего текста, оформленные несколькими отдельными приложениями, приводятся:

- протокольное описание процесса **сборки ядра** Linux из исходных кодов ядра по шагам: вещь, которая должна быть хорошо известна любому системному программисту, но ... в качестве полезного напоминания;
- краткая справка по утилите `make`, с которой приходится работать постоянно - вещи хорошо известные, но в качестве памятки не помешает;
- пример интерфейса (проект DAHDI) к физическим линиям связи VoIP телефонных коммутаторов (PBX, SoftSwitch) — как **образец** наиболее обширного и развитого (из известных автору) приложений техники модулей ядра в Linux последних лет развития;
- исходный код тестов (примеры кода) динамического выделения памяти, обсуждение и результаты этих тестов - вынесены отдельным приложением из-за их объёмности и детальности.

В любом случае, текст этой книги произошёл от первоначального конспекта тренингового курса, проведенного с совершенно прагматическими намерениями для контингента профессиональных разработчиков программного обеспечения. Таким конспектом он и остаётся: никаких излишних подробных разъяснений, обсуждений... Хотелось бы надеяться, что он, совместно с прилагаемыми примерами кода, может быть отправной точкой, шаблоном, оттолкнувшись от которого можно продолжать развитие реального проекта, затрагивающего область ядра Linux.

Соглашения принятые в тексте

Для ясности чтения текста, он размечен шрифтами по функциональному назначению. 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**.
- Тексты программных листингов, вывод в ответ на консольные команды пользователя размечен `моноширинным шрифтом`.
- Таким же `моноширинным шрифтом` (прямо в тексте) будут выделяться: имена команд, программ, файлов ... т.е. всех терминов, которые должны оставаться неизменяемыми, например: `/proc`, `mkdir`, `./myprog`, ...
- Ввод пользователя в консольных командах (сами команды, или ответы в диалоге), кроме того, выделены **`жирным моноширинным шрифтом`**, чтобы отличать от ответного вывода системы.
- Имена файлов программных листингов записаны 1-й строкой, предшествующей листингу, и выделены ***`жирным подчёркнутым курсивом`***.

Примечание: В некоторых примерах команды работы с модулем будут записываться так:

```
# insmod ./module
```

В других вот так (что будет означать то же самое: **#** - будет означать команду с правами `root`, `$` - команду в правами ординарного пользователя):

```
$ sudo insmod ./module
```

В некоторых — это будет так (что, опять же, то же самое):

```
$ sudo /sbin/insmod ./module
```

Последний случай обусловлен тем, что примеры проверялись на нескольких различных инсталляциях и дистрибутивах

Linux (для большей адекватности, и в силу продолжительности работы ... что было под рукой, на том и проверялось), в некоторых инсталляциях каталог `/sbin` не входит в переменную `$PATH` для ординарного пользователя, а править скопированный с терминала протокол выполнения команд произвольно (как «должно быть», а не «как есть») я не хотел во избежание внесения несоответствий.

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. В тексте, где обсуждаются коды примеров, везде, по возможности, будет указано в скобках имя архива в 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 представлено 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 может «сползти» при правках), но указаны архивы, где их можно найти в полном виде. В архивах примеров могут содержаться файлы вида `*.hist` (расширение — hist, 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.

Некоторые из обсуждаемых примеров заимствованы из публикаций (перечисленных в конце под рубрикой «Источники информации») - в таких случаях я старался везде указать источник заимствования. Другие из примеров возникли в обсуждениях с коллегами по работе, часть примеров реализована ими, или совместно с ними в качестве идеи теста... Во всех случаях я старался сохранить отображение первоначального авторства в кодах (в комментариях, в авторских макросах).

И ещё одно немаловажное замечание относительно программных кодов. Я отлаживал и проверял эти примеры не менее чем на полутора десятков различных инсталляций Linux (реальных и виртуальных, 32 и 64 разрядные архитектуры, установленных из различных дистрибутивов, и разных семейств дистрибутивов: Fedora, CentOS, 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>
```

- которая для других инсталляций будет избыточной (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 версии ядра

Этот текст, в теперешнем его виде, подготавливался, писался, изменялся и дополнялся на протяжении продолжительного времени. К нему отбирались, писались, отрабатывались и совершенствовались примеры реализации кода. За всё это время «в обиходе» сменились версии ядра от 2.6.32 до 3.3. Примеры и команды, показываемые в тексте, отрабатывались и проверялись на всём этом диапазоне, и, в дополнение, на нескольких различных дистрибутивах Linux: Fedora, CentOS, Ubuntu. Помимо этого, в качестве основы для некоторых примеров брался код из практических реализаций прошлых лет в системе:

CentOS 5.2 :

```
$ 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 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 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, а они продолжают фигурировать во множестве описаний. Ниже приводятся для примера короткий фрагмент хронологии выходов нескольких последовательных версий ядра (в последней колонке указано число дней от предыдущей версии до текущей), взято из http://www.docstoc.com/docs/23932299/Linux-Kernel-Development :

```
...
2.6.30    2009-06-09    78
2.6.31    2009-09-09    92
2.6.32    2009-12-02    84
2.6.33    2010-02-24    84
2.6.34    2010-05-15    81
2.6.35    2010-08-01    77
...
```

Среднее время до выхода очередного ядра на протяжении нескольких последних лет (2005-2010) составляло 81 день, или около 12 недель (взято там же). Нет оснований полагать, что этот темп сменится в дальнейшем.

Уникальным ресурсом, позволяющем изучить и сравнить исходный код ядра для различных версий ядра и аппаратных платформ, является: http://lxr.free-electrons.com/. Это основной источник (из известных автору), позволяющий сравнивать изменения в API и реализациях от версии к версии (начиная от 2.6.28 и далее).

Источники информации

Самая неоценимая помощь компании Global Logic, которую только можно было оказать, состояла в том, что на протяжении всей работы компания заказывала на Amazon (http://www.amazon.com/) подлинники всех книг, изданных за последние несколько лет в мире, которые я мог найти полезными для этой работы. Как оказалось, таких изданий в мире не так и много, не более двух-трёх десятков (это если считать со смежными с программированием модулей вопросами). Некоторые, 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. По некоторым источникам 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 по Интернет, удаётся установить авторство.

Беглый взгляд на программирование модуля

Все мы умеем, и имеем больший или меньший опыт написания программ в Linux[2], 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[3]. 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 [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 супервизорных 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[4].

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

2 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.

3 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.

4 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 то же минимальное требование, которое обязательно при работе с модулями ядра.

Наш первый модуль ядра

««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 начального 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 и наблюдать исполнение. И только потом, ознакомившись с некоторыми основополагающими принципами и приёмами работы из мира модулей, перейти к их систематическому изучению.

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

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

Makefile :

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

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

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

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

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

Делаем сборку модуля:

```
$ 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 X11 вывод не попадает в терминал, и его можно видеть только в лог файле `/var/log/messages`. Но и в текстовую консоль вывод направляется не непосредственно, а через демон системного журнала, и выводится на экран только если демон конфигурирован для вывода таких сообщений, вопросы использования и конфигурирования демонов журнала будут детально рассмотрены позже.

```
$ 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
$ 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!
```

Выше 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. 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 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.

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.

Внутренняя структура 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 оъектных 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 важдной 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):

```
$ 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 второй 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 **:**

```
#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` не попадает, 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 показанном варианте окажется неудачной: в системе 2.6.32 (`rsyslog`) будет выведено 3 строки, а в 2.6.18 (`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. Большинство нормальных разработчиков, по крайней мере при определённых обстоятельствах или в определённые периоды, ведут отработку модулей в X11, и иногда крайне удобно иметь возможность параллельно контролировать вывод на текстовой консоли (а иногда это и единственный способ видеть диагностику, когда она не успевает дойти до системного журнала перед гибелью системы). Всю оставшуюся часть этого раздела мы будем обсуждать, как удобнее это сделать, и как управлять всеми этими консолями, поэтому, если эти возможности вас не интересуют, эту часть можно спокойно опустить.

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». Но вот распределение экранов по i может быть разным (в зависимости от способа конфигурации дистрибутива Linux), я встречал:

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

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

Большой неожиданностью может стать отсутствие вывода `printk()` в текстовую консоль. Но этот 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. В более старых дистрибутивах в качестве демонов логирования используются `syslogd` и `klogd`, проверить это можете:

```
$ ps -A | grep logd
 4094 ?        00:00:00 syslogd
 4097 ?        00:00:00 klogd
```

Нас, в данном случае, интересует `klogd`. В более свежих дистрибутивах может использоваться один демон `rsyslogd`, берущий на себя функции и `syslogd` и `klogd`:

```
$ ps -A | grep logd
 1227 ?        00:00:00 rsyslogd
```

С какими параметрами предстоит перезапускать демон журнала зависит, естественно, от вида демона... Детальную информацию вы можете получить командами (и, соответственно, точно так же и для варианта `klogd`):

```
$ man rsyslogd
...
$ rsyslogd --help
...
```

Для более старого `klogd` нужная нам возможность (изменить порог вывода) - это ключ `-c`. Для модульного `rsyslogd`, идущего на смену `sysogd` и `klogd` - это определяется в его конфигурационном файле `/etc/rsyslog.conf`, где находим такие строки:

```
$ cat /etc/rsyslog.conf
....
#### RULES ####
# Log all kernel messages to the console.
# Logging much else clutters up the screen.
#kern.*                                                 /dev/console
...
```

Раскомментируем эту строку, немного изменив её вид:

```
kern.*                                                  /dev/tty12
```

- после этого вывод модулей будет направляться на консоль 12 (любую не инициированную, т.е. стандартно: с номером больше 6), на которую переключаемся: <Ctrl><Alt><F12>. Если мы хотим получать на экран сообщения не всех уровней, то эту строку перепишем по образцу:

```
kern.debug;kern.info;kern.notice;kern.warn              /dev/tty12
```

После этого нужно заставить демон перечитать правленый конфигурационный файл, для чего можно: а). перезапустить демон (что более хлопотно), б). направить ему сигнал `SIGHUP` (по такому сигналу многие из демонов Linux перечитывают и обновляют свою конфигурацию):

```
$ ps -Af | grep logd
root     14614     1  0 21:34 ?        00:00:00 /sbin/rsyslogd -c 4
root     14778 12935  0 21:37 pts/14   00:00:00 grep logd
# kill -HUP 14614
```

При этом в системном журнале (или в текстовой консоли вывода) вы должны увидеть строки:

```
# cat /var/log/messages | tail -n2
Apr  3 21:37:34 notebook kernel: imklog 4.4.2, log source = /proc/kmsg started.
Apr  3 21:37:34 notebook rsyslogd: [origin software="rsyslogd" swVersion="4.4.2" x-pid="14614" x-
info="http://www.rsyslog.com"] (re)start
```

Уровни диагностики в /proc

Ещё один механизм управления (динамического) уровнями диагностического вывода реализован через файловую систему `/proc`:

```
$ cat /proc/sys/kernel/printk
3       4       1       7
```

Где цифры показывают установленные пороги уровни вывода. Нас интересует первое значение — уровень сообщений `printk()`, начиная с которого и выше сообщения не будут выводиться на текстовую консоль (`KERN_ERR` и выше при показанных в примере значениях — не выводятся).

Записав в этот файл новое значение, можно изменить принимаемые по умолчанию значения. Но сделать это не так просто (из-за прав доступа к файлу):

```
$ echo 8 > /proc/sys/kernel/printk
bash: /proc/sys/kernel/printk: Отказано в доступе
$ sudo echo 8 > /proc/sys/kernel/printk
bash: /proc/sys/kernel/printk: Отказано в доступе
$ ls -l /proc/sys/kernel/printk
-rw-r--r-- 1 root root 0 Июн 13 16:09 /proc/sys/kernel/printk
```

Сделать это можно **только** с терминала с регистрацией под именем `root` :

```
# echo 8 > /proc/sys/kernel/printk
$ cat /proc/sys/kernel/printk
8       4       1       7
```

После такой переустановки диагностический модуль (архив `log_level.tgz`, мы его не обсуждаем детально) выведет на **текстовую** консоль:

```
# insmod log_level.ko
message level: <7>
message level: <6>
message level: <5>
message level: <4>
message level: <3>
message level: <2>
message level: <1>
insmod: error inserting 'log_level.ko': -1 Operation not permitted
```

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.

`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 реализаций 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.

Ошибка (сообщение) может возникнуть и при попытке выгрузить модуль. Более того, обратите внимание, что прототип функции выгрузки модуля `void module_exit( void )` - не имеет возможности вернуть код 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]
...
```

Обсуждение

Здесь и далее, некоторые разделы будут завершаться короткими текстами под названием «Обсуждение». Там, где они есть (а есть они далеко не ко всем разделам), это не непосредственные итоги предыдущего обсуждения, а скорее, некоторые отступления в сторону «вынесенные за скобки», те попутные мысли, которые возникали при обсуждении, но не сформулировались в законченную форму: иногда это дополнительные рутинные детали, важные на практике, иногда вопросы по обсуждённому материалу, или дополнения к нему...

Итак, мы только что создали первый свой загружаемый модуль ядра. К этому времени можно взять на заметку следующее:

1. Программирование модуля ядра ничем принципиально не отличается от программирования в пространстве пользователя. Однако, для обеспечения функциональности модуля мы используем другой набор API (`printk()`, например, вместо привычного `printf()`). Такая дуальность вызовов будет наблюдаться практически для всех разделов API (управление памятью, примитивы синхронизации, ...), но имена и форматы вызовов API ядра будут отличаться. Относительно API пространства пользователя существуют стандарты

(POSIX и др.) и они хорошо описаны в литературе. API пространства ядра плохо описаны, и могут существенно изменяться от одной версии ядра к другой (главным образом, именно из-за отсутствия таких сдерживающих стандартов, как в API пространства пользователя). Поиск адекватных вызовов API для текущей версии ядра и есть одной из существенных трудностей программирования модулей ядра Linux. Мы ещё неоднократно вернёмся к этим вопросам по ходу дальнейшего текста.

2. В примерах выше, и далее по всему тексту, команды загрузки модуля я часто буду записывать вот так:

```
$ sudo insmod ./hello_printk.ko
```

Это связано с тем, что команда `insmod`, в отличие от подобной по действию команды `modprobe`, на которой мы остановимся позже, не предполагает какого либо поиска указанного ей к загрузке **файла** модуля: либо этот файл указан со своим полным путевым именем (абсолютным или относительным) и тогда файл загружается, либо файл не находится по такому имени, и тогда `insmod` аварийно завершается:

```
$ sudo insmod zzz.ko
insmod: can't read 'zzz.ko': No such file or directory
```

Указание в примере выше имени файла модуля: `./hello_printk.ko` и есть указание корректного полного относительного путевого имени. Однако успешной будет загрузка модуля и командой, записанной в такой форме:

```
$ sudo insmod hello_printk.ko
$ lsmod | grep hello
hello_printk            557  0
```

Это происходит потому, что утилита `insmod` открывает файл (модуля), указанный ей в качестве параметра вызовом API `open()`, который может[5] интерпретировать имя файла, записанное без указания пути, как файл в текущем рабочем каталоге.

Примечание: Это несколько противоречит интуитивной **аналогии** загрузки модуля и запуска процесса (интерпретатором `bash`) пользовательского приложения, которые подобны по своему назначению: выполнение некоторого программного кода, содержащегося в указанном файле. При запуске пользовательского приложения, файл запрашиваемого приложения разыскивается последовательным перебором по списку путевых имён в переменной окружения `PATH`, когда командный интерпретатор (`bash`) выполняет системный вызов `execve()`:

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

Поскольку, обычно, путь «текущий каталог» (`./`) не включен явно в список путей `PATH`, то запуск процесса без обозначения в команде запуска текущего каталога завершится ошибкой (поиска). При загрузке же (выполнении инсталляционного кода) модуля, запускающая утилита запрашивает имя файла модуля вызовом `open()`, допуская указание имени файла без ссылки на текущий каталог, но и не выполняя никакого поиска в случае неудачи. Подтвердить это можно простейшим экспериментом:

```
$ echo $PATH
...:/usr/local/sbin:...
$ ls /usr/local/sbin
$ sudo mv hello_printk.ko /usr/local/sbin
$ ls /usr/local/sbin
hello_printk.ko
$ sudo insmod hello_printk.ko
insmod: can't read 'hello_printk.ko': No such file or directory
```

Как полезное следствие из рассмотрения этого пункта: файл модуля может быть загружен из любого места файловой системы, если вы знаете и точно указываете путевое имя этого файла в любой удобной вам форме, например:

5 Ни man-страница (`$ man 2 open`), ни стандарт POSIX, ни описания в литературе не регламентируют строго поведение `open()` при указании имени файла без путевого имени, в Linux это интерпретируется как файл в текущем рабочем каталоге, но это не значит, что так же будет в других POSIX системах.

```
$ sudo insmod ~/hello_printk.ko
$ sudo insmod $HOME/hello_printk.ko
```

3. Ещё одна особенность (которая досаждает поначалу при работе с модулями и о которой приходится помнить), это то, что при установке модуля мы говорим:

```
# insmod hello_printk.ko
```

Но при выгрузке (остановке) этого же модуля, мы **должны** сказать:

```
# rmmod hello_printk
```

- то есть, без путевого имени и расширения имени. Это связано с тем, что в этих родственных командах мы под подобным написанием указываем совершенно разные сущности: при установке — **имя файла** из которого должен быть установлен модуль ядра, а при выгрузке — **имя модуля** в RAM пространстве ядра (уже установленного и известного ядру), которое по написанию только совпадает с именем файла.

Обращаем внимание ещё на одну особенность: как бы мы не переименовывали **имя файла** уже ранее скомпилированного модуля, имя, под которым модуль **будет известным системе**, останется неизменным:

```
$ cp hello_printk.ko hello_printk_0.ko
$ sudo insmod hello_printk_0.ko
$ sudo rmmod hello_printk_0
ERROR: Module hello_printk_0 does not exist in /proc/modules
$ lsmod | head -n3
Module                  Size  Used by
hello_printk             557  0
fuse                   48375  2
$ sudo rmmod hello_printk
```

Другими словами, имя файла, которое указывается команде `insmod` в качестве параметра, является только **именем контейнера**, из которого загружается код модуля; но имя модуля, известное системе, содержится в самом бинарном **коде** модуля. Это ещё один уровень защиты целостности системы, о которых мы ещё будем говорить позже.

Архитектура и вокруг...

«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».

Для ясного понимания чем является модуль для ядра, необходимо вернуться к рассмотрению того, как пользовательские процессы взаимодействуют с сервисами ядра, что представляют из себя системные вызовы, и какие интерфейсы возникают в этой связи от пользователя к ядру, или к модулям ядра, представляющим функциональность ядра.

Ядро: монолитное и микроядро

«... message passing as the fundamental operation of the OS is just an exercise in computer science masturbation. It may feel good, but you don't actually get anything done.»

Linus Torvalds

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, ...), 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).

Примечание: Это же свойство обеспечивает и экстремально высокие живучесть и устойчивость микроядерных систем по сравнению с моноядерными: вышедший из строя драйвер можно перезагрузить не останавливая систему. Так что с утверждением Линуса Торвальдса, процитированным в качестве эпиграфа, можно было бы согласиться (и то с некоторой натяжкой) ... да и то, если бы в природе не существовало такой операционной системы как QNX, которая уже одна своим существованием оправдывает существование микроядерной архитектуры. Но это уже совсем другая история, а сегодня мы занимаемся исключительно Linux.

= = = = = = = = = =

здесь Рис. 1а: системный вызов в моноядерной ОС.

==========

==========

здесь Рис. 1б: системный вызов в микроядерной ОС.

==========

В макроядерной архитектуре все услуги для прикладного приложения выполняют отдельные ветки кода внутри ядра (в пространстве 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).

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 рассмотрения 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 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 перейти к более детальному рассмотрению прохождения 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).

==========

здесь Рис.2 : системный вызов Linux

==========

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 С-библиотека:

```
$ ls -l /lib/libc.*
lrwxrwxrwx 1 root root 14 Мар 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, 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 трансфорваться 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).

В этом смысле очень показательно наблюдаемое разделение (упорядочение) справочных страниц системы Linux по секциям:

```
$ man man
...
       The standard sections of the manual include:
      1      User Commands
      2      System Calls
      3      C Library Functions
      4      Devices and Special Files
      5      File Formats and Conventions
      6      Games et. Al.
      7      Miscellanea
      8      System Administration tools and Deamons
```

Таким образом, в подтверждение выше сказанного, справочную информацию по библиотечным функциям мы должны искать в секции 3, а по системным вызовам — в секции 2:

```
$ man 3 printf
...
$ man 2 write
...
```

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
...
```

Кроме `syscall()` Linux поддерживает и другой механизм системного вызова — `lcall7()`, устанавливая шлюз системного вызова, так, чтобы поддерживать стандарт iBCS2 (Intel Binary Compatibility Specification),

благодаря чему на x86 Linux может выполняться **бинарный код**, подготовленный для операционных систем FreeBSD, Solaris/86, SCO Unix. Больше мы этот механизм упоминать не будем.

Системные вызовы `syscall()` в Linux **на процессоре x86** выполняются через прерывание `int 0x80`. Соглашение о системных вызовах в Linux отличается от общепринятого в Unix 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. Рассмотрим, для подтверждения этого, сравнительные определения некоторых (подобных) системных вызовов (последних в таблице системных вызовов) для двух разных архитектур (в исходных кодах ядра 2.6.37.3):

Архитектура x86:

```
$ cat /usr/src/linux-2.6.37.3/arch/x86/kernel/syscall_table_32.S | tail -n10
        .long sys_pipe2
        .long sys_inotify_init1
        .long sys_preadv
        .long sys_pwritev
        .long sys_rt_tgsigqueueinfo   /* 335 */
        .long sys_perf_event_open
        .long sys_recvmmsg
        .long sys_fanotify_init
        .long sys_fanotify_mark
        .long sys_prlimit64           /* 340 */
```

Архитектура ARM:

```
$ cat /usr/src/linux-2.6.37.3/arch/arm/kernel/calls.S | tail -n 20
                CALL(sys_epoll_create1)
                CALL(sys_dup3)
                CALL(sys_pipe2)
/* 360 */       CALL(sys_inotify_init1)
                CALL(sys_preadv)
                CALL(sys_pwritev)
                CALL(sys_rt_tgsigqueueinfo)
                CALL(sys_perf_event_open)
/* 365 */       CALL(sys_recvmmsg)
                CALL(sys_accept4)
                CALL(sys_fanotify_init)
                CALL(sys_fanotify_mark)
                CALL(sys_prlimit64)
...
```

- последний (по номеру) системный вызов `sys_prlimit64` имеет в x86 номер 340, а в ARM — 369. И, если мы уж отклонились в рассмотрение отдельных архитектур, посмотрим, какие архитектуры поддерживаются этим ядром 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
```

Возьмите на заметку ещё, 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
```

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;
};
```

```
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, мы такую возможность поверхностно рассмотрим позже, когда перейдём к обзору инструментария модульного программирования, но пример и так прост, легко читается и интуитивно понятен: последовательно загружаются регистры значениями из переменных С кода, и вызывается прерывание. 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 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 между Intel платформами IA-32 и IA-64, а запись и набор регистров 64-бит IA-64 отличается от AMD64. Запись, использующая `syscall()`, будет корректно компилироваться в соответствии указанной целевой платформы.

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 модуля (`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** результатов 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))
```

Отличия программ пространств ядра и пользователя

Того, что мы достигли в обсуждении, уже достаточно для того, чтобы подвести некоторые итоги, и систематизировать основные отличия от пространства пользовательских процессов, которые нас будут подстерегать в пространстве ядра [26]:

1. Это самое главное отличие для программиста, пишущего код: ядро не имеет доступа к стандартным библиотекам языка C (как, собственно, и к любым другим библиотекам). Причин на то много, и их обсуждение не входит в наши планы. А как следствие, ядро оперирует со своим собственным набором API, отличающимся от POSIX API (отличающихся набором функций, их наименованиями). Мы это видели 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 — 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.

Как следствие этой двойственности 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.

Большинство механизмов ядра по своей функциональности сильно напоминают дуальные им и гораздо лучше известные механизмы POSIX, но специфика исполнения их в ядре (и ещё историческая преемственность) накладывает на них отпечаток на форму API, делая вызовы отличающимися как по наименованию, так и по формату. Иногда очень помогает отслеживание аналогичных вызовов пространств пользователя и ядра, примеры только некоторых из них собраны в табл.1 и они красноречиво говорят сами за себя.

2. 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.

3. 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).

4. 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 ia-32 систем и 16 Кбайт для ia-64.

Интерфейсы модуля

Модуль ядра является некоторым согласующим элементов потребностей в пространстве пользователя с возможностями, обеспечивающими эти потребности в пространстве ядра. Модуль (код модуля) может иметь (и пользоваться 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 (внаружу) и в направлении механизмов (внутрь) ядра.

==========

здесь Рис. : место модуля в системе Linux.

==========

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 именем 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 C 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.

Взаимодействие модуля с уровнем пользователя

Если с интерфейсом модуля в сторону ядра всё относительно единообразно, то вот с уровнем пользователя (командами, приложениями, системными файлами, внешними устройствами и другое разное, то что заметно пользователю) у модуля есть много разнообразных 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), поэтому мы должны его здесь выделить из числа прочих API ядра и отметить.

2. **Копирование данных** в программном коде между пользовательским адресным пространством и пространством ядра (выполняется только по инициативе модуля). Конечно, опять же, это всё те же вызовы из числа API ядра, но эти четыре вызова 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 put_user
c05c634c T __put_user_1
c05c6360 T __put_user_2
c05c6378 T __put_user_4
c05c6390 T __put_user_8
...
$ cat /proc/kallsyms | grep get_user
...
c05c628c T __get_user_1
c05c62a0 T __get_user_2
c05c62b8 T __get_user_4
...
$ cat /proc/kallsyms | grep copy_to_user
...
c05c6afa T copy_to_user
...
$ cat /proc/kallsyms | grep copy_from_user
...
c04abfeb T iov_iter_copy_from_user
c04ac053 T iov_iter_copy_from_user_atomic
...
c05c69e1 T copy_from_user
...
```

Вызовы `put_user()` и `get_user()` - это макросы, которые пытаются определить размер пересылаемой порции данных (1, 2, 4 байта - для `get_user()`; 1, 2, 4, 8 байт - для `put_user()`[6]). Вызовы `copy_to_user()` и `copy_from_user()` являются вызовами API ядра для данных произвольного размера, но они просто используют в цикле `put_user()` и `get_user()`, соответственно, нужное им число раз. Определения всех этих API можно найти в `<asm/uaccess.h>`, 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( void *x, void __user *ptr );
int get_user( void *x, const void __user *ptr );
```

Каждый из этих вызовов возвращает реально скопированное число байт или код ошибки операции (отрицательное значение). В комментариях утверждается, что передача коротких порций данных через `put_user()` и `get_user()` будет осуществляться быстрее.

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. Будет детально рассмотрено

6 Почему такая асимметрия я не готов сказать.

далее.

4. Взаимодействие через файлы (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 управляющую. Эти файлы в /proc доступны для чтения-записи всеми стандартными командами Linux (в пределах регламента прав доступа, установленных для конкретного файлового имени). Будет детально рассмотрено далее.

5. Взаимодействие через файлы (имена) системы /sys (файловая система sysfs). Эта файловая система подобна (по назначению) /proc, но возникла заметно позже, считается, что её функциональность выше, и она во многих качествах будет заменять /proc. Будет детально рассмотрено далее.

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 других протокольных семейств). Будет детально рассмотрено далее.

Загрузка модулей

Утилита insmod получает **имя файла модуля**, и пытается загрузить его без проверок взаимосвязей, как это описано ниже. 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 :

```
$ 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.

Примечание: если загрузка модуля производится непосредственно утилитой `insmod`, указанием ей **имени файла модуля**, то утилита никакие зависимости не проверяет, и, если обнаруживает неразрешённое имя — завершает загрузку аварийно.

Утилита `rmmod` выгружает ранее загруженный модуль, в качестве параметра утилита должна получать **имя модуля** (не **имя файла модуля).** 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, естественно, 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 );
```

- 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, в). права доступа (к параметру, отображаемую как путевое имя в системе `/sys`).

Значения параметрам могут быть установлены во время загрузки модуля через `insmod` или `modprobe`; последняя команда также можете прочитать значение параметра 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)
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;
module_param( iparam, int, 0 );

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

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

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

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

static int __init mod_init( void ) {
   int j;
   char msg[ 40 ] = "";
   printk( KERN_INFO "=======================================\n" );
   printk( KERN_INFO "iparam = %d\n", iparam );
   printk( KERN_INFO "nparam = %d\n", k );
   printk( KERN_INFO "sparam = %s\n", sparam );
   printk( KERN_INFO "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( KERN_INFO "%s\n=======================================\n", msg );
   return -1;
}

module_init( mod_init );
```

В коде этого модуля присутствуют две вещи, которые могут показаться непривычным программисту на языке C, нарушающие стереотипы этого языка, и по началу 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 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, а затем с переопределением значений всех параметров:

```
$ sudo /sbin/insmod ./mod_params.ko
insmod: error inserting './mod_params.ko': -1 Operation not permitted
$ dmesg | tail -n7
======================================
iparam = 0
nparam = 0
sparam =
cparam =  {0}
aparam [ 5 ] =  0  0  0  0  0
======================================
$ sudo /sbin/insmod ./mod_params.ko iparam=3 nparam=4 sparam=str1 \
cparam=str2 aparam=5,4,3
insmod: error inserting './mod_params.ko': -1 Operation not permitted
$ dmesg | tail -n7
======================================
iparam = 3
nparam = 4
sparam = str1
cparam = str2 {4}
aparam [ 3 ] =  5  4  3
======================================
```

- массив `aparam` получил новую размерность `arnum`, и присвоены значения его элементам.

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:

```
$ sudo /sbin/insmod ./mod_params.ko aparam=5,4,3,2,1,0
insmod: error inserting './mod_params.ko': -1 Invalid parameters
$ dmesg | tail -n2
aparam: can only take 5 arguments
mod_params: `5' invalid for parameter `aparam'
```

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

```
$ 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'
```

- не определённый в модуле параметр.

```
$ sudo /sbin/insmod ./mod_params.ko iparam=qwerty
```

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.

```
insmod: error inserting './mod_params.ko': -1 Invalid parameters
$ dmesg | tail -n1
mod_params: `qwerty' invalid for parameter `iparam'
```

- попытка присвоения не числового значения числовому типу.

```
$ sudo /sbin/insmod ./mod_params.ko cparam=123456789
insmod: error inserting './mod_params.ko': -1 No space left on device
$ dmesg | tail -n2
cparam: string doesn't fit in 4 chars.
mod_params: `123456789' too large for parameter `cparam'
```

- превышена максимальная длина для строки, передаваемой копированием.

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 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.

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

- 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
#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"
```

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

config.c :

8 Тема подсказана одним из читателей рукописи.

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

```
#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 очень путанных по описаниям) понятий из сферы модулей есть подсчёт ссылок использования модуля. 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 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 будет невозможно `i2c_core`.

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 осторожно, поскольку если мы увеличим счётчик и симметрично его позже не уменьшим, то мы 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`).

Обсуждение

Из этой части рассмотрения мы можем уже вынести следующие заключения:

1. Программирование модулей ядра Linux - это не только создание драйверов специфических устройств, но это вообще более широкая область: динамическое расширение функциональности ядра, добавление возможностей, которыми ранее ядро не обладало.

2. Программирование модулей ядра Linux так, чтобы принципиально, не отличается во многом от программирования в пространстве процессов. Однако, для его осуществления невозможно привлечь существующие в пространстве пользователя POSIX API и использовать библиотеки; поэтому в пространстве ядра предлагаются «параллельные» API и механизмы, большинство из них дуальны известным механизмам POSIX, но специфика исполнения в ядре (и историческая преемственность) накладывает на них отпечаток, что делает их отличающимися как по наименованию, так и по формату вызова и функциональности. Интересно отследить несколько аналогичных вызовов пространств пользователя и ядра, и рассмотреть их аналогичность — вот только некоторые из них:

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. Если по POSIX API существуют многочисленные обстоятельные справочники, то по именам ядра (вызовам и структурам данных) таких руководств нет. А общая размерность имён ядра (`/proc/kallsyms`) приближается к 100000, из которых до 10000 — это экспортируемые имена ядра (как будет подробно показано далее). Рассмотреть описательно такой объём не представляется возможным, а единственным путём, который обещает успех в освоении техники модулей ядра, видится максимальный объём законченных примеров работающих модулей «на все случаи жизни». Таким построением изложения мы и воспользуемся.

Окружение и инструменты

«Учитель, всегда нужно знать, куда попал, если, стреляя в цель, промахнулся!»

Милорад Павич «Вывернутая перчатка».

Прежде, чем переходить к детальному рассмотрению кода модулей и примеров использования, бегло посмотрим на тот инструментарий, который у нас есть в наличии для такой деятельности.

Основные команды

Вот тот очень краткий «джентльменский набор» специфических команд, требуемых наиболее часто при работе с модулями ядра (что не отменяет требования по применению достаточно широкого набора общесистемных команд Linux):

```
# sudo insmod ./hello_printk.ko
```

- загрузка модуля в систему.

```
# rmmod hello_printk
```

- удаление модуля из системы.

```
# modprobe hello_printk
```

- загрузка модуля, ранее установленного в систему, и всех модулей, требуемых его зависимостями.

```
$ modinfo ./hello_printk.ko
```

- вывод информации о **файле** модуля.

Команды `rmmod, modprobe` требуют указания **имени модуля**, а команды `insmod, modinfo` - указания **имени файла** модуля.

```
$ lsmod
```

- список установленных модулей и их зависимости.

```
# depmod
```

- обновление зависимостей модулей в системе.

```
$ dmesg
```

- вывод системного журнала, в том числе, и сообщений модулей.

```
# cat /var/log/messages
```

- вывод системного журнала, в том числе, и сообщений модулей, формат отличается от `dmesg`, требует прав `root`.

```
$ nm mobj.ko
```

- команда, дающая нам список имён объектного файла, в частности, имён модуля с глобальной сферой видимости (тип T) и имён, импортируемых данным модулем для связывания (тип U).

```
$ objdump -t hello_printk.ko
```

- детальный анализ объектной структуры модуля.

```
$ readelf -s hello_printk.ko
```

- ещё один инструмент анализа объектной структуры модуля.

Системные файлы

Краткий перечень системных файлов, на которые следует обратить внимание, работая с модулями:

`/var/log/messages` — журнал системных сообщений, в том числе и сообщений модулей ядра.

`/proc/modules` — динамически создаваемый (обновляемый) список модулей в системе.

`/proc/kallsyms` — **динамически** создаваемый список имён ядра, формата: <адрес> <имя>.

`/boot/System.map-`uname -r`` - файл с именем вида: `/boot/System.map-2.6.32.9-70.fc12.i686.PAE` — содержит **статическую** таблицу имён ядра для образа, с которого загружена система, эта таблица может несколько отличаться от `/proc/kallsyms`, посмотреть таблицу можно так:

```
cat /boot/System.map-`uname -r` | head -n3
00000000 A VDSO32_PRELINK
00000000 A xen_irq_disable_direct_reloc
00000000 A xen_save_fl_direct_reloc
...
```

`/proc/slabinfo` — динамическая детальная информация сляб-алокатора памяти (о нём мы будем говорить отдельно и подробно).

`/proc/meminfo` — сводная информация о использовании памяти в системе.

`/proc/devices` — список драйверов устройств, встроенных в действующее ядро.

`/proc/dma` — задействованные в данный момент каналы DMA.

`/proc/filesystems` — файловые системы, встроенные в ядро.

`/proc/interrupts` — список задействованных в данный момент прерываний.

`/proc/ioports` — список задействованных в данный момент портов ввода/вывода.

`/proc/version` — версия ядра в формате:

```
$ cat /proc/version
Linux version 2.6.32.9-70.fc12.i686.PAE (mockbuild@x86-02.phx2.fedoraproject.org) (gcc version
4.4.3 20100127 (Red Hat 4.4.3-4) (GCC) ) #1 SMP Wed Mar 3 04:57:21 UTC 2010
```

`/lib/modules/2.6.18-92.el5/build/include` — каталог такого вида (точный вид зависит от версии ядра) содержит все необходимые хэдер-файлы для включения определений в код модуля, и для получения справки; точный вид имени каталога можете получить так:

```
$ echo /lib/modules/`uname -r`/build/include
/lib/modules/2.6.18-92.el5/build/include
```

Подсистема X11, терминал и текстовая консоль

Ряд авторов утверждают, что графическая подсистема X11 не подходит как среда для разработки приложений ядра, для этого годится только текстовая консоль... хотя дальше они тут же сами отказываются от такого своего утверждения, и демонстрируют примеры, явно выполняемые в **графическом терминале** подсистемы X11. Таким образом, все подобные утверждения относятся, скорее, к области красивых народных легенд. Тем не менее, нужно отчётливо представлять соотношения текстовых и графических режимов (интерфейсов пользователя) в Linux, их особенности и ограничения.

= = = = = = = = = = =

здесь Рис.3 : место графической подсистемы X11 в системе Linux.

= = = = = = = = = = =

Графическая подсистема X11 (в реализациях X11R6 или Xorg) **не является** органичной (неотъемлемой) составной частью операционной системы Linux (UNIX), а является надстройкой пользовательского уровня (более того, для работы непосредственно с видео оборудованием использующей работу с видеоадаптером наборы API пространства пользователя, а не пространства ядра!). Это **принципиально отличает** Linux от систем семейства Windows. О графической системе X11, в наших целях, достаточно знать и постоянно помнить следующее:

а). Это надстройка над операционной системой, работающая в пользовательском адресном пространстве.

б). Протокол X (пользовательского уровня модели OSI), по которому взаимодействуют X-клиент (GUI приложения) и X-сервер (графическая подсистема), является **сетевым** протоколом; грубые нарушения в настройках и функционировании сетевой подсистемы могут приводить к полной потере работоспособности графической подсистемы (что очень удивляет пользователей из системы Windows).

в). Сетевой протокол X может использовать в качестве транспортного уровня альтернативно различные протоколы, в частности, это может быть не только TCP/IP, но и (взамен него) потоковый доменный протокол UNIX (UND).

г). Вывод (и ввод) на **терминал** (работающий в графической системе X11) проходит через множество промежуточных слоёв, в отличие от **текстовой консоли**, и может значительно отличаться по поведению при работе с программами ядра.

Далее, в силу её значимости для отработки программ ядра, возвратимся к чистой текстовой консоли. Число текстовых консолей (обычно по умолчанию 6) в Linux (в отличие, например, от FreeBSD) — величина легко изменяемая динамически (настройками, в ходе работы, без пересборки ядра). При работе именно с программами ядра число консолей может понадобится значительно увеличить... В некоторых более старых дистрибутивах (и других UNIX системах) используется хорошо описанный способ — конфигурационный файл `/etc/inittab`:

```
$ uname -r
2.6.18-92.el5
$ cat /etc/inittab
...
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
...
```

Значения полей следующие: идентификатор записи, уровень (или уровни) выполнения (runlevels), для которого эта запись имеет силу, акция, выполняемая при этом, и собственно исполняемая команда (в данном случае - команда авторизации консоли `mingetty`). Добавление новых строк будет давать нам новые консоли.

Но в некоторых новых дистрибутивах файл `/etc/inittab` практически пустой:

```
# uname -r
2.6.32.9-70.fc12.i686.PAE
# cat /etc/inittab
...
# Terminal gettys (tty[1-6]) are handled by /etc/event.d/tty[1-6] and
# /etc/event.d/serial
...
```

В этом варианте начальная инициализация консолей, как нам и подсказывает показанный комментарий, происходит в каталоге :

```
# ls /etc/event.d/tty*
tty1  tty2  tty3  tty4  tty5  tty6
# cat /etc/event.d/tty6
...
respawn
exec /sbin/mingetty tty6
...
```

Как и в предыдущем случае, создание дополнительных консолей очевидно: а). создайте новый файл `/etc/event.d/tty7` (и т. д.) , б). скопируйте в него содержимое `/etc/event.d/tty6` и в). подредактируйте в показанной строке номер соответствующего `tty`...

Для проверки того, сколько сейчас активных консолей, у вас в арсенале есть команда:

```
$ fgconsole
7
```

- 6 текстовых + X11, не удивляйтесь, если в некоторых дистрибутивах (новых) вы получите странный результат, например, число 3 : команда даёт число **открытых** консолей, на которых уже произведен logon!

Сколько много может быть создано текстовых консолей в системе? Максимальное число — 64, поскольку для устройств tty* статически зарезервирован диапазон младших номеров устройств до 63 :

```
$ ls /dev/tty*
/dev/tty    /dev/tty16  /dev/tty24  /dev/tty32  /dev/tty40  /dev/tty49  /dev/tty57  /dev/tty8
/dev/tty0   /dev/tty17  /dev/tty25  /dev/tty33  /dev/tty41  /dev/tty5   /dev/tty58  /dev/tty9
...
/dev/tty14  /dev/tty22  /dev/tty30  /dev/tty39  /dev/tty47  /dev/tty55  /dev/tty63
/dev/tty15  /dev/tty23  /dev/tty31  /dev/tty4   /dev/tty48  /dev/tty56  /dev/tty7
$ ls -l /dev/tty63
crw-rw---- 1 root tty 4, 63 Map 12 10:15 /dev/tty63
```

Последний вопрос: как бегло переключаться между большим числом консолей?

1. Посредством клавиатурной комбинации <Ctrl>+<Alt>+<Fi> - где i — номер функциональной клавиши: 1...12.

2. В режиме текстовой консоли во многих дистрибутивах по клавише PrintScreen включено «пролистывание» активизированных консолей, начиная с первой.

3. Самый универсальный способ — команда (смена виртуального терминала):

```
# chvt 5
```

- которая переносит нас в ту консоль, номер которой указан в качестве ее аргумента. Эта команда может потребовать `root` привилегий, и может вызвать недоумение сообщением:

```
$ chvt 2
chvt: VT_ACTIVATE: Операция не позволяется
```

Пример того, как получить информацию (если забыли) кто, как и где зарегистрирован в системе, и как эту информацию толковать:

```
$ who
root     tty2         2011-03-19 08:55
olej     tty3         2011-03-19 08:56
olej     :0           2011-03-19 08:22
olej     pts/1        2011-03-19 08:22 (:0)
olej     pts/0        2011-03-19 08:22 (:0)
olej     pts/2        2011-03-19 08:22 (:0)
olej     pts/3        2011-03-19 08:22 (:0)
olej     pts/4        2011-03-19 08:22 (:0)
olej     pts/5        2011-03-19 08:22 (:0)
olej     pts/6        2011-03-19 08:22 (:0)
olej     pts/9        2011-03-19 09:03 (notebook)
```

- здесь: а). 2 (строки 1, 2) регистрации в текстовых **консолях** (# 2 и 3) под разными именами (`root` и `olej`); б). X11 (строка 3) регистрация (консоль #7, CentOS 5.2 ядро 2.6.18); в). 7 открытых графических **терминалов** в X11, дисплей :0; г). одна удалённая регистрация по SSH (последняя строка) с компьютера с именем `notebook`.

Компилятор GCC

Исходный код модулей ядра в Linux, при всём богатстве языковых средств разработки в Linux для **прикладного** программирования, создаётся **исключительно** на языке С. Но написав программный код модуля, его ещё нужно и скомпилировать. Даже для одного только языка С в Linux могут использоваться, присутствуют и используются несколько различных компиляторов, как то:

- основной компилятор Linux `gcc`, компилятор из GNU-проекта GCC;

- компилятор `cc` из состава интегрированной среды разработки IDE Solaris Studio операционной системы Open Solaris (ещё недавно фирмы Sun Microsystems, а на сегодня фирмы Oracle);

- активно развивающийся в рамках проекта LLVM компилятор Clang (кандидат для замены `gcc` в операционной системе FreeBSD, причина чему — лицензия);

- PCC (Portable C Compiler) — новая реализация компилятора, развиваемого ещё с 70-х годов, получившая новую жизнь в операционных системах NetBSD и OpenBSD.

Все из них, и ещё некоторые из свободно развиваемых проектов, могут использоваться в Linux с успехом (утверждается, что код, компилированный IDE Solaris Studio в ряде случаев заметно производительнее компилированного `gcc`). Тем не менее, вся эта альтернативность возможна только **в проектах пользовательского адресного пространства**, в программировании ядра и, соответственно, модулей ядра, на сегодня применяется исключительно компилятор `gcc`. Причина этому — значительные синтаксические расширения `gcc` относительно стандартов языка С.

Примечание: Существуют экспериментальные проекты по сборке Linux компилятором, отличным от GCC. Есть сообщения о том, что компилятор Intel C имеет достаточную поддержку расширений GCC чтобы компилировать ядро Linux. Но при всех таких попытках пересборка может быть произведена только полностью, «с нуля»: начиная со сборки ядра и уже только потом сборка модулей. В любом случае, ядро и модули должны собираться одним компилятором.

Начало GCC было положено Ричардом Столлманом, который реализовал первый вариант GCC в 1985 на нестандартном и непереносимом диалекте языка Паскаль; позднее компилятор был переписан на языке Си

Леонардом Тауэром и Ричардом Столлманом и выпущен в 1987 как компилятор для проекта GNU (http://ru.wikipedia.org/wiki/GCC). Компилятор GCC имеет возможность осуществлять компиляцию:

- с нескольких языков программирования (точный перечень зависит от опций сборки самого компилятора `gcc`) ;
- в систему команд множества (нескольких десятков) процессорных архитектур;

Достигается это 2-х уровневым процессом: а). лексический анализатор (вариант GNU утилиты `bison`, от общей UNIX реализации анализатора `yacc`; в комплексе с лексическим анализатором `flex`) и б). независимый генератор кода под архитектуру процессора.

Одно из свойств (для разработчиков модулей Linux), отличающих GCC в положительную сторону относительно других компиляторов, это расширенная многоуровневая (древовидная) система справочных подсказок, включённых в саму утилиту `gcc`, начиная с:

```
$ gcc --version
gcc (GCC) 4.4.3 20100127 (Red Hat 4.4.3-4)
Copyright (C) 2010 Free Software Foundation, Inc.
...
```

И далее ... самая разная справочная информация, например, одна из полезных — опции компилятора, которые включены по умолчанию при указанном уровне оптимизации:

```
$ gcc -Q -O3 --help=optimizer
Следующие ключи контролируют оптимизацию:
  -O<number>
  -Os
  -falign-functions                           [включено]
  -falign-jumps                               [включено]
...
```

Для подтверждения того, что установки опций для разных уровней оптимизации отличаются, и уточнения в чём состоят эти отличия, проделаем следующий эксперимент:

```
$ gcc -Q -O2 --help=optimizer > O2
$ gcc -Q -O3 --help=optimizer > O3
$ ls -l O*
-rw-rw-r-- 1 olej olej 8464 Май  1 11:24 O2
-rw-rw-r-- 1 olej olej 8452 Май  1 11:24 O3
$ diff O2 O3
...
49c49
<   -finline-functions                        [выключено]
---
>   -finline-functions                        [включено]
...
```

Существует множество параметров GCC, специфичных для каждой из поддерживаемых целевых платформ, которые можно включать при компиляции модулей, например, в переменную `EXTRA_CFLAGS` используемую `Makefile`. Проверка платформенно зависимых опций может делаться так:

```
$ gcc --target-help
Ключи, специфические для целевой платформы:
...
  -m32                Генерировать 32-битный код i386
...
  -msoft-float        Не использовать аппаратную плавающую арифметику
  -msse               Включить поддержку внутренних функций MMX и SSE при генерации кода
  -msse2              Включить поддержку внутренних функций MMX, SSE и SSE2  при генерации кода
...
```

GCC имеет значительные синтаксические расширения (такие, например, как инлайновые ассемблерные

вставки, или использование вложенных функций), не распознаваемые другими компиляторами языка C — именно поэтому альтернативные компиляторы вполне пригодны для сборки приложений, но мало непригодны для пересборки ядра Linux и сборки модулей ядра.

Невозможно в пару абзацев даже просто назвать то множество возможностей, которое сложилось за 25 лет развития проекта, но, к счастью, есть исчерпывающее полное руководство по GCC более чем на 600 страниц, и оно издано в русском переводе [8], которое просто рекомендуется держать под рукой на рабочем столе в качестве справочника.

Ассемблер в Linux

Необходимо несколько слов сказать об ассемблере. Язык ассемблера практически никогда не бывает нужен **прикладному** программисту. Но разработчику модулей в отдельных случаях он может помочь, часто даже не столько для написания конечного кода, сколько для понимания того, с чем приходится иметь дело (в коде Linux ядра достаточно много ассемблерных инлайновых вставок), для экспериментов, для отладки, и поиска тонких ошибок и неисправностей. В сложных случаях иногда бывает полезным изучить ассемблерный код, генерируемый компилятором GCC как промежуточный этап компиляции, и убедиться, что ... компилятор вас неправильно понял. В конечном счёте, вопрос о том, нужно ли разработчику знание ассемблера должен разрешаться так: разработчик модулей не обязан писать на языке ассемблера, но он должен понимать, написанное на ассемблере..., а уж тем более этот код различать. Мы могли бы просто игнорировать ассемблерные возможности `gcc`, но некоторая часть наших иллюстрирующих примеров использует такой код, поэтому сделаем короткий экскурс в язык ассемблера.

В сложных случаях иногда бывает нужно изучить ассемблерный код, генерируемый GCC как промежуточный этап компиляции. Увидеть сгенерированный GCC ассемблерный код можно компилируя командой с ключами:

```
$ gcc -S -o my_file.S my_file.c
```

Примечание: Посмотреть результат ещё более ранней фазы препроцессирования можно, используя редко применяемый ключ `-E` :

```
$ gcc -E -o my_preprocessed.c my_file.c
```

Возможно использование ассемблерного кода для всех типов процессорных архитектур (x86, PPC, MIPS, AVR, ARM, ...) поддерживаемых GCC — но синтаксис записи будет **отличаться**.

Для генерации кода GCC вызывает `as` (раньше часто назывался как `gas`), конфигурированный под целевой процессор:

```
$ as --version
GNU assembler 2.17.50.0.6-6.el5 20061020
Copyright 2005 Free Software Foundation, Inc.
...
This assembler was configured for a target of `i386-redhat-linux'.
```

Нотация AT&T

Ассемблер GCC использует синтаксическую нотацию AT&T, в отличие от нотации Intel (которую используют, например, все инструменты Microsoft, компилятор C/C++ Intel, многоплатформенный ассемблер NASM).

Примечание: Обоснование этому простое - все названные инструменты, использующие нотацию Intel, используют её применительно к процессорам архитектуры x86. Но GCC является много-платформенным инструментом, поддерживающим не один десяток аппаратных платформ, ассемблерный код каждой из этих множественных платформ может быть записан в AT&T нотации.

В нотации AT&T строка записанная как:

```
movl %ebx, %eax
```

Выглядит в Intel нотации так:

```
mov eax, ebx
```

Основные принципы AT&T нотации, принципиально отличающие её от более привычной Intel нотации (идущей и известной ещё из MS-DOS):

1. Порядок операндов (направление выполнения операции): <Операция> <Источник>, <Приемник> (слева - направо) - в Intel нотации порядок обратный (справа - налево).
2. Названия регистров имеют явный префикс % указывающий, что это регистр. То есть `%eax, %dl, %esi, %xmm1` и так далее. То, что названия регистров не являются зарезервированными словами, — несомненный плюс: для различных процессорных архитектур это может быть самая различная запись, например: `%1, %2, ....` (VAX, Motorola 68000).
3. Обязательное явное задание размеров операндов в суффиксах команд: b-byte, w-word, l-long, q-quadword. В командах типа `movl %edx, %eax` это может показаться излишним, однако является весьма наглядным средством, когда речь идет о: `incl (%esi) или xorw $0x7, mask`
4. Названия констант начинаются с `$` и могут быть выражением. Например: `movl $1,%eax`
5. Значение без префикса означает адрес. Это еще один камень преткновения новичков. Просто следует запомнить, что:
   `movl $123, %eax` — записать в регистр `%eax` число 123,
   `movl 123, %eax` — записать в регистр `%eax` содержимое ячейки памяти с адресом 123,
   `movl var, %eax` — записать в регистр `%eax` **значение** переменной `var`,
   `movl $var, %eax` — загрузить **адрес** переменной var
6. Для косвенной адресации необходимо использовать круглые скобки. Например: `movl (%ebx), %eax` — загрузить в регистр `%eax` значение переменной, по адресу находящемуся в регистре `%ebx` .
7. SIB-адресация: смещение (база, индекс, множитель).

Однострочные примеры:

```
popw  %ax             /* извлечь 2 байта из стека и записать в %ax */
movl  $0x12345, %eax  /* записать в регистр константу 0x12345
movl  %eax, %ecx      /* записать в регистр %ecx операнд, который находится в регистре %eax */
movl  (%ebx), %eax    /* записать в регистр %eax операнд из памяти, адрес которого
                         находится в регистре адреса %ebx  */
```

Пример: Вот как выглядит последовательность ассемблерных инструкций для реализации системного вызова на `exit( EXIT_SUCCESS )` на x86 архитектуре:

```
movl  $1, %eax        /* номер системного вызова exit - 1   */
movl  $0, %ebx        /* передать 0 как значение параметра  */
      int $0x80       /* вызвать exit(0)                    */
```

Всё, что записано в программе на ассемблере, будь это ассемблерный отдельно компилируемый файл, или описываемые ниже инлайновые ассемблерные вставки — всё это будет **платформенно зависимо**! Начиная с самых основ, с синтаксиса такого фрагмента: символьные обозначения регистров на каждой платформе будут различаться. Более того, близкие платформы, не разделяемые обычно на пользовательском уровне, такие как x86 32-бит (IA-32) и 64-бит (IA-64), а также AMD64 — будут все между собой различаться в ассемблерной записи! Поэтому ассемблерный код более пригоден для изучения и иллюстрации происходящего, чем для практических целей. Если же это потребуется в рабочем проекте (о чём нужно тщательно продумать), то такие фрагменты прописываются раздельно для всех платформ, где проект может эксплуатироваться, и «обкладываются» соответствующими пропроцессорными `#ifdef` директивами.

Примечание: Во всех рассматриваемых примерах, здесь и везде далее, где для ясности понимания будет привлекаться ассемблерный код, он будет записан в синтаксической нотации 32-бит платформы x86.

Инлайновый ассемблер GCC

GCC Inline Assembly — встроенный ассемблер компилятора GCC, представляющий собой язык **макроописания** интерфейса компилируемого высокоуровнего кода с ассемблерной вставкой. Синтаксис инлайн

вставки в С-код - это оператор вида (в терминологии синтаксиса языка С — оператор, **отделяемый** от следующего символом `;`):

```
asm [volatile] ( "команды и директивы ассемблера" /* как последовательная текстовая строка */
                 : [<выходные параметры>] : [<входные параметры>] : [<изменяемые параметры>]
               );
```

Где, в простейшем виде:

- выходные параметры — это запись того, как значения регистров будут записаны в выходные переменные С кода **после** выполнения фрагмента, например: `"=a"(res)` — содержимое регистра `%eax` будет записано в целочисленную переменную `res`;

- входные параметры — это запись того, как переменные С кода будут занесены в регистры процессора **перед** выполнением фрагмента, например: `"b"((long)(str)),"c"((long)(len))` — значение указателя (адрес) строки `str` будет загружено в регистр `%ebx`, а целочисленное значение длины строки `len` в регистр `%ecx`;

- **изменяемые параметры** — это запись списка (через запятую) тех регистров, содержимое которых может портиться при выполнении фрагмента, так, чтобы компилятор мог принять меры по их хранению, например: `"%ecx", "%edi"` – два регистра процессора объявлены модифицируемыми в процессе выполнения фрагмента.

- **команды и директивы ассемблера** — это записанный в виде текстовой строки языка С фрагмент (может быть достаточно большой) программы, подлежащей макро преобразованию в язык ассемблера; если это многострочный фрагмент программы, то его строки завершаются ограничителями `\n` или `\n\t`.

В простейшем случае, когда фрагмент ассемблера не требует параметров, это может быть простейшая запись вида:

```
asm [volatile] ( "команды ассемблера" );
```

Примеры:

1. То, как записать несколько строк инструкций ассемблера:

```
asm volatile( "nop\n"
              "nop\n"
              "nop\n"
            );
```

2. Пример выполнения системного вызова `write()`, (показанный ранее в архиве `int80.tgz`):

```
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;
}
```

3. Вот как в форме инлайнового включения в программе определяется макровызов с параметрами (произвольный в общем случае, умножения на 5 конкретно в показанном примере). Этим вводится определение нового функционального вызова, который вызывается позже из С-кода простой и привычной записью вызова `times5( n, n )`:

```
#define times5(arg1, arg2) \
__asm__ ( "leal (%0,%0,4),%0" \
          : "=r" (arg2) \
          : "r" (arg1) );
```

Для чего в случае `asm` служит ключевое слово `volatile`? Для того чтобы указать компилятору, что вставляемый ассемблерный код может давать побочные эффекты, поэтому попытки оптимизации могут привести к логическим ошибкам.

Ещё один, более сложный пример - как в ядре записан макрос системного вызова с 4-мя параметрами, как это было записано в одной из предыдущих версий ядра (очень рекомендовал бы здесь остановиться на мгновение и проанализировать написанное, чтобы понимать что из себя представляет код ядра Linux):

```
#define _syscall4(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4) \
type name(type1, arg1, type2, arg2, type3, arg3, type4, arg4) \
{\
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name), "b" ((long)(arg1)), "c" ((long)(arg2)), \
          "d" ((long)(arg3)), "S" ((long)(arg4)) ); \
__syscall_return(type,_res); \
}
```

Пример использования ассемблерного кода

Примеры разнообразного использования ассемблерного кода собраны в архивы примеров `gas-prog.tgz` и `as-inline-8.tgz`. Для сравнения того, как внешне выглядит функционально идентичный код, записанный на C (`gas2_0.c`), в виде ассемблерного файла (`gas2_1.c`) и инлайновой ассемблерной вставки (`gas2_2.c`), рассмотрим такой пример (`архив gas-prog.tgz`); прежде всего его сценарий сборки :

Makefile :

```
LIST = gas1 gas2_0 gas2_1 gas2_2

all:    $(LIST)

gas2_1: gas2_1.c exit.S
        gcc -c gas2_1.c -o gas2_1.o
        gcc -c exit.S -o exit.o
        gcc gas2_1.o exit.o -o gas2_1
        rm -f *.o

# gas2_0 и gas2_2 собираются по умолчанию на основании суффикса, и не требуют целей
```

И далее сами файлы реализации:

gas2_0.c :

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] ) {
   printf( "----- begin prog\n" );
   int ret = 7;
   exit( ret );
   printf( "----- final prog\n" );
   return 0;    // never!
};
```

gas2_1.c :

```
#include <stdio.h>

extern void asmexit( int retcod );
int main( int argc, char *argv[] ) {
   printf( "----- begin prog\n" );
   int ret = 7;
   asmexit( ret );
   printf( "----- final prog\n" );
   return 0;    // never!
};
```

exit.S :

```
#  коментарий может начинаться или с # как AT&T,
```

```
// так и ограничиваться как в C: // & /* ... */
/* void asmnexit( int retcod ); */
.globl asmexit
        .type   asmexit, @function
asmexit:
        pushl   %ebp            // соглашение о связях
        movl    %esp, %ebp
        movl    $1, %eax
        movl    8(%ebp), %ebx
        int     $0x80
        popl    %ebp            // соглашение о связях
        ret
```

gas2_2.c :

```
#include <stdio.h>

int main( int argc, char *argv[] ) {
   printf( "----- begin prog\n" );
   int ret = 7;
   asm volatile (
     "movl  $1, %%eax\n"
     "movl  %0, %%ebx\n"
     "int $0x80\n"
     : : "b"(ret) : "%eax"
   );
   printf( "----- final prog\n" );
   return 0;    // never!
};
```

Убеждаемся, что по исполнению все три варианта абсолютно идентичные:

```
$ ./gas2_0
----- begin prog
$ echo $?
7
$ ./gas2_1
----- begin prog
$ echo $?
7
$ ./gas2_2
----- begin prog
$ echo $?
7
$ echo $?
0
```

Вполне достаточного уровня описания ассемблерного программирования в Linux (которое, вообще-то говоря, вещь гораздо более редкая, чем в мире DOS / Windows) можно найти в [27], [28].

В деталях о сборке

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

Параметры компиляции

Параметры компиляции модуля можно существенно менять, изменяя переменные, определённые в скрипте, осуществляющем сборку, 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 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 определяется путь к корневому каталогу исходных файлов сборки, так 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
```

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.

```
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 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 имя 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
dev  exec  first_hello  IRQ  Makefile  net  pci   signal  sys_call_table  time   tree.txt  user_space
dma  file  int80        load_module    memory    netproto  proc  sys     thread  tools  usb
```

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

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, 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` и б). обновить информацию о 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!

Нужна ли новая сборка ядра?

Это первый вопрос всякого, кто приступает к разработке собственного модуля ядра. Из уже сказанного ранее (и что не раз будет подтверждаться деталями дальнейшего изложения) должно уже быть понятно: для

11 Предложен одним из читателей рукописи книги.

компиляции и сборки самого элементарного модуля необходима некоторая информация, генерируемая в ходе компиляции и сборки ядра. А именно: **абсолютные адреса** экспортируемых имён **ядра и модулей**, по которым происходит связывание имён в компилируемом модуле (те адреса, которые мы наблюдаем в `/proc/kallsyms`).

Для того, чтобы объяснить происходящее, возьмём в исследование произвольный символ экспортируемый ядро... Символы **экспортируемые** ядром, вместе с их абсолютными **адресами в пространстве ядра**, после сборки ядра записаны статически в файл `/lib/modules/`uname -r`/build/System.map`:

```
$ cat /lib/modules/`uname -r`/build/System.map | grep ' T ' | tail -n5
c0c8c259 T crypto_fpu_exit
c0c8c4a3 T crypto_exit_proc
c0c8ce66 T libata_transport_exit
c0c8d101 T xhci_unregister_pci
c0c8d232 T rtc_dev_exit
```

Выберем последнее из показанных имён `rtc_dev_exit()` (я не знаю, что оно означает, и для нашего рассмотрения это не имеет значения). То же имя из динамически формируемой таблицы `/proc/kallsyms`:

```
$ cat /proc/kallsyms | grep rtc_dev_exit
c0c8d232 T rtc_dev_exit
```

Пока всё нормально: адрес символа взятый статически при сборке модуля из файла `System.map` совпадает с его адресом, формируемым динамически (фактическим адресом размещения). Теперь поищем тот же символ в таблицах другой, достаточно близкой версии ядра, для чего просто заглянем в соседний каталог:

```
$ cat /lib/modules/2.6.35.14-106.fc14.i686.PAE/build/System.map | grep rtc_dev_exit
c0ac095d T rtc_dev_exit
```

- это **совершенно другой адрес**, и ясно, что если мы по данным той (2.6.35) системы соберём модуль использующий вызов `rtc_dev_exit()`, для использования в текущей (2.6.42), или наоборот (2.6.42 для 2.6.35), то исполнение такого модуля просто вдребезги разнесёт ядро Linux. Более того, та же история произойдёт и если мы заново соберём **то же самое** ядро, изменив при его конфигурировании один из великого множества параметров — все адреса точек входов «поплывут».

В итоге, мы приходим к выводу, что для сборки любого своего простейшего модуля ядра необходимы данные сборки самого ядра. Это плата за монолитность ядра Linux! **В самом общем случае**, для сборки модуля ядра необходимо собрать само ядро. Точно так же поступают и пакеты кросс-построения ядра и модулей для иных процессорных архитектур (ARM, MIPS etc.), например, пакет BuildRoot.

Ещё одним компонентом, обязательно присутствующим для сборки модуля является **дерево** файлов определений (.h) для текущей версии ядра (которые могут отличаться и по составу и по прототипам даже для ближайших ядер). Такое дерево должно быть установлено (и может быть найдено) в `/lib/modules/`uname -r`/build/include`, самую актуальную часть определений ищем в каталоге `linux` этого дерева.

Но компиляция текущего ядра из исходных кодов (которая детальнее описана в приложении):

- трудоёмкий процесс, который может потребовать от 20-30 мин. Непрерывного процессорного времени (на самых быстрых процессорах) до нескольких часов (на более старых компьютерах);
- сам процесс сборки может привнести ошибки соответствия текущему рабочему ядру — за счёт расхождений конфигурационных параметров, что создаёт очень большие опасности.

Поэтому дистрибьюторы основных пакетных дистрибутивов Linux включают пакеты, достаточные для работы с модулями ядра. При этом для сборки и отработки модулей ядра перекомпиляция самого ядра (и загружаемого образа системы), в обязательном порядке, - **не нужна**. Один из таких пакетов — `kernel-devel-*` (который вы должны установить для работы с модулями):

```
$ rpm -ql kernel-devel-* | grep System
/usr/src/kernels/2.6.35.14-106.fc14.i686.PAE/System.map
/usr/src/kernels/2.6.42.12-1.fc15.i686.PAE/ System.map
$ rpm -ql kernel-PAE-devel.i686 > | grep System
/usr/src/kernels/3.3.0-8.fc16.x86_64/System.map
```

- этот пакет устанавливает всю (пустую) иерархию дерева сборки ядра и все необходимые результирующие данные сборки (`System.map` и др.) такого ядра.

Кроме того, как уже сказано было, для работы с модулями необходимо наличия **заголовочных файлов ядра** (в точности соответствующих загруженной версии ядра!), сам же исходный программный код системы — **не нужен**. Эти заголовочные файлы устанавливаются другим пакетом из той же группы: `kernel-headers-*`. Обычно заголовочные файлы, необходимые для разработки модулей, присутствуют в вашей системе (это определяется предпочтениями дистрибьюторов вашей Linux системы). Но может оказаться, что это и не так, в этом случае **символьная ссылка** `/lib/modules/`uname -r`/build` окажется неразрешённой, а каталог кодов ядра пустой:

```
$ ls /usr/src/kernels
$
```

В любом случае нужно либо доустановить, либо проверить на наличие, как минимум, пакетов вида:

```
$ uname -r
2.6.35.14-95.fc14.x86_64
$ yum list all kernel-*
...
kernel-devel.x86_64                     2.6.35.14-95.fc14                              @updates
kernel-headers.x86_64                   2.6.35.14-95.fc14                              @updates
...
```

Здесь же показано **точное** требуемое соответствие версий пакетов той версии ядра, в которой вы ведёте разработку и компиляцию модулей. Если пакета нет, устанавливаем его (показано на примере одного из пакетов):

```
# yum install kernel-devel.x86_64
...
Установка:
 kernel-devel         x86_64         2.6.35.13-95.fc14         updates         6.6 M
...
Объем загрузки: 6.6 M
Будет установлено: 24 M
...
Установлено:
  kernel-devel.x86_64 0:2.6.35.13-95.fc14
```

Здесь показана установка в 64-разрядной системе, в 32-разрядной, естественно, это будет `kernel-devel.i686`, точно определять требуемые пакеты вы можете, пользуясь шаблонами имён в командах `yum`.

В любом случае, мы должны убедиться, что вся минимально необходимая файловая иерархия, соответствующая версии исполняющейся системы, у нас установлены:

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

Но и новая сборка ядра Linux, с чего мы и начали обсуждение этой главы, может оказаться полезной и нужной в некоторых случаях: для сборки ядра с некоторыми специальными качествами, например, с повышенными отладочными уровнями. Для сложных комплексных и долгосрочных проектов сборка рабочей версии ядра может оказаться желательной. Все новые специальные качества ядра, о которых упоминается выше, определяются исключительно **конфигурационными параметрами ядра**, определяемыми пользователем в ходе диалога конфигурирования ядра, предшествующего сборке. Установленные конфигурационные параметры ядра доступны позже в коде разрабатываемых модулей, в виде символьных констант вида `CONFIG_*`. О том, как использовать значения конфигурационных параметров ядра в коде модулей, рассказывалось ранее.

Сборка (и установка) нового ядра в новых версиях Linux может быть сопряжена с некоторыми сложностями, связанными не с самой сборкой (сборка ядра в более ранних версиях производилась вообще без проблем), а с некоторыми сопутствующими обстоятельствами взаимодействия ядра с другими частями загружаемой системы, из которых можно назвать: необходимость начального загрузочного образа, установка системы в виртуальную файловую систему...

Если же вы решите пересобрать ядро, то первое, что нужно сделать — выяснить: какое и откуда грузится ваше текущее ядро (все последующие примеры — с реального компьютера!):

```
$ uname -r
2.6.18-92.el5
$ sudo cat /boot/grub/grub.conf
...
title CentOS (2.6.18-92.el5)
    root (hd1,5)
    kernel /boot/vmlinuz-2.6.18-92.el5 ro root=LABEL=/ rhgb quiet
    initrd /boot/initrd-2.6.18-92.el5.img
...
```

Здесь нужно соблюдать величайшую осторожность:

```
$ ls /dev/hd*
/dev/hda   /dev/hde   /dev/hde1  /dev/hde2  /dev/hde5  /dev/hdf
/dev/hdf1  /dev/hdf2  /dev/hdf4  /dev/hdf5  /dev/hdf6
$ ls -l /dev/cdrom
lrwxrwxrwx 1 root root 3 Map 12 10:15 /dev/cdrom -> hda
$ sudo /sbin/fdisk /dev/hdf
...
Команда (m для справки): p
Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/hdf1   *           1         501     4024251   4f  QNX4.x 3-я часть
/dev/hdf2            1394        2438     8393962+   f  W95 расшир. (LBA)
/dev/hdf4             502        1393     7164990    c  W95 FAT32 (LBA)
/dev/hdf5            1394        1456      506016   82  Linux своп / Solaris
/dev/hdf6            1457        2438     7887883+  83  Linux
```

На данном компьютере (возможно, вопреки тому, что могло ожидаться на первый взгляд):

а). два HDD,

б). устройство `/dev/hda` — это CD-ROM,

в). 2-м HDD соответствуют `/dev/hde` и `/dev/hdf` (аппаратный EIDE контроллер ... но это не принципиально важно — диски при инсталляции могут быть «расставлены» самым замысловатым образом);

г). диску `(hd1,5)`, указанному как загрузочный в меню загрузчика `grub`, соответствует `/dev/hdf` (т. е. **2-й** диск) –`grub` «считает» диски, начиная с 0;

д). по той же причине, загрузочному разделу диска `(hd1,5)` соответствует `/dev/hdf6` (т. е. **6-й** раздел);

е). это верно только для старых версий загрузчиков `lilo` и `grub` :

```
$ sudo /sbin/grub
 GNU GRUB  version 0.97
grub> help
blocklist FILE                          boot
cat FILE                                chainloader [--force] FILE
clear                                   color NORMAL [HIGHLIGHT]
...
grub> quit
```

ж). Загрузчик `grub` версий 1.X (так называемый GRUB2), идущий на смену версиям 0.X (GRUB) - «ведёт счёт» начиная с 1!

Примечание: Выше специально показано, что `grub` имеет развитую интерактивную командную оболочку ... но это уже выходит за рамки нашего рассмотрения.

Если вы определились, откуда и какое у вас загружено ядро, и в том, что компилировать и собирать новое ядро вам **необходимо**, то вся вспомогательная информация для вас о сборке ядра размещена отдельным приложением в конце текста.

Обсуждение

1. В этой части обсуждения мы проделали рассмотрение, хоть и ограниченное в своём объёме, минимального набора инструментальных средств для создания и отладки модулей ядра. Это именно тот «необходимый и достаточный» набор инструментария, который позволяет вам выполнять такую работу. Есть ещё достаточно много инструментов, расширяющих ваши возможности в этой области, или существенно повышающие производительность работы. Но это всё **дополнительные** инструменты, и вы соберёте оптимальный для себя набор инструментов, экспериментируя с ними.

2. Представляющие интерес системные файлы перечислены нами на примере дистрибутивов группы RedHat / Fedora / CentOS (дистрибутивы RPM группы) - проверьте то же самое на примерах Debian / Ubuntu ... или других доступных дистрибутивах. Это очень полезное упражнение, такой сравнительный анализ, и он приведёт к заключению, что ничего принципиально отличающегося тип используемого дистрибутива Linux не привносит.

3. Поупражняйтесь в работе с потоковым редактором `sed`, языком программирования `awk (gawk)`, или другими подобными средствами работы с текстовыми образцами (`perl`, `grep`, `find`, ...), широко применяющимися в системах UNIX — они в высшей степени полезны при работе с конфигурационными файлами системы, сценариями сборки `make`, и вообще при отработке модулей ядра.

4. Хотя в тексте и сказано, что собирать ядро есть реальная необходимость не так и часто, но когда то, рано или поздно, вам его придётся пересобирать. В порядке упражнения - пересоберите образ вашей используемой системы.

Внешние интерфейсы модуля

«Частота использования goto для ядра в целом составляет один goto на 260 строк, что представляет собой довольно большое значение»

Скотт Максвел «Ядро Linux в комментариях»

Под внешними интерфейсами модуля мы будем понимать, как уже указывалось, те связи, которые может и должен установить модуль с «внешним пространством» Linux, видимым пользователю, с которыми пользователь может взаимодействовать либо из своего программного кода, или посредством консольных команд системы. Такими интерфейсами-связями есть, например, имена в файловых системах (в `/dev`, `/proc`, `/sys`), сетевые интерфейсы, сетевые протоколы... Понятно, что регистрация таких механизмов взаимодействия со стороны модуля, это не есть программирование в смысле алгоритмов и структур данных, а есть строго формализованное (регламентированное как по номенклатуре, так и по порядку вызова) использование предоставляемых для этих целей API ядра. Это занятие скучное, но это та первейшая фаза проектирования всякого модуля (драйвера): создание тех связей, через которые с ним можно взаимодействовать. Этим мы и станем заниматься на протяжении всего этого раздела.

Драйверы: интерфейс устройства

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):

- Каким способом драйвер будет регистрироваться в системе, как станет известно системе, что у неё появился в распоряжении новый драйвер?
- Каким образом драйвер создаёт (или использует) имя соответствующего ему устройства в каталоге `/dev`, и как он (драйвер) увязывается с старшим и младшим номерами этого устройства?
- После того, как драйвер увязан с устройством, какие будут использованы особенности в реализации основных операций 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 мдулем `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`.

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 и не выполнять вообще никаких 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 такой не обработчик по умолчанию будет всегда возвращать код ошибки (`mmap`), 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 открытии устройства. Вытекающие отсюда эффекты мы увидим в примерах в дальнейшем.

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 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 (misccellaneous) drivers.

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.

Примеры реализации

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 находит 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 связывав 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`).

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 возможностей системы `sysfs`). Ниже показаны только принципиальные отличия (дополнения) относительно предыдущего варианта:

dyndev.c :

```
#include <linux/device.h>
...
static struct cdev hcdev;
static struct class *devclass;

static int __init dev_init( void ) {
   int ret, i;
   dev_t dev;
   ...
   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" );
#define DEVNAME "dyn"
   for( i = 0; i < DEVICE_COUNT; i++ ) {
      char name[ 10 ];
      dev = MKDEV( major, DEVICE_FIRST + i );
      sprintf( name, "%s_%d", DEVNAME, i );
      device_create( devclass, NULL, dev, "%s", name );
   }
   printk( KERN_INFO "======== module installed %d:[%d-%d] ==========\n",
           MAJOR( dev ), DEVICE_FIRST, MINOR( dev ) );
err:
   ...
}

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 );
   ...
}
```

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

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.

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 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 такие устройства регистрируются в `sysfs` в едином 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 операция, которые часто упоминают как операции ввода-вывода в основном потоке данных, в Linux очень широко используется операция `ioctl()`, применяемая для управления устройством, осуществляющая обмен с устройством все 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 создания устройства:

ioctl_dev.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 );
}
```

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 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*`.

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 во всех рассматриваемых примерах);

2. Драйвер допускает только **единственное** открытие устройства; попытки параллельного открытия будут завершаться ошибкой до тех пор, пока использующий его процесс не закроет устройство.

3. Драйвер допускает много **параллельных** сессий использования устройства. При этом драйвер должен реализовать индивидуальный экземпляр данных для каждой копии открытого устройства.

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.

Неблокирующий ввод-вывод и мультиплексирование

Здесь я имею в виду реализацию в модуле (драйвере) поддержки 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.

Примечание: Наилучшую (наилучшую из известных мне) классификаций типов операций ввода-вывода дал У. Р. Стивенс [19], 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 [20], на которое мы будем, без излишних объяснений, опираться в своих примерах.

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
```

- 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.

Блочные устройства

Блочные устройства во многом наследуют технику символьных устройств, детально рассматриваемых на примерах ранее, но должны отрабатывать и дополнительные возможности API, связанные с произвольным (не последовательным) доступом. Регистраций таких устройств производится отдельным API:

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

Но главное отличие, от рассмотренного выше, состоит в использовании в качестве таблицы функций , реализующих операции, вместо `struct file_operations` используется `struct block_device_operations` (ищите её в `<linux/blkdev.h>`):

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

Но смысл и логика основных шагов при разработке драйвера блочного устройства остаётся той же. Разработка модулей поддержки блочных устройства является крайне редкой необходимостью для сторонних разработчиков (не самих производителей нового устройства прямого доступа), поэтому детально здесь не рассматривается, тем более, что она как ничто другое хорошо описана в литературе.

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()`). 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 ограниченном значении, часто соответствующем одной элементарной переменной языка С: `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 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`

По аналогии с тем, как все операции в `/dev` обеспечивались через таблицу файловых операций (`struct file_operations`), операции над именами в `/proc` увязываются (`<linux/proc_fs>`) 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 );
```

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

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/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` выполняет пара вызовов

(<linux/proc_fs.h>):

```
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 );
```

Эти операции, что будет показано далее, оперируют в равной мере как с новыми **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", 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:

```
$ make
...
$ sudo insmod mod_procr.ko
$ dmesg | tail -n50 | grep -v ^audit
! /proc/mod_node installed
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
$ dmesg | tail -n20 | grep -v ^audit
! /proc/mod_node installed
! read: 3072 (buffer=f1f7e000, off=0)
! return bytes: 61
! read: 3072 (buffer=f1f7e000, off=61)
! return bytes: 61
! read: 3072 (buffer=f1f7e000, off=61)
! return bytes: 61
```

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 -n50 | grep -v ^audit
! /proc/mod_node installed
! read: 10 (buffer=f32b1000, off=0)
! return bytes: 61
! read: 10 (buffer=f32b1000, off=10)
! return bytes: 61
! read: 10 (buffer=f32b1000, off=20)
! return bytes: 61
! read: 10 (buffer=f32b1000, off=30)
! return bytes: 61
! read: 10 (buffer=f32b1000, off=40)
! return bytes: 61
! read: 10 (buffer=f32b1000, off=50)
! return bytes: 61
! read: 10 (buffer=f32b1000, off=60)
! return bytes: 61
! read: 9 (buffer=f32b1000, off=61)
! return bytes: 61
! read: 10 (buffer=f32b1000, off=61)
! return bytes: 61
$ ./mcat 70
read + 61 bytes, input buffer: .........1.........2.........3.........4.........5.........6
read + 00 bytes, input buffer: .........1.........2.........3.........4.........5.........6
$ dmesg | tail -n6 | grep -v ^audit
! read: 70 (buffer=f1dfd000, off=0)
! return bytes: 61
! read: 9 (buffer=f1dfd000, off=61)
! return bytes: 61
! read: 70 (buffer=f1dfd000, off=61)
! return bytes: 61
```

Завершение работы модуля:

```
$ sudo rmmod mod_procr
$ ls -l /proc/mod_*
ls: невозможно получить доступ к /proc/mod_*: Нет такого файла или каталога
```

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
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
```

И можем повторить всё множество испытаний, показанных ранее для `mod_procr.ko`.

Варианты реализации чтения[12]

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
```

12 Далее идёт детализация, поэтому этот раздел можно проопустить без ущерба для остального содержания.

```
 * 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
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

Те же операции (чтение, запись) можно осуществлять с именами в `/proc`, используя традиционный механизм работы со всеми именами файловой системы. 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;
   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->proc_fops = &node_fops;
   LOG( "/proc/%s installed", NAME_NODE );
   return 0;
err_node:
   return ret;
}
```

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: %d bytes (ppos=%lld)", 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 %d bytes", 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, len = count < LEN_MSG ? count : LEN_MSG;
   LOG( "write: %d bytes", count );
   res = copy_from_user( buf_msg, (void*)buf, len );
   buf_msg[ len ] = '\0';
   LOG( "put %d bytes", len );
   return len;
}
```

В отличие от упрощенных реализаций, с которыми мы работали с `/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) с данными пространства 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;
   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", 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\n", NAME_NODE );
      goto err_node;
   }
   own_proc_node->uid = own_proc_node->gid = 0;
   own_proc_node->proc_fops = &node_fops;
   LOG( "/proc/%s installed", 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 );
}
```

Примечание: Здесь любопытно обратить внимание на то, с какой лёгкостью имя в `/proc` создаётся то как каталог, то как терминальное имя (файл), в зависимости от выбора единственного бита в флагах создания: `S_IFDIR` или `S_IFREG`.

Эксперименты с `mod_proct.ko` аналогичны предыдущему, с отличающимся путевым именем, которое создаёт модуль в `/proc`:

```
$ sudo insmod ./mod_proct.ko
$ cat /proc/modules | grep mod_
mod_proct 1454 0 - Live 0xf8722000
$ ls -l /proc/mod*
-r--r--r-- 1 root root 0 Июл  2 23:24 /proc/modules
/proc/mod_dir:
итого 0
-rw-rw-rw- 1 root root 0 Июл  2 23:24 mod_node
$ tree /proc/mod_dir
/proc/mod_dir
└── mod_node
0 directories, 1 file
```

Таким образом, показано, что создание структур имен 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 две альтернативных набора функций для реализации ввода вывода. 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
! 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
```

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 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:

```
struct kobject {
```

```
...
   struct kobj_type *ktype;
   struct dentry    *dentry;
};
```

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 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 */;
}
```

Показанная там же структура таблицы операций (`struct sysfs_ops`) содержит два поля — определения функций `show()` и `store()`, соответственно, чтения и записи символьного поля данных ядра, отображаемых этим файлом (и сами функции и их прототипы показаны в примере ниже).

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 аналогичностях 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, 0666, &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 вызываться 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-rw-rw- 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 это новое содержимое > /sys/class/x-class/xxx
$ cat /sys/class/x-class/xxx
это новое содержимое
$ dmesg | tail -n10 | grep -v ^audit
write 39
read 39
$ sudo rmmod xxx
```

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/data1
не инициализировано data1
$ cat /sys/class/x-class/data2
не инициализировано data2
$ cat /sys/class/x-class/data3
```

```
не инициализировано data3
$ echo строка 1 > /sys/class/x-class/data1
$ echo строка 2 > /sys/class/x-class/data2
$ echo строка 3 > /sys/class/x-class/data3
$ cat /sys/class/x-class/data1
строка 1
$ cat /sys/class/x-class/data2
строка 2
$ cat /sys/class/x-class/data3
строка 3
```

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 его автора, 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.

Сеть

Сетевая подсистема является гораздо разветвлённее итерфейса устройств Linux. Но, несмотря на обилие возможностей (например, если судить по числу обслуживающих сетевых утилит: `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. 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. 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 [25]:

- всё, что относится к поддержке оборудования и канальному уровню — описывается как сетевые **интерфейсы**, и обозначается как L2, преимущественно это Ethernet;
- 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 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, но именно там создаются и потребляются сетевые пакеты.

= = = = = = = = = =

здесь Рис. 5: сетевые уровни и уровни стека протоколов.

==========

Инструменты наблюдения

В отличие от всех прочих **устройств** в системе, 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). Одному аппаратному сетевому устройству может соответствовать одновременно несколько различных сетевых интерфейсов.

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. Самым известным из инструментов для этих целей 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 (CiscoSystemsVPNClient) от Cisco Systems (`cipsec0`), работающий через один и тот же физический канал (что подтверждает сказанное выше о возможности нескольких сетевых интерфейсов над одним 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 к тому же удалённому серверу-хосту, то мы получим совершенно другой (и даже ещё один дополнительно, параллельно) сетевой интерфейс:

```
$ 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)
```

Гораздо менее известным, но более развитым инструментом, является 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, 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.

Для анализа трафика разрабатываемого сетевого интерфейса вам безусловно потребуются что-то из числа известных утилит, таких как `tcpdump` (http://www.tcpdump.org/), или её GUI эквивалент `Wireshark` (http://www.wireshark.org/). Вот как для одного из сетевых интерфейсов

```
$ 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
```

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).

Структуры данных

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 рассмотрены, при обсуждении прохождения пакетов сквозь стек сетевых протоколов. А пока наша задача — создание той концевой точки (интерфейса), где эти последовательности действий начинаются и завершаются.

Ниже показан пример простого создания и регистрации в системе нового сетевого интерфейса (примеры этого раздела заимствованы из [6] 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;
```

```
    printk( KERN_INFO "my_setup(%s)\n", dev->name );
    /* 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.

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`, но реально разрабатываемые модули реализуют только некоторую малую часть из них.

Теперь созданное нами ранее (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
$ 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 устанавливается в качестве MAC (аппаратного) адреса созданного интерфейса.

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, например (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;
...
   /* Interface address info. */
   unsigned char  perm_addr[ MAX_ADDR_LEN ]; /* permanent hw address    */
   unsigned char  addr_len;                  /* hardware address length */
...
}
```

- где 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               */
```

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, не только сетевой подсистемы. Указатель такой приватной структуры помещается в структуру сетевого интерфейса, доступ к нему (а значит и к приватной структуре) 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 );
}
```

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 связный список. Следующий пример визуализирует такой список:

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 `lab1_network.ko`):

```
$ sudo insmod lab1_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
```

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[13]). 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 опеределённого 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):

13 Естественно, до какого времени он будет «новым» неизвестно — до появления ещё более нового.

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,
                                   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, упоминавшейся выше.

Часто задаваемый вопрос: а где же в этом процессе место (код), где реально создаётся информация, помещаемая в буфер, или где потребляется информация из принимаемых 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/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" );
   child = alloc_netdev( sizeof( struct priv ), ifstr, setup );
   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 обработчика входящих пакетов вызовом `netdev_rx_handler_unregister()`, который появился в API ядра начиная с 2.6.36 (ранее это приходилось делать более изощрёнными способами).

Работа с таким интерфейсом выглядит следующим образом:

- на 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 и конфигурируем его (на подсеть, отличную от `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 такой подсети на другом хосте LAN, это создать алиасный IP для сетевого интерфейса этого 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 из 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.
$
```

Протокол сетевого уровня

На этом уровне 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. В нашем случае в функцию отбираются (попадают) те буфера, которые удовлетворяют критерием, заложенным в структуре `struct packet_type`.

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>

int test_pack_rcv( struct sk_buff *skb, struct net_device *dev,
                   struct packet_type *pt, struct net_device *odev ) {
   printk( KERN_INFO "packet received with length: %u\n", skb->len );
   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 );
   printk( KERN_INFO "module loaded\n" );
   return 0;
}
```

```
static void __exit my_exit( void ) {
   dev_remove_pack( &test_proto );
   printk( 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, когда операционная система 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));
```

Протокол транспортного уровня

На этом уровне обеспечивается обработка таких протоколов, как: UDP, TCP, SCTP... 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 );
```

- где 2-й параметр вызова функций как раз и есть константа из числа `IPPROTO_*`.

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. До тех пор, пока в таблице роутинга, демонстрируемой 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
```

— 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`.

Внутренние механизмы 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, и **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 **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[14].

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 лошических адресов составляет 892Mb.

7. Различия в отображении единых логических адресов пространства пользователя (отличающихся процессов) в различные физические адресные пространства происходит не за счёт различий сегментных регистров, а за счёт различий на уровне страничного отображения памяти.

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[15]) 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).

14 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.

15 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).

Динамическое выделение участка памяти размером `size` байт производится вызовом:

```
#include <linux/slab.h>
void *kmalloc( size_t size, int flags );
```

Выделенная таким вызовом область памяти является **непрерывной в физической** памяти.

Впервые встреченный нами параметр `flags` очень часто фигурирует в коде ядра (в отличие, например, 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 чем `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 возвратиться 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, в виду объёмности сам тест (архив `mtest.tgz`) и результаты снесены в отдельно приложение, а здесь приведём только сводную таблицу:

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[16]

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),

16 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.

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,
                        void (*ctor)( void*, kmem_cache_t*, unsigned long flags ) );
```

Наконец, в 2.6.32, 2.6.35 и 2.6.35 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 );
```

А вот как выглядит выполнение этого размещения (картина весьма поучительная, поэтому остановимся на ней подробнее):

```
$ sudo insmod ./slab.ko
$ dmesg | tail -n300 | grep -v audit
allocate 31 objects into slab: my_cache
object size 7 bytes, full size 217 bytes
constructor called 257 times
$ cat /proc/slabinfo | grep my_
# name   <active_objs> <num_objs> <objsize> ...
my_cache     256    256     16  256    1 : tunables    0    0    0 : slabdata      1      1      0
$ sudo rmmod slab
```

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_
# name   <active_objs> <num_objs> <objsize> ...
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.

А вот 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[17].

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 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` 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
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

17 Что напомнило высказывание классика отечественного юмора М. Жванецкого: «А вы не пробовали слабительное со снотворным? Удивительный получается эффект!».

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 );
```

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 )
```

- 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.

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. Альтернативы (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 */
};
```

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 );
```

- добавляет новую запись `new` сразу же после головы списка, 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 );
```

- добавляет элемент `new` перед головой данного списка - 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 );
```

- данная запись удаляется из списка. 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 );
```

- данная запись удаляется из своего текущего положения и перемещается (запись) в начало голову 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 вставкой нового списка `list` сразу после головы `head`.

Структуры `list_head` хороши для реализации линейных списков, но использующие его программы часто больше заинтересованы в некоторых более крупных структурах, которые увязываются в список как целое. Предусмотрен макрос `list_entry`, который связывает указатель структуры `list_head` обратно с указателем на структуру, которая его содержит. Он вызывается следующим образом:

```
list_entry( struct list_head *ptr, type_of_struct, field_name );
```

- где `ptr` является указателем на используемую структуру `list_head`, `type_of_struct` является типом структуры, содержащей этот `ptr`, и `field_name` является именем поля списка в этой структуре.

Пример: в нашей ранее показанной структуре `todo_struct` поле списка называется просто `list`. Таким образом, мы бы хотели превратить запись в списке `listptr` в соответствующую структуру, то могли бы выразить это такой строкой:

```
struct todo_struct *todo_ptr = list_entry( listptr, struct todo_struct, list );
```

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 сохранить список объектов `todo_struct`, отсортированный в порядке убывания. 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 создание циклов, которые перебирают списки. 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:

```
list_for_each( struct list_head *cursor, struct list_head *list )
```

- 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` 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 )
```

- эти **макросы** облегчают процесс просмотр списка, содержащего структуры данного типа `type`. Здесь `cursor` является указателем на содержащий структуру тип, и `member` является именем структуры `list_head` внутри содержащей структуры. С этими макросами нет необходимости помещать внутри цикла вызов макроса `list_entry()`.

В заголовках `<linux/list.h>` определены ещё некоторые дополнительные декларации для описания динамических структур.

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

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. 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.

Время: измерение и задержки

«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 `#defined`) 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
```

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>`. 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 )
```

- где - `unknown` - это обычно значение переменной `jiffies`, а параметр `known` - значение, с которым его необходимо сравнить. Макросы возвращают значение `true`, если выполняются соотношения момент ов 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 размерности 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:

- «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"

unsigned long long rdtsc( void ) {
   unsigned long long int x;
   asm volatile ( "rdtsc" : "=A" (x) );
   return x;
}
```

- 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 ) {
   uint32_t n, m, sum = 0;
   n = m = ( rep >= 0 ? NUMB : rep );
   while( n-- ) {
      uint64_t cf, cs;
      cf = rdtsc();
      cs = rdtsc();
      sum += (uint32_t)( cs - cf );
   }
   return sum / m;
}
```

- измерение частоты процессора (число процессорных тактов за секундный интервал):

proc_hz.c :

```
#include "libdiag.h"

unsigned long proc_hz( void ) {
   time_t t1, t2;
   uint64_t cf, cs;
   time( &t1 );
   while( t1 == time( &t2 ) ) cf  = rdtsc();
   while( t2 == time( &t1 ) ) cs  = rdtsc();
   return (unsigned long)( cs - cf - calibr( 1000 ) );
}
```

- перевод потока в реал-тайм режим диспетчирования (в частности, на FIFO дисциплину), что бывает очень важно 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
```

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;  /* наносекунды */
}
...
struct timeval {
   time_t       tv_sec;  /* секунды */
   suseconds_t  tv_usec; /* микросекунды */
};
...
```

```
#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 );
```

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
```

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 диспетчирование в ядре

Диспетчеризация 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 приоритетом `nice` на каждом периоде диспетчирования, в зависимости от приоритета процесса, назначается период активности (timeslice) — 10-200 системных тиков, который динамически в ходе выполнения этого проценсса может быть ещё расширен в пределах 5-800, в зависимости от характера интерактивности процесса. На этом построена схема диспетчирования процессов в Linux сложности O(1) - не зависящая по производительности от числа диспетчируемых процессов, которой очень гордятся разработчики ядра Linux (возможно, вполне оправдано). Но всё это уже далеко выходит за пределы нашего рассмотрения... Нам же здесь важно зафиксировать, что все диспетчируемые 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).

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( 1, &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:

`int spin_try_lock( spinlock_t *sl );` - попытка захвата без блокирования, если блокировка уже захвачена, функция возвратит ненулевое значение

`int spin_is_locked( spinlock_t *sl );` - возвращает ненулевое значение, если блокировка в данный момент захвачена

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();
```

– 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( j1; ... )
   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 операторы до вызова и после вызова :

```
void barrier( void );
```

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!

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.

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) 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 архитектурно зависима (по взаимодействию с контроллером прерываний), даже с более современным чипом APIC контроллера процессора x86 схема взаимодействия в деталях будет выглядеть по-другому. Это недопустимо для много-платформенной операционной системы, которой является Linux. Поэтому вводится логическая модель обработки прерываний, в которой аппаратно зависимые элементы взаимодействия берёт на себя ядро, а обработка прерывания разделяется на две последовательные фазы:

- Регистрируется функция обработчика «верхней половины», который выполняется **при запрещённых прерываниях** локального процессора. Именно этой функции передаётся управление при возникновении аппаратного прерывания. Функция возвращает управление ядру системы традиционным `return`.
- Перед своим завершением функция-обработчик активирует последующее выполнение «нижней половины», 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 и возвращает управление из прерывания уже именно командой `iret`...
- А вот запланированная выше к выполнению функция нижней половины будет вызвана ядром в некоторый момент позже (но часто это может быть и непосредственно после завершения `return` из верхней половины), 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**.

= = = = = = = = = =

здесь Рис. : модель обработки аппаратных прерываний

= = = = = = = = = =

Регистрация обработчика прерывания

Функции и определения, реализующие интерфейс регистрации прерывания, объявлены в `<linux/interrupt.h>`. Первое, что мы должны всегда сделать — это зарегистрировать функцию обработчик 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 );
```

Но позже оказалось целесообразным и удобным использовать именно в качестве `*dev` — указатель на специфическую для устройства структуру, которая и содержит все характерные данные экземпляра: поскольку для каждого экземпляра создаётся своя копия структуры, то указатели на них и будут уникальны, что и требовалось. 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 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
...
```

Отображение 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 параметром `name` в вызове `request_irq()`). Ниже показана «раскладка» прерываний в архитектуре x86, здесь источник прерываний — стандартный программируемый контроллер прерываний PC 8259 (XT-PIC):

```
$ 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
...
```

Примечание: 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). Эта «классика» начала постепенно вытесняться только в последние 5-10 лет, в связи с широким наступлением SMP архитектур, и применением для них нового контроллера: APIC. Одним из первых ставших стандартным образцом стал чип 82489DX, но на сегодня функции APIC просто вшиты в чипсет системной платы. Архитектура APIC позволяет обслуживать число линий IRQ больше 16-ти, что было пределом на протяжении многих лет.

Те линии IRQ, для которых не установлены текущие обработчики прерываний, не отображаются в `/proc/interrupts`. Вот то же самое, но на существенно более новом компьютере с 2-мя процессорами (ядрами), когда источником прерываний является усовершенствованный контроллер прерываний IO-APIC (отслеживаются прерывания по фронту и по уровню: `IO-APIC-edge` или `IO-APIC-level`):

```
$ cat /proc/interrupts
           CPU0       CPU1
   0:   47965733          0   IO-APIC-edge      timer
   1:         10          0   IO-APIC-edge      i8042
   4:          2          0   IO-APIC-edge
   7:          0          0   IO-APIC-edge      parport0
   8:          1          0   IO-APIC-edge      rtc0
   9:      24361          0   IO-APIC-fasteoi   acpi
  12:        157        743   IO-APIC-edge      i8042
  14:     700527          0   IO-APIC-edge      ata_piix
  15:     525957          0   IO-APIC-edge      ata_piix
  16:    1146924          0   IO-APIC-fasteoi   i915, eth0
  18:         78     441659   IO-APIC-fasteoi   uhci_hcd:usb4, yenta
  19:          3        777   IO-APIC-fasteoi   uhci_hcd:usb5, firewire_ohci, tifm_7xx1
  20:    2087614          0   IO-APIC-fasteoi   ehci_hcd:usb1, uhci_hcd:usb2
  21:        190      11976   IO-APIC-fasteoi   uhci_hcd:usb3, HDA Intel
  22:          0          0   IO-APIC-fasteoi   mmc0
  27:          0          0   PCI-MSI-edge      iwl3945
 NMI:          0          0   Non-maskable interrupts
...
```

Ещё одним источником (динамической) информации о произошедших (обработанных) прерываниях является файл `/proc/stat`:

```
$ cat /proc/stat
cpu  2949061 32182 592004 6337626 301037 8087 4521 0 0
cpu0 1403528 14804 320895 3068116 167380 6043 4235 0 0
cpu1 1545532 17377 271108 3269510 133657 2043 285 0 0
intr 139510185 47968356 10 0 0 2 0 0 0 1 24361 0 0 900 0 700531 525967 1147282 0 441737 780
2087674 12166 0 0 0 0 0 0 ...
```

Здесь строка, начинающаяся с `intr` содержит суммарные по всем процессорам значения обработанных прерываний для всех последовательно линий IRQ.

Теперь, умея хотя бы наблюдать происходящие в системе прерывания, мы готовы перейти к управлению ними.

Обработчик прерываний, верхняя половина

Прототип функции обработчика прерывания уже показывался выше:

```
typedef irqreturn_t (*irq_handler_t)( int irq, void *dev );
```

где :

- `irq` — линия IRQ;

- `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 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.

Управление линиями прерывания

Под управлением линиями прерываний, в этом месте описаний, мы будем понимать запрет-разрешение прерываний поодной 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

Отложенная обработка прерывания предполагает, что некоторая часть действий по обработке результатов прерывания может быть отложена на более позднее выполнение, когда система будет менее загружена. Главная достигаемая здесь цель состоит в том, что отложенную обработку можно производить не в самой функции обработчика прерывания, и к этому моменту времени может быть уже восстановлено разрешение прерываний по обслуживаемой линии (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**. 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` нам недостаточно хэдеров, и необходимо опуститься в рассмотрение исходных кодов реализации ядра (файл `<kernel/softirq.c>`, если у вас не установлены исходные тексты ядра, что совершенно не есть необходимостью для всего прочего нашего рассмотрения, то здесь вы будете вынуждены это сделать, если хотите повторить наш экскурс):

```
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 статичнее:

```
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
      HI:          0          0
   TIMER:   16940626   16792628
  NET_TX:       4936          1
  NET_RX:      96741       1032
   BLOCK:     176178          2
BLOCK_IOPOLL:      0          0
 TASKLET:        570      50738
   SCHED:     835250    1191280
 HRTIMER:       6286       5457
     RCU:   17000398   16867989
```

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 перекомпиляцию ядра и создание нового уровня `softirq`, то для этого необходимо:

- Определить новый индекс (уровень) отложенного прерывания, вписав (файл `<linux/interrupt.h>`) его константу `XXX_SOFT_IRQ` в перечисление, где-то, очевидно, на одну позицию выше `TASKLET_SOFTIRQ` (иначе зачем переопределять новый уровень и не использовать тасклет?).

- Во время инициализации модуля должен быть зарегистрирован (объявлен) обработчик отложенного прерывания с помощью вызова `open_softirq()`, который принимает три параметра: 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 );
}
```

- Функция-обработчик отложенного прерывания (в точности как и рассматриваемого ниже тасклета) должна в точности соответствовать правильному прототипу:

```
void xxx_analyze( unsigned long data );
```

- 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 (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;
}
```

- Затем в подходящий (для системы) момент времени отложенное прерывание выполняется. Обработчик отложенного прерывания выполняется при разрешенных 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 блокироваться.

Главная причина использования отложенных прерываний — **масштабируемость на многие процессоры**. Если нет необходимости масштабирования на многие процессоры, то лучшим выбором будет механизм тасклетов.

Тасклеты

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.

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, для 2-х процессоров это будут `events/0` и `events/1` :

```
$ ps -ALf | head -n12
...
root          9     2     9  0    1 08:55 ?        00:00:00 [events/0]
root         10     2    10  0    1 08:55 ?        00:00:00 [events/1]
...
```

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 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 мне неоднократно задавали 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/jiffies.h>
#include <linux/interrupt.h>
#include <linux/timex.h>

MODULE_LICENSE("GPL");

cycles_t cycles1, cycles2;
static u32 j1, j2;

char tasklet_data[] = "tasklet_function was called";

/* Bottom Half Function */
void tasklet_function( unsigned long data ) {
    j2 = jiffies;
    cycles2 = get_cycles();
    printk( "%010lld [%05d] : %s\n", (long long unsigned)cycles2, j2, (char*)data );
    return;
}

DECLARE_TASKLET( my_tasklet, tasklet_function, (unsigned long)&tasklet_data );

int init_module( void ) {
    j1 = jiffies;
```

```
   cycles1 = get_cycles();
   printk( "%010lld [%05d] : tasklet_scheduled\n", (long long unsigned)cycles1, j1 );
   /* Schedule the Bottom Half */
   tasklet_schedule( &my_tasklet );
   return 0;
}

void cleanup_module( void ) {
  /* Stop the tasklet before we exit */
  tasklet_kill( &my_tasklet );
  return;
}
```

Вот как выглядит его исполнение:

```
$ uname -a
Linux notebook.localdomain 2.6.32.9-70.fc12.i686.PAE #1 SMP Wed Mar 3 04:57:21 UTC 2010 i686 i686
i386 GNU/Linux
$ sudo insmod mod_tasklet.ko
$ dmesg | tail -n100 | grep " : "
51300758164810 [30536898] : tasklet_scheduled
51300758185080 [30536898] : tasklet_function was called
$ sudo rmmod mod_tasklet
$ sudo nice -n19 ./clock
00002EE46EFE8248
00002EE46F54F4E8
00002EE46F552148
1663753694
```

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 20000 процессорных тактов частоты 1.66 Ghz (показан уже обсуждавшийся тест из раздела о службе времени, нас интересует только последняя строка его вывода), это составляет порядка 12 микросекунд.

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/jiffies.h>
#include <linux/interrupt.h>
#include <linux/timex.h>

MODULE_LICENSE("GPL");

static struct workqueue_struct *my_wq;

typedef struct {
  struct work_struct my_work;
  int    id;
  u32    j;
  cycles_t cycles;
} my_work_t;

/* Bottom Half Function */
static void my_wq_function( struct work_struct *work ) {
   u32 j = jiffies;
   cycles_t cycles = get_cycles();
   my_work_t *wrk = (my_work_t*)work;
```

```
   printk( "#%d : %010lld [%05d] => %010lld [%05d]\n",
           wrk->id,
           (long long unsigned)wrk->cycles, wrk->j,
           (long long unsigned)cycles, j
         );
  kfree( (void *)wrk );
  return;
}

int init_module( void ) {
   my_work_t *work1, *work2;
   int ret;
   my_wq = create_workqueue( "my_queue" );
   if( my_wq ) {
      /* Queue some work (item 1) */
      work1 = (my_work_t*)kmalloc( sizeof(my_work_t), GFP_KERNEL );
      if( work1 ) {
         INIT_WORK( (struct work_struct *)work1, my_wq_function );
         work1->id = 1;
         work1->j = jiffies;
         work1->cycles = get_cycles();
         ret = queue_work( my_wq, (struct work_struct *)work1 );
      }
      /* Queue some additional work (item 2) */
      work2 = (my_work_t*)kmalloc( sizeof(my_work_t), GFP_KERNEL );
      if( work2 ) {
         INIT_WORK( (struct work_struct *)work2, my_wq_function );
         work2->id = 2;
         work2->j = jiffies;
         work2->cycles = get_cycles();
         ret = queue_work( my_wq, (struct work_struct *)work2 );
      }
   }
   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 компьютере):

```
$ sudo insmod mod_workqueue.ko
$ lsmod | head -n3
Module                  Size  Used by
mod_workqueue           1079  0
vfat                    6740  1
$ ps -ef | grep my_
root     17058     2  0 22:43 ?        00:00:00 [my_queue/0]
root     17059     2  0 22:43 ?        00:00:00 [my_queue/1]
olej     17061 11385  0 22:43 pts/10   00:00:00 grep my_
```

- видим, как появился новый обрабатывающий поток ядра, с заданным нами именем, причём по одному экземпляру такого потока на каждый процессор системы.

```
$ dmesg | grep "=>"
#1 : 54741885665810 [32606771] => 54741890115000 [32606774]
#2 : 54741885675880 [32606771] => 54741890128690 [32606774]
```

```
$ sudo rmmod mod_workqueue
```

На этот раз мы помещаем в очередь отложенных действий два экземпляра работы, и каждый из них отстрочен на 3 системных тика от точки планирования — здесь латентность реакции существенно больше случая тасклетов, что и соответствует утверждениям в литературе.

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.

Обслуживание периферийных устройств

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 контакте с разработчиком «железа», в постоянных консультациях по каким портам ввода-вывода выполнять операции и с какой целью. Поэтому задачи непосредственно организации обмена данными не 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.

Анализ оборудования

Целью такого анализа, обычно производимого предварительно, перед написанием модуля драйвера, является уточнение специфических численных параметров тех или иных образцов оборудования. Общеизвестные команды для этих целей, это, например, `lspci` и `lsusb`, о которых мы будем вспоминать далее подробно и обстоятельно. 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.

```
notebook.localdomain
    description: Notebook
    product: HP Compaq nc6320 (ES527EA#ACB)
    vendor: Hewlett-Packard
    version: F.0E
    serial: CNU6250CFF
    width: 32 bits
```

```
    capabilities: smbios-2.4 dmi-2.4
...
```

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.

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[18], 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 );
```

- где `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)
00:1c.0 0604: 8086:27d0 (rev 01)
```

18 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.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 Sddress 0
0x14 — Base Sddress 1
0x18 — Base Sddress 2
0x1C — Base Sddress 3
0x20 — Base Sddress 4
0x24 — Base Sddress 5;
0x3C — IRQ Line
0x3D — IRQ Pin
```

Вся регистрация устройства PCI и связывание его параметров с кодом модуля происходит исключительно через значения, считанные из конфигурационного пространства устройства. Обработку конфигурационной информации (уже сформированной при установке PCI устройства) 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 цикла;
- конфигурационные параметры никогда не читаются напрямую, по их смещениям; для этого существуют макросы вида `pci_read_config_byte()` и `pci_read_config_word()`;
- конкретный вид считываемого конфигурационного параметра задаётся символьными константами вида `PCI_*` - 2-й параметр макроса;
- результат (значение конфигурационного параметра) возвращается в виде побочного эффекта в 3-й параметр макроса.

А теперь самое время рассмотреть небольшой начальный фрагмент результата выполнения написанного выше модуля:

```
$ 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 устройства 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 `pci_device_id` должна быть экспортирована в пользовательское пространство, чтобы позволить системам горячего подключения и загрузки модулей 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, указывет, 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.

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 );
```

- 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 );
```

- 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 передачи и объём передачи); процесс после этого блокируется;

- 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 получает входные данные, подтверждает прерывание и переводит процесс в активное состояние, процесс теперь имеет возможность читать данные.

Установленные в системе каналы обмена по DMA отображаются в файловую систему `/proc`:

```
$ cat /proc/dma
 2: floppy
 4: cascade
```

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, `dma_alloc_coherent()` допускает минимальное выделение в одну физическую страницу; в этом случае оптимальным становится `dma_pool()`.

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 является USB хост, а ведомым периферийное устройство. Контроллер хоста, в свою очередь, является одним из устройств на PCI шине, как это обсуждалось выше. USB терминология охватывает три версии стандарта — скорости функционирования: оригинальный стандарт 1.0 (называемый низко-скоростным) специфицирующий 1.5MBps, стандарт 1.1 (называемый полно-скоростным) специфицирующий 12MBps, стандарт 2.0 (называемый высоко-скоростным), поддерживающий до 480MBps, последний стандарт является на сегодня текущим. Также появляются уже устройства, работающие согласно ещё более высоко-скоростного 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[19].

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 Linux в значительной мере нивелирует различия спецификаций контроллера для уровня API, используемого при написании модулей поддержки устройств.

Схема идентификации устройства парой индексов VendorID — DeviceID, показанная для устройств PCI, оказалась настолько плодотворной, что подобный ей же вариант используется для устройств USB (пара цифр, выводимая после ID):

```
$ lsusb
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 002: ID 046d:c517 Logitech, Inc. LX710 Cordless Desktop Laser
Bus 004 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 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 008: ID 152d:2329 JMicron Technology Corp. / JMicron USA Technology Corp.
Bus 001 Device 007: ID 08ff:2580 AuthenTec, Inc. AES2501 Fingerprint Sensor
Bus 001 Device 006: ID 03f0:171d Hewlett-Packard Wireless (Bluetooth + WLAN) Interface [Integrated
Module]
Bus 001 Device 005: ID 046d:080f Logitech, Inc.
Bus 001 Device 002: ID 0424:2503 Standard Microsystems Corp. USB 2.0 Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Та же информация может быть показана в виде иерархии шин USB:

```
$ lsusb -t
Bus#  5
`-Dev#   1 Vendor 0x1d6b Product 0x0001
Bus#  4
`-Dev#   1 Vendor 0x1d6b Product 0x0001
  `-Dev#   2 Vendor 0x046d Product 0xc517
Bus#  3
`-Dev#   1 Vendor 0x1d6b Product 0x0001
Bus#  2
`-Dev#   1 Vendor 0x1d6b Product 0x0001
Bus#  1
`-Dev#   1 Vendor 0x1d6b Product 0x0002
  |-Dev#   2 Vendor 0x0424 Product 0x2503
  | |-Dev#   6 Vendor 0x03f0 Product 0x171d
  | `-Dev#   7 Vendor 0x08ff Product 0x2580
  |-Dev#   8 Vendor 0x152d Product 0x2329
  `-Dev#   5 Vendor 0x046d Product 0x080f
```

Каждое устройство USB характеризуется своей адресной идентификацией, которая (в отличие от PCI) имеет формат: шина : устройство. Устройство с номером 1 на каждой шине — это есть корневой разветвитель

19 Стандарт USB OTG, кроме этого, предусматривает 5-й контакт для идентификации по признаку хост-устройство.

USB для этой шины. До 127 устройств (считая с разветвителями) могут быть подключены к одной шине. Адресная идентификация может существенно меняться в зависимости от того, в какой разъём USB включается устройство.

С другой стороны, список идентификаторов USB (производитель:устройство) является константным значением, однозначно определяющим тип устройства (а значит и модуль ядра, который должен осуществлять поддержку этого устройства). Список идентификаторов USB 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 по URL: http://www.linux-usb.org/usb.ids (образец достаточно свежего такого файла помещён в соответствующий раздел примеров). Для одного из приведенных выше устройств (WEB-камеры), для которого мы будем проводить тест, запись в этом списке выглядит так (сравните с выводом команды `lsusb` выше):

```
#       List of USB ID's
# Date:    2011-04-14 20:34:04
046d  Logitech, Inc.
...
        080f  Webcam C120
...
```

Пример подключения (и отключения) и регистрацию USB-устройства показывает модуль (архив `usb.tgz`) `lab1_usb.ko` (заимствован из [6] при замене, естественно, в коде ID USB устройства на наблюдаемые выше, и достаточно существенных изменениях в 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:

```
$ 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
```

... 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. Стандарты USB предусматривают четыре режима обмена с устройством (я даю оригинальные названия без переводов):

- Control transfer: используется для передачи конфигурационной и управляющей информации;
- Bulk transfer: для передачи больших объёмов некритичной по времени информации;
- Interrupt transfer: для передачи малых порций критичной по времени информации;
- Isohronous transfer: для передачи реал-тайм потоков на фиксированных скоростях передачи;

Для каждой единицы адресации устройства (endpoint) должен быть установлен соответствующий режим обмена.

Собственно реализация обмена с USB устройством, после его связывания с модулем ядра, является существенно более зависимой от специфики самого устройства и протоколов его обмена. Это уже не есть предметом непосредственно общей функциональности модуля ядра, а поэтому и не может рассматриваться в общем виде. На этом мы завершим краткое введение в поддержку USB устройств...

Расширенные возможности

«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 востребовакны одинаково широко, некоторы используются заметно реже, чем типовые средства, разбираемые выше. 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 оснований[20] (так же, например, как и 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 );
```

20 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 собственника.

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 становитсят 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 );
```

То 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 немногих 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 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 потокам пространства ядра. Возможность таких взаимодействий должна быть понятна из простого соображения того, что большинство вызовов API ядра, которые рассматривались на протяжении всего протяжённого предыдущего рассмотрения, и 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`);
- диалоговый пользовательский процесс `ioctl`, который указывает модулю ядра `ioctl_signal.ko`: какой именно сигнал отсылать (по номеру) и какому процессу; процесс `ioctl` будет передавать команды для `ioctl_signal.ko` посредством вызовов `ioctl()`, что уже рассматривалось 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` - 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 процесс 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 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»):

sigreq.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. А теперь выполним весь комплекс: процесс `ioctl` последовательностью вызовов `ioctl()` заставляет загруженный модуль ядра 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 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` 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.

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.

В качестве иллюстрации выполнения таких операций позаимствуем пример из [6] (архив `user_io.tgz`):

lab1_ioports.c :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/io.h>
#include <stdlib.h>
#include <fcntl.h>
#define PARPORT_BASE 0x378

int do_ioperm( unsigned long addr, unsigned long nports ) {
   unsigned char zero = 0, readout = 0;
   if( ioperm( addr, nports, 1 ) )
      return EXIT_FAILURE;
   printf( "Writing: %6d  to  %lx\n", zero, addr );
   outb( zero, addr );
   usleep( 1000 );
   readout = inb( addr + 1 );
   printf( "Reading: %6d from %lx\n", readout, addr + 1 );
   if( ioperm( addr, nports, 0 ) )
      return EXIT_FAILURE;
   return EXIT_SUCCESS;
}

int do_read_devport( unsigned long addr, unsigned long nports ) {
   unsigned char zero = 0, readout = 0;
   int fd;
   if( ( fd = open( "/dev/port", O_RDWR ) ) < 0 )
      return EXIT_FAILURE;
   if( addr != lseek( fd, addr, SEEK_SET ) )
      return EXIT_FAILURE;
   printf( "Writing: %6d  to  %lx\n", zero, addr );
   write( fd, &zero, 1 );
   usleep(1000);
   read( fd, &readout, 1 );
```

```
   printf( "Reading: %6d from %lx\n", readout, addr + 1 );
   close( fd );
   return EXIT_SUCCESS;
}

int main( int argc, char *argv[] ) {
   unsigned long addr = PARPORT_BASE, nports = 2;
   if( argc > 1 )
      addr = strtoul( argv[ 1 ], NULL, 0 );
   if ( argc > 2 )
      nports = atoi( argv[ 2 ] );
   if( do_read_devport( addr, nports ) )
      fprintf( stderr, "reading /dev/port method failed\n" );
   if( do_ioperm( addr, nports ) )
      fprintf( stderr, "ioperm method failed\n" );
   return EXIT_SUCCESS;
}
```

Программа выполняет операции записи/чтения, с небольшой задержкой друг за другом, по двум портам с последовательными адресами, причём, делается это двумя способами: через операции `outb()/inb()` и через устройство отображения портов `/dev/port` :

```
$ ./lab1_ioports
reading /dev/port method failed
ioperm method failed
$ sudo ./lab1_ioports
Writing:      0  to  378
Reading:    120 from 379
Writing:      0  to  378
Reading:    120 from 379
$ cat /proc/ioports
...
00f0-00ff : fpu
...
$ sudo ./lab1_ioports 240
Writing:      0  to  f0
Reading:    255 from f1
Writing:      0  to  f0
Reading:    255 from f1
```

Из операций пространства пользователя, относящихся к вводу/выводу можно ещё отметить вызов:

```
int ioperm( unsigned long from, unsigned long num, int turn_on );
```

- устанавливает биты привилегий для доступа к области портов ввода/вывода, где:

- `from` - начальный порт области;

- `num` - число портов в области;

- `turn_on` — разрешить (1) или запретить (0) привилегированные операции.

Этот вызов, главным образом, для x86 архитектуры, на большинстве других он будет возвращать ошибку. 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()`. Сменить уровень привилегий (вызывающей задачи) на специфицируемый уровень:

```
#include <sys/io.h>
int iopl( int level );
```

Уровень привилегий обычного пользовательского процесса 0, уровень привилегий может быть задан от 0 до 3, иначе будет возвращена ошибка.

Естественно, что для получения привилегий процесс должен обладать правами root. После получения привилегий процесс может выполнять:

outb() / outw() / outl() - запись в указанный порт;

inb() / inw() / inl() - чтение из указанного порта;

Помимо возможности ввода/вывода, для таких пользовательских программ, как правило, нужно предотвратить выгрузку страниц 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;

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). 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 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 `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** 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
```

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 некоторые структуры ядра, и, к счастью, ядро предоставляет нам вызов `kallsyms_lookup_name()` (экспортирует его), позволяющий делать поиск в этой структуре . 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 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.

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;

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[21]. А сделаем куда более красивый трюк (архив `new_sys.tgz`): в качестве иллюстрации возможности и реализации технических приёмов, мы выполним пользовательский библиотечный[22] вызов `printf()` из кода модуля ядра! Выглядит это так:

mod_wrc.c :

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

21 Кроме того, вирусописателям тоже нужно оставить часть работы на самостоятельную проработку?

22 О которых мы говорили в начале книги, что из ядра нельзя вызывать библиотечные вызовы POSIX API. Из ядра можно всё! Но только ... осторожно.

```
#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
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 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 С.

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 [26]. 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`[23], 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
```

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
```

23 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.

```
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 x86, но и защиту записи мы наблюдаем в x86! На другой платформе будут найдены свои аналогичные решения. В архитектуре ia-32 за контроль записи отвечает 16-й бит в управляющем регистре процессора CR0 (называемый WP бит), в архитектуре ia-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, и несколько макросов, оперирующих с этим регистром. Макросы `rw_enable()` (разрешающий запись в сегмент для чтения) и `rw_disable()` (восстанавливающий контроль записи), реализованы 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 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 стандартами языка C), как определение вложенной функции `symb_fn()`, локальной по отношению к вызывающей (такое очень характерно, 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.

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. И 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;

– 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`;

В конечном счёте, почти все эти элементы встречались ранее, теперь только объединяем их вместе. А вот как выглядит протокол выполнения этого примера:

```
$ sudo insmod mod_wrchg.ko
:-) $ :-) e:-) c:-) h:-) o:-)  :-) c:-) т:-) p:-) o:-) к:-) a:-)
:-) 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. 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 (да и 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 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 своё 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 используемом обоими программами:

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 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

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"
#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 [ffffffda], 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 [fffffda], 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 длс рассмотрения в архиве `hidden.tgz`.

Динамическая загрузка

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, т 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 мне не удалось найти ни образцов корректного использования `init_module()` и `delete_module()`, ни даже их корректных прототипов вызова. Напротив, все источники полнятся просто синтаксически некорректными, устаревшими примерами. Поэтому восстанавливать примеры их использования пришлось именно обратным реинжинирингом через `syscall()`.

... 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 делаются выглядят так:

- Адреса символов ядра `"sys_init_module"` и `"sys_delete_module"` разыскиваются функцией `find_sym()` (мы подобное рассматривали раньше);
- 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);
- Выполняется косвенный вызов по указателям функций `sys_init_module` и `sys_delete_module`;
- 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, изображающее численное значение (в различных системах счисления: 8, 10, 16), и ожидает получить обратно вычисленное численное 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
```

Этот пример выводит так много 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».

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.

- 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 ситуация 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 свойста, состоит в том, что команда разработчиков ядра Linux крайне негативно относится вообще к идее интерактивных отладчиков для их ядра. 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; 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 динамично развивающихся проектов виртуальных машин QEMU (свободный проект http://wiki.qemu.org) и VirtualBox (также основанный на QEMU проект от Sun Microsystems, ныне от Oracle). 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;
- простота связи (загрузка модуля, наблюдение результатов) со средой разработки по внутренней TCP/IP виртуальной сети на основе тунельного интерфейса Linux;
- возможность использования отладчика `gdb` в базовой системе, для наблюдения «извне» за процессами, происходящими в виртуальной машине;
- 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).

Из названных двух близких VM: 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.

Хотелось бы специально остановиться ещё на одной, не совсем очевидной стороне использования виртуальных машин Linux в изготовлении модулей ядра... Напомню, 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 неоднократно показано в некоторых приводившихся ранее примерах).

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, 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. Но переписывать код модуля перед его сдачей — это тоже мало приемлемый вариант, так как такой 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 свя:диагностическим и управляющим переменным;
- 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 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 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 и `echo` для записи. 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 POSIX совместимости 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.

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. 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 A : сборка и установка ядра

В принципе, если вас интересует **только** обновление версии ядра вашей рабочей системы, то лучший способ сделать это — обновление ядра пакетной системой из репозитария того дистрибутива, который вы используете. Я это делаю, например, для дистрибутивов RedHat / Fedora / CentOS:

```
# yum list available kernel*
...
Доступные пакеты
kernel.i686                                 2.6.32.26-175.fc12                 updates
kernel-PAE.i686                             2.6.32.26-175.fc12                 updates
kernel-PAE-devel.i686                       2.6.32.26-175.fc12                 updates
...
```

... и далее с последующей установкой того, что вы хотите обновлять - и тогда вам не нужно и читать этот раздел. Мы же ниже будем рассматривать ту, реально достаточно редко обоснованную необходимость, когда мы собираем совершенно новое ядро из исходных кодов ядра Linux.

Я знаю, по крайней мере, весьма ограниченный перечень ситуаций (в отработке модулей ядра), когда ядро действительно **нужно** заново собирать:

1. Ядро можно собрать с расширенным (в разной мере: большей или меньшей) набором отладочных опций. В сложных случаях отладки модулей это необходимо. Но неудачный выбор набора отладочных опций может очень сильно снизить быстродействие ядра (собственно, по этой причине многие отладочные опции отключены по умолчания).

2. В коде модуля требуется использовать те вызовы API, которые появились только в более поздних версиях ядра. Пример тому, вызов сетевого API `netdev_rx_handler_register()` и родственные ему (большая группа), которые появляются только в ядре 2.6.37, а активно используются в разработках для ядер 3.0 и выше.

3. Отрабатываемый модуль следует тестировать и проверять (в сборке модуля) на некоторой «сетке» версий ядра. В противном случае вы рискуете при переустановке у заказчика модуля на следующую версию ядра получить рекламацию: модуль просто не станет компилироваться из-за изменчивости API ядра. Для этого вам, возможно, придётся скомпилировать несколько последовательных версий ядра с одинаковыми конфигурациями. Тестировать модуль далее можно в инсталляциях этих ядер в виртуальных машинах, для проверки на совместимость этого достаточно.

Компиляция, сборка и установка ядра описана в публикациях в интернет сотни раз. Всякий «линуксоид» умеет собирать ядро! Но мы в своём рассмотрении остановимся на некоторых деталях, которые важны применительно к работам по модулям ядра, которые обычно не существенны для эксплуатационщика или администратора, когда они собирают ядро в своих нуждах.

Собирать ядро можно в любом месте файловой системы: в `/usr/src` (как дань традиции), домашнем каталоге пользователя, в каталоге сборки пакета ядра, предопределённом спецификацией сборки (`~/rpmbuild`), или любом другом нравящемся вам месте. Это станет важным уже в самом ближайшем нашем рассмотрении...

Откуда берётся код ядра?

Вопрос этот возникает из тех обстоятельств, что существуют (поддерживается и обновляется) официальные (ещё известные как **ванильные**) версии кода ядра, которые размещаются на

http://www.kernel.org/. Это и есть единственный эталон кода ядра. Но дистрибьюторы вашего дистрибутива (Fedora, CentOS, Debian, Ubuntu, ...) вносят изменения в исходный код ядра, которые оформляют заплатками (утилиты `diff` и `patch`). Это связано с закрытием проблемных мест в безопасности, описанных ко времени сборки дистрибутива, возможно включение каких-то опубликованных кодов, которые не сочли нужным включать официальные разработчики ядра... В любом случае, возникает вопрос: какое ядро берём для сборки? Здесь нет однозначного ответа даже у дистрибьюторов. Можно предложить следующее:

- Если мы собираем текущее (по версии) ядро, но с изменёнными параметрами (это типично, например, для разрешения отладочных опций в ядре), то мы попытаемся использовать исходный код ядра дистрибутива и его конфигурацию в качестве начальной точки изменений.

- Если мы собираем более новое ядро (для проверки и тестирования), то мы не имеем права применять заплатки, предназначенные к старой версии, к исходному коду более позднего ядра. В этом случае берём код официального ядра.

Но рассмотрим мы эти возможности в порядке, обратном перечисленному — так проще для понимания.

Официальное ядро

До самого последнего времени, и на протяжении нескольких десятилетий, исходный код ядра брался в виде архивов `*.tgz` или `*.bz2` из официального источника: http://www.kernel.org/. Но уже в ходе издательской подготовки данного текста, в конце сентября 2011г., официальный источник ядра Linux был разрушен в результате хакерской атаки, больше чем на неделю выведен из строя, после чего было объявлено, что доступ к кодам ядра по протоколу HTTP прекращается. Создаётся новая техника распространения кода ядра, основанная на протоколе контроля версий GIT и программном средстве `gitolite`. Техника установки и использования этого инструмента вскоре станут общеизвестны...

Если же вы получаете код более ранних версий (до 3.04) в виде архива, то вы должны его развернуть в дерево исходных кодов. Для определённости будем считать, что мы разворачиваем дерево в каталоге `/usr/src` и здесь его разархивируем:

```
$ cd /usr/src
$ ls -l linux*
-rw-rw-r-- 1 olej olej 73632687 Map 13 13:33 linux-2.6.37.3.tar.bz2
$ tar -jxvf linux-2.6.37.3.tar.bz2
...
```

Удобно сразу сделать там же ссылку `linux` (так это рекомендуют) на каталог рабочих исходных кодов, как на самый последний вариант. Тогда при последующих изменениях версии ядра вы сможете только переставлять ссылку. Теперь у нас есть дерево исходных кодов для дальнейших действий:

```
$ ln -s linux-2.6.37.3 linux
$ cd linux
$ du -hs
479M    .
```

Показан суммарный объём исходных кодов ядра, мы к нему ещё будем возвращаться...

Если вас устраивает это ядро, и вас не занимает как получить код вашего дистрибутива, то весь дальнейший текст до конфигурации ядра вы можете опустить.

Ядро из репозитария дистрибутива

Большинство дистрибутивов содержат в репозитариях, помимо программных пакетов, пакет исходных кодов ядра этого дистрибутива. Часто этот пакет лежит не в основном репозитарии. То, как получить и использовать патченное ядро дистрибутива, я расскажу на примере дистрибутива Fedora (для RPM-дистрибутивов), где этот процесс, может, сложнее чем в других, но хорошо описан [29]. В других дистрибутивах процесс подобен, здесь важен общий принцип: мы из пакета исходных кодов должны воссоздать то дерево исходных кодов, из которого пакет создавался пакетным построителем (в данном случае `rpmbuild`). Делаем последовательно:

– Воссоздаём общую структуру для построения любого пакета, ещё до загрузки какого-либо пакета. Обратите внимание, что путь построенной структуры каталогов не зависит от места, в котором мы выполняем команду, он зависит от имени пользователя, от которого мы делаем команду: дерево каталогов создаётся в домашнем каталоге этого пользователя (`$HOME/rpmbuild`):

```
$ rpmdev-setuptree
$ tree rpmbuild
rpmbuild
|-- BUILD
|-- RPMS
|-- SOURCES
|-- SPECS
`-- SRPMS
5 directories, 0 files
```

– Теперь из репозитария дистрибутива загружаем пакет исходных кодов (`*.src.rpm`, `yum` здесь не поможет), здесь, может, нужно будет запретить или добавить дополнительные репозитарии, как уже говорилось ранее:

```
$ yumdownloader --source --disablerepo=russianfedora-fixes* kernel*
...
kernel-2.6.35.14-97.fc14.src.rpm                                    |  68 MB     02:20
```

– Разрешаем зависимости (устанавливаем те программные пакеты, которые указаны как необходимые для работы с данным, они нам не понадобятся, но так делать правильно):

```
$ sudo yum-builddep kernel-2.6.35.14-97.fc14.src.rpm
...
Зависимости разрешены

================================================================================
 Пакет                  Архитектура      Версия                  Репозиторий      Размер
================================================================================
Установка:
 asciidoc               noarch           8.4.5-5.fc14            fedora           183 k
 binutils-devel         i686             2.20.51.0.7-8.fc14      updates          733 k
 elfutils-devel         i686             0.152-1.fc14            updates           69 k
 newt-devel             i686             0.52.12-1.fc14          fedora            47 k
 perl-ExtUtils-Embed    noarch           1.28-146.fc14           updates           31 k
 python-devel           i686             2.7-8.fc14.1            fedora           376 k
 redhat-rpm-config      noarch           9.1.0-8.fc14            updates           64 k
 xmlto                  i686             0.0.23-3.fc13           fedora            45 k
Установка зависимостей:
 elfutils-libelf-devel  i686             0.152-1.fc14            updates           31 k
 flex                   i686             2.5.35-11.fc14          fedora           278 k
 flex-static            i686             2.5.35-11.fc14          fedora            12 k
 slang-devel            i686             2.2.3-1.fc14            updates           91 k

Результат операции
================================================================================
Install      12 Package(s)

Объем загрузки: 1.9 M
Будет установлено: 5.6 M
Продолжить? [y/N]: y
...
```

– Все предварительные шаги были подготовительными, вот теперь мы устанавливаем пакет в дерево `~/rpmbuild` (здесь может быть множество предупреждений относительно пользователя и группы сборщика пакета, но мы это игнорируем):

```
$ rpm -Uvh kernel-2.6.35.14-97.fc14.src.rpm
предупреждение: пользователь mockbuild не существует - используется root
предупреждение: группа mockbuild не существует - используется root
...
```

– К этому шагу у нас есть: в каталоге SOURCES архив исходных кодов и много (150) файлов заплаток (*.patch) к нему; в каталоге SPECS спецификацию пакета; и пустой пока каталог BUILD, в котором производится сборка:

```
$ cd /home/olej/rpmbuild/SOURCES
$ ls -l linux*.bz2
-rw-r--r--. 1 olej olej 69305709 Авг  2  2010 linux-2.6.35.tar.bz2
$ ls -l *.patch | wc -l
150
$ cd /home/olej/rpmbuild
BUILD  RPMS  SOURCES  SPECS  SRPMS
$ ls BUILD/
$ du -hs BUILD/
4,0K    BUILD/
$ cd SPECS
$ ls
kernel.spec
```

– Обратите внимание, что архив исходных кодов, показанный на этом шаге, это **официальный архив**, в точности соответствующий тому, что мы получали в предыдущем разделе.

– Разворачиваем (строим) пакет в тот изначальный вид, из которого он создавался, и для той архитектуры, в которой мы это делаем. При построении пакета, в том числе, и накладываются все заплатки на исходное ядро:

```
$ uname -m
i686
$ rpmbuild -bp --target=`uname -m` kernel.spec
Платформы для сборки: i686
Сборка для платформы i686
Выполняется(%prep): /bin/sh -e /var/tmp/rpm-tmp.QV9ZSr
+ umask 022
+ cd /home/olej/rpmbuild/BUILD
...
Patch12086: linux-2.6-cgroups-rcu.patch
+ case "$patch" in
+ patch -p1 -F1 -s
+ ApplyPatch sched-05-avoid-side-effect-of-tickless-idle-on-update_cpu_load.patch
+ local patch=sched-05-avoid-side-effect-of-tickless-idle-on-update_cpu_load.patch
+ shift
+ '[' '!' -f /home/olej/rpmbuild/SOURCES/sched-05-avoid-side-effect-of-tickless-idle-on-
update_cpu_load.patch ']'
...
+ mkdir configs
+ for cfg in 'kernel-2.6.35.14-*.config'
++ grep -c kernel-2.6.35.14-arm.config
++ echo kernel-2.6.35.14-i686-PAE.config kernel-2.6.35.14-i686-PAEdebug.config kernel-2.6.35.14-
i686-debug.config kernel-2.6.35.14
+ '[' 0 -eq 0 ']'
...
+ find . '(' -name '*.orig' -o -name '*~' ')' -exec rm -f '{}' ';'
+ cd ..
+ exit 0
$ cd /home/olej/rpmbuild/BUILD
$ ls
kernel-2.6.35.fc14
```

```
$ du -hs
530M    .
```

– Теперь у нас, в ещё недавно пустом каталоге, находится дерево исходных кодов, объёмом 530Mb, или, точнее, два дерева: официальное дерево (ванильное, о котором мы говорили раньше), и то же самое дерево, на которое наложены заплатки от сборщиков дистрибутива:

```
$ cd /home/olej/rpmbuild/BUILD/kernel-2.6.35.fc14
$ ls
linux-2.6.35.i686  vanilla-2.6.35
$ du -hs linux-2.6.35.i686
457M    linux-2.6.35.i686

$ du -hs vanilla-2.6.35
452M    vanilla-2.6.35
```

– Дистрибутивное дерево отличается ещё тем, что в нём присутствует множество файлов конфигураций, для которых собирался этот дистрибутив (и даже отдельный каталог `configs`). Это хорошее начальное приближение для конфигурации нашей будущей сборки. В официальном дереве файлов конфигурации нет:

```
$ cd /home/olej/rpmbuild/BUILD/kernel-2.6.35.fc14/linux-2.6.35.i686
$ ls *config*
config-arm       config-ia64-generic       config-powerpc32-smp    config-s390x
config-debug     config-local              config-powerpc64        config-sparc64-generic
config-generic   config-nodebug            config-powerpc-generic  config-x86_64-generic
config-i686-PAE  config-powerpc32-generic  config-rhel-generic     config-x86-generic

configs:
kernel-2.6.35.14-i686.config        kernel-2.6.35.14-i686-PAE.config
kernel-2.6.35.14-i686-debug.config  kernel-2.6.35.14-i686-PAEdebug.config
$ cd /home/olej/rpmbuild/BUILD/kernel-2.6.35.fc14/vanilla-2.6.35
$ ls *config*
ls: невозможно получить доступ к *config*: Нет такого файла или каталога
```

– Копируем файл конфигурации, соответствующий нашей архитектуре, вместо текущего файла .config:

```
$ cp configs/kernel-2.6.35.14-i686-PAE.config ./.config
```

– Объявляем это новое содержимое конфигурационным файлом ядра:

```
$ time make oldconfig
scripts/kconfig/conf -o arch/x86/Kconfig
#
# configuration written to .config
#
real    0m0.488s
user    0m0.224s
sys     0m0.163s
```

Теперь мы дошли в этом дереве до того состояния, в котором оставили дерево, скачанное с официального сайта ядра Linux. Дальше мы можем переходить к выбору конфигурации нового ядра.

Конфигурация

Теперь нам предстоит провести конфигурирование ядра, что должно закончиться созданием файла `./config` в каталоге исходных кодов. Хорошей идеей будет использовать в качестве **начального приближения** тот файл `./config`, по которому собиралось ваше текущее рабочее ядро. Обычно копия этого файл (переменованного с указанием имени ядра) сохраняется в `/boot`:

```
$ ls /boot/config*
/boot/config-2.6.18-92.el5  /boot/config-2.6.24.3-1.rt1.2.el5.ccrmart
```

- в этой системе установлено два альтернативных ядра (два варианта загрузки). Один из этих файлов (разобравшись какой из них соответствует загруженной системе) является конфигурацией работающей системы, и такой файл конфигурации может быть скопирован под именем `./config` в каталог исходных кодов.

Перед запуском конфигуратора хорошо сделать очистку каталога от следов предыдущей сборки:

```
$ make mrproper
```

На этом этапе вы можете подправить одну (4-я) строку в `Makefile`, поменяв в ней:

```
EXTRAVERSION =
```

на

```
EXTRAVERSION = myOWN
```

Это приведет к тому, что сделанное вами ядро (и все сопутствующие файлы) будет называться `linux-2.6.37.3-myOWN` (то есть конкатенация версии ядра с суффиксом `EXTRAVERSION`), - так легко различать ваши модификации (и так делают все сборщики дистрибутивов).

Переходим к заданию конфигурации. У нас есть на выбор несколько вариантов целей (в `Makefile`) для выполнения конфигурации:

```
$ make xconfig
$ make gconfig
$ make menuconfig
$ make config
$ make oldconfig
```

Я не вижу оснований, почему имея X11 не пользоваться графическим конфигуратором, но другие альтернативы (из перечисленных) могут быть полезны для малых встроенных конфигураций; выполняем:

```
$ make xconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/basic/docproc
  CHECK   qt
* Unable to find the QT4 tool qmake. Trying to use QT3
*
* Unable to find any QT installation. Please make sure that
* the QT4 or QT3 development package is correctly installed and
* either qmake can be found or install pkg-config or set
* the QTDIR environment variable to the correct location.
...
make[1]: *** Нет правила для сборки цели `scripts/kconfig/.tmp_qtcheck', требуемой для
`scripts/kconfig/qconf.o'.  Останов.
make: *** [xconfig] Ошибка 2
```

Вот так! Уже неоднократно выполняя похожие действия ранее, я попадаю на ошибку выполнения. Потому, что я выполнял это в GNOME, в KDE это, наверное, завершилось бы удачей. Повторяем попытку так:

```
$ make gconfig
  HOSTCC  scripts/kconfig/gconf.o
...
```

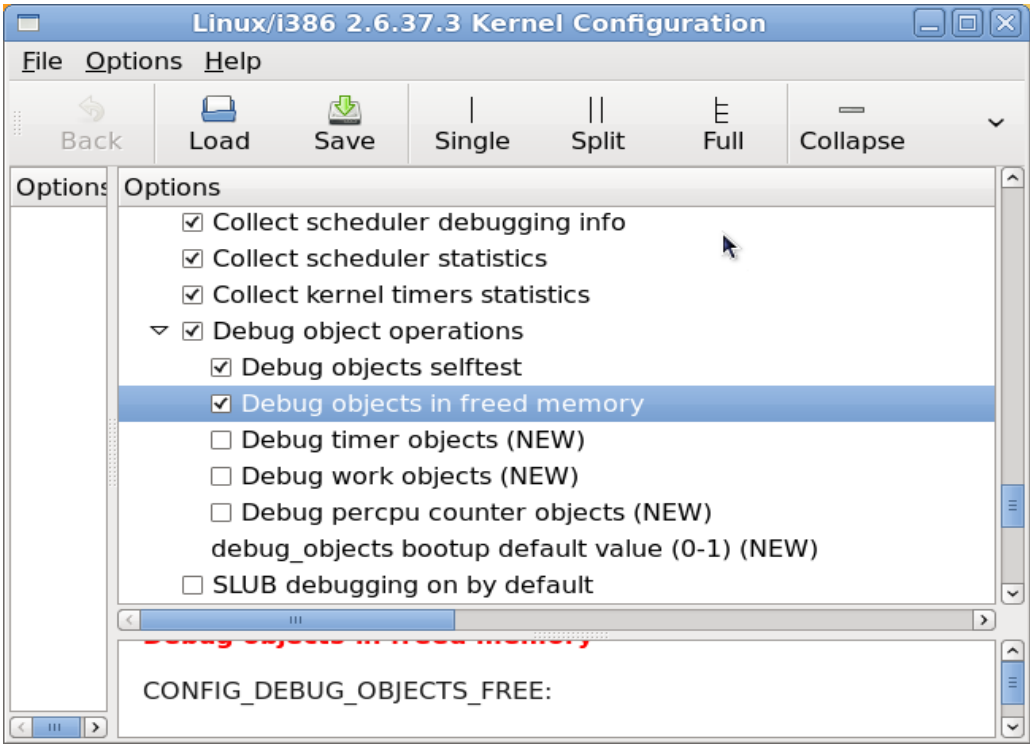
Теперь запуск успешен, как это показано на рисунке, и после того, как мы поэтапно пройдём и завершеним графическое конфигурирование, как позано на рисунке, получим сообщение:

```
# configuration written to .config
#
```

На этом мы имеем сконфигурированное ядро, и можем приступать к его сборке:

```
$ ls -l .config
-rw-rw-r-- 1 olej olej 120513 Map 13 17:00 .config
```

Здесь вы убеждаетесь, что созданный файл конфигурации `.config` отличается от ранее скопированного вами (из `/boot`) как по размеру, так и по времени и дате создания.



Компиляция

Компиляция ядра — это весьма продолжительная операция, даже на быстром процессоре, я буду показывать сборку на 2-х ядерном процессоре:

```
$ 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         : 1666.000
cache size      : 2048 KB
...
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 14
model name      : Genuine Intel(R) CPU           T2300  @ 1.66GHz
stepping        : 8
cpu MHz         : 1667.000
cache size      : 2048 KB
...
```

Компиляция ядра:

```
$ time make bzImage
  HOSTLD  scripts/kconfig/conf
...
  BUILD   arch/x86/boot/bzImage
Root device is (253, 0)
Setup is 14908 bytes (padded to 15360 bytes).
System is 7420 kB
```

```
CRC e39e9d7b
Kernel: arch/x86/boot/bzImage is ready  (#1)
real    23m33.853s
user    19m2.748s
sys     1m48.754s
$ ls -l arch/x86/boot/bzImage
-rw-rw-r-- 1 olej olej 7612704 Map 13 17:27 arch/x86/boot/bzImage
```

Этот процесс занял порядка 25 минут.

Компиляция модулей ядра:

```
$ time make modules
  CHK     include/linux/version.h
...
  Building modules, stage 2.
  MODPOST 2129 modules
...
real    100m0.165s
user    78m59.427s
sys     7m8.274s
```

Компиляция модулей потребовала в 4 раза больше времени, чем компиляция собственно ядра! Но, как вы видите, всё это занятие — не для слабонервных...

Смотрим объём, занимаемый файлами в каталоге исходных кодов:

```
$ du -hs
2,6G    .
```

Объём после компиляции увеличился почти на 2Gb (см. цифру ранее). Вот такой объём свободного места должен обязательно быть на диске для успешной компиляции ядра. Обращаю внимание, что **все операции** до этого места я выполнял **без прав** `root`!

Установка

Устанавливаем модули ядра (до и после установки смотрим состояние каталога `/lib/modules`):

```
$ ls /lib/modules
2.6.32.9-70.fc12.i686.PAE
$ sudo make modules_install
...
  DEPMOD  2.6.37.3
$ ls /lib/modules
2.6.32.9-70.fc12.i686.PAE  2.6.37.3
$ cd /lib/modules/2.6.37.3/
$ du -hs
343M    .
```

У нас появился каталог модулей новой версии: `/lib/modules/2.6.37.3` . Итоговый размер собранных модулей не такой уж и впечатляющий. Теперь устанавливаем собранное ядро (смотрим при этом содержимое `/boot` до и после установки):

```
$ ls /boot
config-2.6.32.9-70.fc12.i686.PAE         lost+found
efi                                      System.map-2.6.32.9-70.fc12.i686.PAE
grub                                     vmlinuz-2.6.32.9-70.fc12.i686.PAE
initramfs-2.6.32.9-70.fc12.i686.PAE.img
$ sudo make install
sh /usr/src/linux-2.6.37.3/arch/x86/boot/install.sh 2.6.37.3 arch/x86/boot/bzImage \
        System.map "/boot"
```

```
$ ls /boot
config-2.6.32.9-70.fc12.i686.PAE          System.map
efi                                       System.map-2.6.32.9-70.fc12.i686.PAE
grub                                      System.map-2.6.37.3
initramfs-2.6.32.9-70.fc12.i686.PAE.img   vmlinuz
initramfs-2.6.37.3.img                    vmlinuz-2.6.32.9-70.fc12.i686.PAE
lost+found                                vmlinuz-2.6.37.3
```

У нас появились 3 новых файла: `vmlinuz-2.6.37.3` — ядро, `initramfs-2.6.37.3.img` – образ начальной загружаемой системы, `System.map-2.6.37.3` — таблица символов нового ядра. Инсталляция ядра 2.6.37.3 в моём примере корректно отредактировала файл меню загрузки `/boot/grub/grub.conf` загрузчика GRUB ... После перезагрузки система стартует с теми установками, с которыми она запускалась раньше:

```
$ uname -r
2.6.37.3
```

С загрузчиком GRUB не всегда выходит так гладко, чтобы он сам безошибочно прописал меню стартовых конфигураций. Но это совсем не сложно, и, отчасти, было затронуто в основном тексте, подредактировать стартовое меню `/boot/grub/grub.conf` под свои вкусы и потребности.

Как ускорить сборку ядра

Одним из главных факторов, делающих сборку столь продолжительной, является скорость накопителя HDD при записи великого множества объектных файлов. Но оборудование многих компьютеров позволяет использовать в качестве устройства хранения для сборки `tmpfs` (RAM диск):

```
$ free
             total       used       free     shared    buffers     cached
Mem:       4124164    1516980    2607184          0     248060     715964
-/+ buffers/cache:     552956    3571208
Swap:      4606972          0    4606972
$ df -m | grep tmp
tmpfs                     2014         1      2014   1% /dev/shm
```

Проверим такую возможность, для этого скопируем дерево исходных кодов в каталог `/dev/shm` :

```
$ pwd
/dev/shm/linux-2.6.35.i686
$ time make bzImage
...
  HOSTCC  arch/x86/boot/tools/build
  BUILD   arch/x86/boot/bzImage
Root device is (8, 1)
Setup is 13052 bytes (padded to 13312 bytes).
System is 3604 kB
CRC 418921f4
Kernel: arch/x86/boot/bzImage is ready  (#1)

real    9m23.986s
user    7m4.826s
sys     1m18.529s
```

Очень неплохой результат: сборка ядра Linux менее чем в 10 минут.

Обсуждение

Мы только что собрали полностью новое ядро Linux, и теперь можем наслаждаться работой в новой, обновлённой до последнего релиза, версии операционной системы. Означает ли это, что такими последовательными обновлениями мы можем поддерживать свою систему в самом свежем состоянии, адекватном новым дистрибутивам? Нет, не означает! Мы таким своим действием обновляем ядро и его модули

(драйвера), но версии всех утилит, библиотек, компиляторов и всего прочего у нас остаются устаревшими. Кроме того, через некоторое время у нас начнутся проблемы с устареванием репозиториев, указанных пакетной системе для поиска обновлений программ. Отстрочить эту проблему мы можем, аккуратно подредактировав вручную ссылки на репозитарии в каталоге `/etc/yum.repos.d` ... но это уже совсем другая история.

Приложение Б: 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 [23]. 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 выше всех остальных). 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 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 B: Пример - открытые VoIP PBX: Asterisk, FreeSwitch, и другие

Отличной практической иллюстрацией ко всему, о чём рассказывалось ранее, есть структура модулей ядра открытых проектов телефонных и VoIP коммутаторов (Soft Switch), таких, как известнейший и старейший в своём классе Asterisk (http://www.asterisk.org), и менее известные (более поздние), но очень динамично развивающиеся: FreeSWITCH (http://www.freeswitch.org/) или YATE (Yet Another Telephony Engine - http://yate.null.ro). Интерес рассмотрения их структуры имеет в своей основе несколько аспектов:

- реализации Soft Switch — это первые подходы к совершенно новым стратегическим технологическим решениям: NGN — New Generation Net: интегральные сети передачи разнородной информации (голос, видео, цифра, мультимедия, ...);

- интерфейс ко всему разнообразию оконечного оборудования (при всём его различии) аналоговых или цифровых (E1/T1) линий для телефонии во всех PBX (при их отличиях), обеспечивается набором модулей канала под общим названием DAHDI (Digium Asterisk Hardware Device Interface, ранее именовавшийся интерфейсом Zaptel), ставшим постфактум стандартом в области IP телефонии;

- вы можете написать (требующий относительно небольшой трудоёмкости) свой небольшой модуль ядра поддержки собственного, проприетарного физического канала обмена данными (хоть кабель параллельного порта), и тем самым интегрировать свой канал в общемировую систему телефонных коммуникаций и сигнализаций;

- таким путём обеспечивается обслуживание физических линий связи во всех этих PBX под самыми разнообразными операционными системами, под которыми реализован слой интерфейсов DAHDI (Linux, FreeBSD, с ограниченной функциональностью Solaris), а отсутствием интерфейсов DAHDI обусловлена невозможность работы с физическими линиями связи в системах семейства Windows (обслуживаются только сетевые сигнализации SIP, H.323 и IAX2);

Интерфейс устройств zaptel/DAHDI

Крайне бегло рассмотрим схематически структуру интерфейса поддержки физических линий связи DAHDI, при этом, для определённости, ограничим рассмотрение:

- исключим их рассмотрения аналоговые сигнализации FXO/FXS как более простые и вписывающиеся в общую схему;

- из цифровых линий с временным уплотнением каналов будем рассматривать только E1 (европейский стандарт), для T1 (американский стандарт) будет всё то же самое с некоторым численными отличиями (24 канала вместо 32);

- стандарт E1 предусматривает уплотнение в один передаваемый кадр 256 битов, разделенных на 32 временных интервала (тайм-слота) по 8 бит в каждом, и содержащих передаваемые данные;

- передача синхронная, скорость передачи составляет 8000 кадров в секунду, что соответствует 2048 kbit/sec для линии и, следовательно, для каждого канала данных (тайм-слота) обеспечивается полоса 64 kbit/sec;

- обычно временной интервал 0 зарезервирован для целей синхронизации, а число доступных пользователю тайм-слотов составляет 31, из которых один (часто) или несколько используются для обеспечения сигнализации (DSS1, PRI, SS7), а остальные — для передачи оцифрованного аудио потока.

Пакет DAHDI (http://downloads.asterisk.org/pub/telephony/) содержит один общий модуль ядра `dahdi.ko`, и по одному модулю ядра для поддержки каждого типа используемого оконечного оборудования (например, плата Digium TE405P/TE407P/TE410P/TE412P: PCI 4 порта T1/E1/J1). Модуль `dahdi.ko` ничего не знает о каналах передачи (от получает данные от канальных модулей), он обеспечивает конфигурирование каналов, обработку управления по сигнализации (PRI, SS7), программное эхо-подавление и другие высокоуровневые функции.

Формирование потоков данных (входных и выходных) осуществляют канальные модули ядра. Точно таким же образом, как и модули из поставки DAHDI, могут быть дописаны и использованы собственные канальные модули для поддержки своей необычной платы, назовём такой модуль, для примера: `xxx.ko`. Такой модуль:

- Должен инициализировать поддерживаемые ним аппаратные каналы (создать PCI устройство, установить обработчик прерывания, настроить DMA...) и вызвать экспортируемую модулем `dahdi` по имени функцию:

  ```
  int dahdi_register( struct dahdi_span *span, int prefmaster );
  ```

  В терминологии DAHDI span — это линия, магистраль, для аналоговой линии связи она будет совпадать с каналом, для E1 span будет включать в себя 31 chan, для T1 — 24 chan.

- При выполнении конфигурирующей программы `/sbin/dahdi_cfg`, модуль `dahdi.ko` читает конфигурацию будущей станции PBX из текстового файла `/etc/dahdi/system.conf`, и создаёт в каталоге устройств (`/dev/dahdi`) набор виртуальных устройств - именованных каналов, в виде единой «плоской» последовательности имён вида: `/dev/dahdi/1`, `/dev/dahdi/2`, `/dev/dahdi/3`... При этом каналы из разных магистральных линий разных технологий (цифровые, аналоговые) «выстраиваются» в единую однородную последовательность каналов, с которыми далее можно работать традиционными API: `read()`, `write()`, ... (`write()` при этом будет соответствовать передаче последовательности байт в соответствующий канал линии, а `read()` - чтению байт из канала).

- Модуль канала `xxx.ko` должен в своём обработчике прерываний (который будет срабатывать строго 8000 раз в секунду — линии синхронные) принимать очередной кадр (31 байт для E1) из линии, и передавать очередной кадр в линию (по DMA). Приём и передача производится в/из накопительных буферов (CHUNK в терминологии DAHDI), размер CHUNK - 8 (`DAHDI_CHUNKSIZE`) кадров.

- При завершении обработки очередного CHUNK (а значит 1000 раз в секунду) модуль `xxx.ko` обменивается следующими порциями данных (размером в CHUNK) с `dahdi.ko`, делая последовательно два вызова (экспортированы `dahdi.ko`):

  ```
  int dahdi_receive( struct dahdi_span *span );

  int dahdi_transmit( struct dahdi_span *span );
  ```

  - только-что принятый и накопленный из линии CHUNK передаётся на уровень модуля `dahdi.ko`, а от него очередной CHUNK поступает для передачи в линию.

- Вся остальная обработка (исключая физическое взаимодействие с линией) осуществляется уровнем модуля `dahdi.ko`, и всеми вышележащими обработчиками PBX (Asterisk, FreeSWITCH, ...) и не требуют никакого вмешательства разработчика канала.

В высшей степени остроумные принятые решения! И все составляющие механизмы для их использования: подключение к шине PCI, установка обработчика прерывания, настройка DMA, экспорт-импорт имён модулями — мы уже рассмотрели в изложении ранее.

Приложение Г: 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]. «The Linux Kernel Module Programming Guide», Peter Jay Salzman, Michael Burian, Ori Pomerantz, 2001.

Перевод: Андрей Киселёв, «Руководство по программированию модулей ядра Linux», 2004:

http://citforum.univ.kiev.ua/operating_systems/linux/lkmpg/

[2]. «Linux Device Drivers», by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, (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]. «Linux Kernel Development», Robert Love, (3rd Edition), 2010.

Русское 2-е издание: Р. Лав, «Разработка ядра Linux», М.: «И.Д.Вильямс», 2006, стр. 448.

[4]. «Professional Linux Kernel Architecture (Wrox Programmer to Programmer)», by Wolfgang Mauerer, Wiley Publishing Inc., 2008, p.1335.

[5]. «Essential Linux Device Drivers», by Sreekrishnan Venkateswaran, Prentice Hall, 2008, p.714.

Сайт книги: http://elinuxdd.com

Архив кодов примеров: http://elinuxdd.com/~elinuxdd/elinuxdd.docs/listings/

[6]. «Writing Linux Device Drivers», Jerry Cooperstein, 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]. «Linux Kernel in a Nutshell» :

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]. Greg Kroab-Hartman, «Linux Kernel in a Nutshell», O'Reilly Vtdia, Inc., 2007, ISBN-10: 0-596-10079-5, ,стр. 184.

[18]. 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.

[19]. У. Р. Стивенс, «UNIX: взаимодействие процессов», СПб.: «Питер», 2003, ISBN 5-318-00534-9, стр. 576.

[20]. 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

[21]. W. Richard Stevens' Home Page (ресурс полного собрания книг и публикаций У. Р. Стивенса):

http://www.kohala.com/start/

[22]. 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

[23]. «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

[24] «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

[25]. Cristian Benvenuti, «Understanding Linux Network Internals», O'Reilly Media, Inc., 2006, ISBN: 978-0-596-00255-8, стр.1035.

[26]. А. Соловьев, «Разработка модулей ядра ОС Linux (Kernel newbie's manual)»:

http://rus-linux.net/MyLDP/BOOKS/knm.pdf

[27]. Зубков С.В. «Assembler для DOS, Windows, UNIX», М.:"ДМК Пресс", 2000, ISBN 5-94074-003-0, стр.608.

[28]. Dmitri Gribenko, «Ассемблер в Linux для программистов С», 06.06.2008 :

http://wasm.ru/article.php?article=asm_linux_for_c

[29]. «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)

[30] 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

[31] «GCC-Inline-Assembly-HOWTO», автор перевода мне неизвестен :

http://www.iakovlev.org/index.html?p=1483&m=1