Модули Linux ядра

Проект книги

Олег Цилюрик,

редакция 4.95

10.08.2011г.

Содержание

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 подразделений (http://globallogic.com.ua) компании. Первоначальный курс, начитанный в тренинговом цикле весны-лета 2011 года в Харькове и составил базовую часть текста. К завершению подготовки курса стало ясно, что большую проведенную работу, главным образом по написанию и отладке примеров, жаль терять бессмысленно, только как иллюстративный материал к тренингам. Более того, несколько моих коллег прошедших лет на протяжении работы обращались с просьбой переслать им материал в том «сыром» виде как он есть, и уверяли, 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 качестве этой книги, 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 (и 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 ядра, или в комбинации: 4 процессора по 4 ядра (пусть это пока и в производительных серверах). Плюс каждое ядро может быть дополнено гипертриэдингом. 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.

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

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 плодотворен 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 совершенное знание языка C — единственного необходимого и достаточного языка системного программирования (из числа компилирующих) в Linux. Это необходимо для самостоятельного анализа и понимания приводимых примеров. Очень продуктивно в дополнение к этому (для работы с многочисленными приводимыми примерами, а ещё больше - их 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 дугие. В Linux используются, наибольшим образом, GNU (FSF) реализации таких утилит, которые набором опций часто отличаются (чаще в сторону расширения) от предписаний стандарта POSIX, и отличаются, порой, от своих собратьев в других операционных системах (Solaris, QNX, ...). Но эти отличия не столь значительны, я думаю, чтобы вызвать какие-либо затруднения.

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» про разработке модулей ядра.

Всё последующее описание — это разные стороны техники написания модулей ядра: управление памятью, взаимопонимание со службой времени, обработка прерываний, программные интерфейсы символьных и сетевых устройств ... и многое другое.

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 параметров, в случае их многочисленности приводится не полностью — разъясняются только наиболее употребимые, акцент делается на понимании...
- Обязательно указываются те места для поиска (заголовочные файлы, файлы описаний) где можно разыскать ту же информацию, актуальную в требуемой 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.
- 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) 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 рассмотрением 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 примеры минимально разумного объёма, а рассказа на качественном уровне, «на пальцах» - я старался избегать.

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, 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 Linux (для большей адекватности, и в силу продолжительности работы), в некоторых инсталляциях каталог `/sbin` не входит в переменную `$PATH` для ординарного пользователя, а править скопированный протокол выполнения команд произвольно я не хотел во избежание внесения несоответствий.

Исходный код 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 обсудаются 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 представлен на самом деле каталогом, содержащим файлы этого примера, поэтому относительно «целеуказания» примеров термины архив и каталог будут употребляться как синонимы. Один и тот же архив может упоминаться несколько раз в самых разных главах описания, это не ошибка: в одной теме он может иллюстрировать структуру, в другой — конкретные механизмы ввода/вывода, в третьей — связывание внешних имён объектных файлов и так далее. Листинги, поэтому, специально не нумерованы, 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 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).

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

Примеры и команды, показываемые в тексте, отрабатывались на нескольких различных инсталляциях Linux:

Fedora 14 - 64-бит инсталляция :

```
$ uname -r
2.6.35.13-91.fc14.x86_64
```

Fedora 14 - 32-бит инсталляция, в дистрибутиве RFRemix :

```
$ uname -r
2.6.35.14-96.fc14.i686.PAE
```

Fedora 12 - 32-бит инсталляция :

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
```

CentOS 5.2 :

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

Ubuntu 10.04.3 LTS:

```
$ uname -r
2.6.32-34-generic
```

Как легко видеть, для проверок и сравнений были использованы дистрибутивы по возможности широкого спектра различий. В других дистрибутивах Linux могут быть 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). 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, но 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 http://en.wikipedia.org/wiki/Comparison_of_operating_system_kernels :

```
...
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 5-ти последних лет (2005-2010) составляло 81 день, или около 12 недель (взято там же).

Источники информации

Самая неоценимая помощь компании Global Logic 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. 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[1], которые все, между тем, 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 Eclipse[2]. Это — в подавляющем большинстве встречаемый случай: пользовательское приложение начинающееся с `main()` и завершающееся по `exit()`.

Ещё один встречающийся (но гораздо реже) в UNIX случай — это демоны: программы, стартующие с `main()`, но никогда не завершающие своей работы (чаще всего это сервера различных служб). В этом случае для того, чтобы стать сервером, всё тот же пользовательский процесс должен выполнить некоторую фиксированную последовательность действий [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[3].

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

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

После чего в модуле начинает выполняться функция инициализации. Возникает вопрос: а можно ли (при необходимости) создать пользовательское приложение, стартующее, как обычно, с точки `main()`, а далее присваивающее себе требуемые привилегии, и выполняющееся в супервизорном режиме (в пространстве ядра)? Да, можно! Для этого изучите исходный код утилиты `insmod` (а Linux — система с абсолютно открытым кодом всех компонент и подсистем), а утилита эта является

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

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

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

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 наличия прав `root`.

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 важной информации о модуле — это список имён модуля (в том числе и экспортируемых 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):

```
$ 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 к файлу модуля добавлено несколько внешних имён, значения которых используются при загрузке модуля для его корректной загрузки.

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()`:

Первому параметру **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 `/lib/modules/2.6.18-92.el5/build/include/linux/kernel.h` :

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

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

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

Вызов `printk()` не производит непосредственно вызов, а направляет вывод 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 `rsyslog`), 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><Alr><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><Alr><F1> - X11, <Ctrl><Alr><F2>...<F7> - текстовые консоли;

- в CentOS 5.2 : <Ctrl><Alr><F1>...<F6> - текстовые консоли, <Ctrl><Alr><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><Alr><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 последовательно показывают установленные уровни вывода; нас интересует первое значение - максимальный уровень сообщений, которые ещё будут выводиться на текстовую консоль.

Записав в этот файл новое значение, можно изменить умалчиваемые 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
```

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 текущем каталоге не указано `./`); 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.

`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

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. 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. Обратите внимание, что в командах загрузки модуля мы всегда записываем:

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

а не :

```
# insmod hello_printk.ko
```

Это происходит потому, что в UNIX текущий рабочий каталог не включается (по умолчанию) в список путей переменной `$PATH`. Можно ли это изменить? Можно, например так:

```
$ export PATH=./:$PATH
```

... но не нужно: считается, что это сильно ухудшает безопасность системы. Совершенно естественно, что при загрузке модуля может быть указано его полное абсолютное путевое имя в файловой системе (это как-раз более правильный подход, и позже мы увидим, что система именно так хранит информацию о известных ей модулях).

Ещё одна особенность (которая досаждает поначалу при работе с модулями): при установке модуля мы говорим:

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

Но при его выгрузке (остановке), 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 подобными написаниями указываем совершенно разные сущности: при установке — имя файла из которого

должен быть установлен модуль ядра, а при выгрузке — имя модуля в RAM пространства ядра, которое (по написанию) только совпадает с именем файла.

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 Darvin 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 (появившаяся с ядер версий 2.0.x).

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) перешли от использования программного прерывания `int` перешли к реализации системного вызова (возврата) через новые команды процессора `sysenter` (`sysexit`). Это было связано с заметной потерей производительности Pentium IV при классическом способе системного вызова. 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 общего числа), куда предварительно эти значения записываются, опять же, специальной новой командой `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».

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

Часть (значительная) вызовов обслуживается непосредственно внутри библиотеки, 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 2 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 3 руководств `man`. Системные вызовы далее преобразовываются в вызов ядра функцией `syscall()`, 1-м параметром которого будет номер требуемого системного вызова, например `NR_execve`. Для конкретности, ещё один пример: вызов `printf(string)`, где: `char *string` — будет трансфороваться в `write(1,string,strlen(string))`, который далее в `sys_call(__NR_write,...)` и далее в `int 0x80` (полный код такого примера 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`. Если системному вызову необходимо передать **большее количество** аргументов, 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.

Пример прямой реализации системного вызова из пользовательского процесса на архитектуре x86 (архив `int80.tgz`) может выглядеть так:

mp.c :

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

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 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 = "write syscall string!\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 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;
```

```
};
```

А вот как происходит выполнение этого примера:

```
$ ./mp
getpid return : 18753
string for write length = 23
write syscall string!
write return : 23
mknod return : -1
$ sudo ./mp
getpid return : 18767
string for write length = 23
write syscall string!
write return : 23
mknod return : 0
$ ls ./Z*
./ZZZ
```

- почему в первом запуске последний вызов завершился ошибкой? Да только потому, что системный вызов `mknod()` (в точности как и консольная команда `mknod` требует прав `root`)! А во всём остальном наша «ручная имитация» системных вызовов завершается успешно (что и подтверждает следующий запуск с правами `root`).

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

Возможен ли системный вызов из модуля?

Такой вопрос мне нередко задавали мне в обсуждениях. Оформим абсолютно тот же код предыдущего примера, но в формате модуля ядра (всё тот же архив `int80.tgz`).

md.c :

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/unistd.h>
#include <linux/string.h>
#include <linux/cdev.h>
#include <linux/fs.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 = "write-call string!";
   int len = strlen( str ) + 1, n;
   printk( KERN_INFO "string for write length = %d\n", len );
   n = write_call( 1, str, len );
   printk( KERN_INFO "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( KERN_INFO "grtpid return : %d\n", n );
}

static int __init hello_init( void ) {
    printk( KERN_INFO "======== start module ==========\n" );
    do_write();
    do_mknod();
    do_getpid();
    return -1;   // конструкция только для тестирования
}

module_init( hello_init );
```

Примечание: Функция инициализации модуля `hello_init()` 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.

Выполняем аналогично тому, как мы это проделывали для процесса пользовательского пространства:

```
$ sudo insmod ./md.ko
insmod: error inserting './md.ko': -1 Operation not permitted
$ sudo cat /var/log/messages | tail -n50
...
Jun 13 14:17:19 notebook kernel: ======== start module ==========
Jun 13 14:17:19 notebook kernel: string for write length = 19
Jun 13 14:17:19 notebook kernel: write return : -14
Jun 13 14:17:19 notebook kernel: mknod return : -14
Jun 13 14:17:19 notebook kernel: grtpid return : 9401
...
```

Вызовы, соответствующие `write()` и `mknod()` завершились ошибкой, но `getpid()` выполнился нормально (сейчас мы не будем останавливаться на вопросе: что за значение он возвратил?). Что произошло? Каждый системный вызов выполняет последовательность действий :

- копирование значений параметров вызова из пространства пользователя в пространство ядра;
- выполнение операции в пространстве ядра;
- копирование значений модифицируемых параметров (передаваемых по ссылке) из пространства ядра в

пространство пользователя;

Вот на первом и/или последнем шагах попытки выполнить системный вызов из модуля и происходит аварийное завершение: в этом случае нет пространства пользователя в которое (или из которого) следует производить копирование данных. Но ничто более не препятствует собственно выполнению кода системного вызова из пространства ядра. Но именно по этой причине (наличие в системном вызове копирования между адресными пространствами, разноадресности параметров и результатов), для работы в пространстве ядра (и из модуля как динамической части ядра) было необходимо создать свой набор API, дублирующий по функциональности большинство библиотечных вызовов и системных вызовов. Например, должно существовать две полностью эквивалентных реализации элементарной и совершенно безобидной (не требующей вмешательства ядра) функции `strlen()`: одна из них будет размещена в теле разделяемой библиотеки `libc.so`, а другая — я коде ядра системы.

Примечание: Возьмите на заметку, что у этих двух обсуждаемых 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, возможно, её хэдер файлы в `/usr/include`) устанавливается, когда вы обновляете версию компилятора. А реализация версии той же функции в ядре принадлежит разработчикам ядра Linux, и будет обновляться когда вы, например, обновляете ядро из репозитария используемого дистрибутива, или самостоятельно собираете ядро из исходных кодов. А эти обновления (компилятора и ядра), как понятно, являются не коррелированными и не синхронизированными. Это не очевидная и часто опускаемая особенность!

Интерфейсы модуля

Модуль (код модуля) может иметь (и пользоваться) набор предоставляемых интерфейсов как в сторону взаимодействия с монолитным ядром Linux (с кодом ядра), так и в сторону взаимодействия с пользователем (пользовательскими приложениями, пространством файловых имён, оборудованием, внешней средой).

Взаимодействие модуля с ядром

Для взаимодействия модуля с ядром, ядро (и подгружаемые модули) экспортируют набор имён, которые 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):

```
$ 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.

Список имён ядра находится в файле `/proc/kallsyms`. Но в этом файле: а). ~85К строчек, и б). далеко не все они 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) хоть может быть и доступно, но не экспортируется ядром, и недоступно для связывания коду модулей (мы ещё 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. Детально изучать их приходится только по исходным кодам Linux.

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) у модуля есть много разнообразынйх 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`;

2. **Копирование данных** в программном коде между пользовательским адресным пространством и пространством ядра (выполняется только по инициативе модуля). Конечно, 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()`[4]). 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):

4 Почему такая асимметрия я не готов сказать.

```
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 далее.

4. Взаимодействие через файлы 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 системы `/sys` (файловая система sys`fs`). 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 (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`) 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 );
```

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
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.

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()`
`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.

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.

```
# 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, экспортируемых имён модуля.

```
$ 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 слаб-алокатора памяти.

`/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 же сами и отказываются от такого своего утверждения и показывают примеры, выполняемые в графическом терминале X11. Тем не менее, нужно отчётливо представлять соотношения текстовых и графических интерфейсов в Linux и их особенностей и ограничений.

==========

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

==========

Графическая подсистема X11 (в реализациях X11R6 или Xorg) не является составной частью операционной системы Linux (UNIX), а является надстройкой пользовательского уровня (даже для работы с видео оборудованием использующей работу с видеоадаптером API пользовательского уровня). Это **принципиально отличает** Linux от систем семейства Windows. О системе графической X11 нужно 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 сети могут приводить к потере работоспособности графической подсистемы.

в). Сетевой протокол X может использовать в качестве транспортного уровня альтернативно различные протоколы, в частности: TCP/IP и потоковый доменный протокол UNIX (UND).

г). Вывод (и ввод) на **терминал** (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 текстовой консоли. Число текстовых консолей (обычно по умолчанию 6) в Linux (в отличие, например, от FreeBSD) — величина легко изменяемая. При работе 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

Основным компилятором Linux является GCC. Но могут использоваться и другие, некоторые примеры таких иных компиляторов (используемых разными коллективами в Linux) являются: а). компилятор CC из состава IDE SolarisStudio операционной системы OpenSolaris, б). активно развивающийся в рамках проекта LLVM компилятор Clang (кандидат для замены GCC в FreeBSD, причина — лицензия), в). PCC (Portable C Compiler) — новая реализация компилятора 70-х годов, широко практикуемый в NetBSD и OpenBSD. Тем не менее, вся эта альтернативность возможна только в проектах пользовательского адресного пространства; в программировании ядра и, соответственно, модулей ядра применим исключительно компилятор GCC.

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 — ещё и поэтому альтернативные компиляторы вполне пригодны для сборки приложений, но 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

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'.
```

Примечание: По моему личному мнению, которое может быть и ошибочно, разработчику модулей ядра Linux совершенно не обязательно умение писать на ассемблере, но в высшей степени на пользу умение хотя бы поверхностно читать написанное не нём. Например, для поиска, в заголовочных файлах или исходных кодах ядра, изменений, произошедших в структурах и API в новой версии ядра.

Нотация AT&T

Ассемблер GCC использует синтаксическую нотацию AT&T, в отличие от нотации Intel (которую используют 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 AT&T строка записанная как:

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

Выглядит в Intel нотации так:

```
mov eax, ebx
```

Основные принципы AT&T нотации:

1. Порядок операндов: <Операция> <Источник>, <Приемник> - в Intel нотации порядок обратный.
2. Названия регистров имеют явный префикс % указывающий, что это регистр. То есть `%eax, %dl, %esi, %xmm1` и т. д. То, что названия регистров не являются зарезервированными словами, — несомненный плюс.
3. 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).

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)                    */
```

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:

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

В простейшем случае это может быть:

```
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;
}
```

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.

Пример использования ассемблерного кода

Для сравнения того, как внешне выглядит функционально идентичный код, записанный на С (`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
```

О сборке модулей детальнее

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`.

Как собрать одновременно несколько модулей?

В уже привычного нам вида `Makefile` может быть описано сборка скольки угодно одновременно собираемых модулей (архив `export.tgz`):

Makefile :

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

Как собрать модуль и использующие программы к нему?

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

Makefile :

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

all: default ioctl

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

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

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

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

Такие файлы содержат общие определения:

ioctl.h :

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

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

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 заменены, соответственно, на `mod_func_B( void )` и `mod_func_C( void )`. 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, всё те же определения переменных — 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]
...
$ 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, например, на каких-то этапах развития, архив примеров к книге имел вид:

```
$ ls
dev  exec         int80  netproto  pci     signal  thread  tools  user_space
dma  first_hello  IRQ    net       parms   proc    sys     time    usb
```

Хотелось бы иметь возможность собирать (или очищать от мусора) всю эту иерархию каталогов-примеров. Для такой цели используем, как вариант, такой `Makefile` :

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.

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!

Нужно ли перекомпилировать ядро?

Для сборки и отработки модулей ядра перекомпиляция самого ядра (и загружаемого образа системы), в обязательном порядке, - **не нужна**. Для работы с модулями достаточно наличия заголовочных файлов ядра (в точности соответствующих загруженной версии ядра!). 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 этом случае нужно доустановить пакет вида:

```
# yum install kernel-devel.x86_64
...
```

```
Установка:
 kernel-devel         x86_64         2.6.35.13-92.fc14         updates         6.6 M
...
Объем загрузки: 6.6 M
Будет установлено: 24 M
...
Установлено:
  kernel-devel.x86_64 0:2.6.35.13-92.fc14
```

- показана установка в 64-разрядной системе , в 32-разрядной, естественно, это будет `kernel-devel.i686`.

В любом случае, мы должны убедиться, что заголовочные файлы, соответствующие 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 сборка ядра Linux, с чего мы начали обсуждение, может оказаться полезной и нужной для сборки ядра с некоторыми специальными качествами, например, с повышенными отладочными уровнями. Для сложных комплексных и долгосрочных проектов сборка рабочей версии ядра желательна.

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 заружаемой 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, только идущий на смену версиям 0.X - «ведёт счёт» начиная с 1!

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

Вся дальнейшая детальная информация по сборке и установке ядра вынесены 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 выполнять такую работу.

2. Представляющие интерес системные файлы перечислены нами на примере дистрибутивов группы RedHat / Fedora / CentOS - проверьте то же самое на примерах Debian / Ubuntu ... или других доступных дистрибутивах.

3. Поупражняйтесь в работе с потоковым редактором `sed`, языком программирования `awk`, или другими подобными средствами работы с текстовыми образцами, широко применяющимися в системах UNIX — они необходимы для работы с системными файлами.

4. 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 из своего программного кода, или посредством консольных команд системы. Такими интерфейсами-связями есть, например, имена 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 взаимодейсвовать. 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 Linuz), 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 парадигам (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. Позже 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 (ядро 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 используется **обработчик по умолчанию**, а не отсутствует обработчик. Такая ситуация (отсутствие переопределённых обработчиков) имеет место достаточно часто, например, в отношении операций `open` и `release` на устройстве, но тем не менее устройства замечательно открываются и закрываются.

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` относится к операциям, которые оперируют с устройствами по их путевым именам, а структура `file_operations` — к операциям, которые оперируют с таким представлением устройств, более понятным программистам, как файловый дескриптор. Но ещё важнее то, что имя ассоциируется с устройством 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, 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 (использующий `register_chrdev()`), статически связывающий модуль со старшим номером, тем самым отдавая под контроль модуля весь диапазон допустимых младших номеров; 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( "5.2" );

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

static ssize_t hello_read( struct file * file, char * buf,
                           size_t count, loff_t *ppos ) {
   int len = strlen( hello_str );
   if( count < len ) return -EINVAL;
   if( *ppos != 0 ) return 0;
   if( copy_to_user( buf, hello_str, len ) ) return -EINVAL;
   *ppos = len;
   return len;
}

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

static void __exit hello_exit( void );
module_exit( hello_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):

hello_dev.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 hello_open( struct inode *n, struct file *f ) {
```

```
   if( device_open ) return -EBUSY;
   device_open++;
   return EOK;
}

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

static const struct file_operations hello_fops = {
   .owner = THIS_MODULE,
   .open = hello_open,
   .release = hello_release,
   .read  = hello_read,
};

#define DEVICE_FIRST  0
#define DEVICE_COUNT  1
#define MODNAME "hello_dev"
static struct cdev hcdev;

static int __init hello_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, &hello_fops );
   hcdev.owner = THIS_MODULE;
   hcdev.ops = &hello_fops;   // обязательно! - cdev_init() недостаточно?
   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 hello_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.

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.

Итак, проверяем работоспособность написанного модуля (немного поэкспериментируем):

```
$ sudo insmod ./hello_dev.ko major=250
insmod: error inserting './hello_dev.ko': -1 Device or resource busy
$ dmesg | tail -n1
Can not register char device region!
$ ls -l /dev | grep ' 250'
crw-rw----  1 root root      250,   0 Июл  2 09:59 hidraw0
crw-rw----  1 root root      250,   1 Июл  2 09:59 hidraw1
crw-rw----  1 root root      250,   2 Июл  2 09:59 hidraw2
```

- здесь нам не повезло: наугад выбранный номер `major` для нашего устройства оказывается уже занятым другим устройством в системе.

Выберем более удачный старший номер:

```
$ sudo insmod ./hello_dev.ko major=200
$ cat /proc/devices | grep hel
200 hello_dev
```

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

```
$ sudo insmod ./hello_dev.ko
$ cat /proc/devices | grep hel
248 hello_dev
```

Но самого такого устройства (с `major` равным 248) у нас пока не существует в `/dev` (мы не сможем пока воспользоваться 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:

```
$ sudo mknod -m0666 /dev/z0 c 248 0
$ cat /dev/z0
Hello, world!
$ sudo rmmod hello_dev
$ cat /dev/z0
cat: /dev/z0: Нет такого устройства или адреса
```

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 dev_t dev;
static struct cdev hcdev;
static struct class *devclass;

static int __init hello_init( void ) {
...
   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 "dyndev"
    device_create( devclass, NULL, dev, "%s", DEVNAME );
...
}

static void __exit hello_exit( void ) {
    device_destroy( devclass, dev );
    class_destroy( devclass );
    cdev_del( &hcdev );
...
}
```

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

```
$ sudo insmod dyndev.ko
$ lsmod | head -n2
Module                  Size  Used by
dyndev                  1366  0
$ ls -l /dev/dyn*
crw-rw---- 1 root root 248, 0 Июн 23 23:28 /dev/dyndev
$ cat /dev/dyndev
Hello, world!
$ cat /proc/modules | grep dyn
dyndev 1366 0 - Live 0xf7ed1000
$ ls -l /sys/class/dyn*
итого 0
lrwxrwxrwx 1 root root 0 Июн 23 23:31 dyndev -> ../../devices/virtual/dyn_class/dyndev
$ sudo rmmod dyndev
$ ls -l /dev/dyn*
ls: невозможно получить доступ к /dev/dyn*: Нет такого файла или каталога
```

Но контроль за номерами устройства, даже создаваемого динамически, сохраняется за модулем, независимо, либо он получает `major` от пользователя при старте, либо запрашивает его у системы. Такое динамическое управление номерами устройства очень упрощает рутинные операции увязывания, и наилучшим образом подходит для создания неких виртуальных устройств-интерфейсов, моделирующих какие-то логические сущности. Примерам такого создания являются общеизвестный интерфейс zaptel/DAHDI к оборудованию, используемый в VoIP проектах SoftSwitch, или подобный интерфейс, развиваемый производителем Sangoma для спектра своего оборудования поддержки линий связи E1/T1/J1 (и E3/T3/J3) — и в том и в другом случае, в `/dev` создаётся набор (возможно, до нескольких сот) фиктивных устройств-имён, взаимно-однозначно соответствующих столь же фиктивным уплотнённым по времени каналам линй E1/T1/J1. Тем не менее, далее можно читать-писать такие устройства совершенно реальными `read()` или `write()`, в точности так, как мы это делаем с реальным физическим устройством «в железе». Бегло структура интерфейса zaptel/DAHDI рассмотрена в качестве наглядной иллюстрации отдельным приложением.

Следующий вариант регистрации драйвера в системе — с динамическим созданием логического «устройства» вне привязки его к фиксированным номерам: сразу же создаётся требуемое имя в `/dev`, с использованием для него произвольных динамических номеров, «понравившихся» системе. Эту технику регистрации драйвера устройста часто называют в литературе как misc (miscelleneous) drivers, и она является частной конкретизацией обсуждавшегося выше механизма, со скрытым использованием использованием `struct cdev`. Это самая простая в кодировании техника создания устройств (из-за её краткости мы будем именно её использовать во множестве дальнейших примеров). Каждое такое устройство создаётся с `major` значением 10, но может выбирать свой уникальный `minor`. Именно поэтому, если драйвер собирается обслуживать целую сетку однотипных устройств, различающихся по `minor`, этот способ не есть хорошим

кандидатом на использование.

В этом варианте драйвер регистрируется и создаёт символическое имя устройства в `/dev` одним единственным вызовом `misc_register()` (архив `misc.tgz`):

hello_dev.c :

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

static const struct file_operations hello_fops = {
   .owner  = THIS_MODULE,
   .read   = hello_read,
};

static struct miscdevice hello_dev = {
   MISC_DYNAMIC_MINOR, "hello", &hello_fops
};

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

static void __exit hello_exit( void ) {
   misc_deregister( &hello_dev );
}
```

Вот, собственно, и весь код всего драйвера. Пример использования такого модуля:

```
$ sudo insmod ./hello_dev.ko
$ lsmod | grep hello
hello_dev               909  0
$ ls -l /dev/hel*
crw-rw---- 1 root root 10, 56 Map  4 00:32 /dev/hello
$ sudo cat /dev/hello
Hello, world!
$ sudo rmmod hello_dev
$ ls -l /dev/hel*
ls: невозможно получить доступ к /dev/hel*: Нет такого файла или каталога
```

Управляющие операции устройства

Здесь (архив `ioctl.tgz`), для реализации уже знакомых нам операций регистрации устройства, мы умышленно воспользуемся, так называемым, старым методом регистрации символьного устройства (`register_chrdev()`). Но эта техника потеряла актуальности, и используются на сегодня — это и будет наш третий альтернативный способ создания устройства. Но, помимо этого, пользуясь тем, что способ регистрации устройства никоим образом не связан с реализующими операции функциями, мы дополним схему нашего устройства обработчиком управляющих операций `ioctl()`, что и есть главной целью нашего следующего примера:

hello_dev.c :

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

// Работа с символьным устройством в старом стиле...
static int hello_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 hello_release( struct inode *n, struct file *f ) {
   return 0;
}

static int hello_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 = hello_open,
   .release = hello_release,
   .read  = hello_read,
   .ioctl = hello_ioctl
};

#define HELLO_MAJOR 200
#define HELLO_MODNAME "hello_dev"
static int __init hello_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 hello_exit( void ) {
  unregister_chrdev( HELLO_MAJOR, HELLO_MODNAME );
}
```

Заголовочный файл, совместно используемый и модулем, и работающим с ним пользовательским процессом, служащий для их согласованного использования типов и констант:

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 )
```

Пользовательский (тестовый) процесс, пользующийся 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( "/dev/hello", 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;
};
```

Makefile :

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

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

all: default ioctl

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

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

Испытываем полученное изделие:

```
$ sudo mknod -m0666 /dev/hello c 200 0
$ ls -l /dev | grep 200
crw-rw-rw-  1 root root    200,   0 Июн 19 00:55 hello
$ cat /dev/hello
cat: /dev/hello: Нет такого устройства или адреса
$ sudo insmod ./hello_dev.ko
$ echo $?
0
$ cat /dev/hello
Hello, world!
$ cat /proc/devices | grep hel
200 hello_dev
$ ./ioctl
Hello, world!
$ sudo rmmod hello_dev
$ ./ioctl
Open device error: No such device or address
$ cat /dev/hello
cat: /dev/hello: Нет такого устройства или адреса
$ ls -l /dev | grep 200
crw-rw-rw-  1 root root    200,   0 Июн 19 00:55 hello
```

```
$ cat /proc/devices | grep hel
$
```

Множественное открытие устройства

В рассмотренных выше вариантах мы совершенно дистанциировались 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`). У нас будет модуль, реализующий все три названных варианта (параметр `mode` запуска модуля):

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( "5.4" );

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;

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 ) return -EBUSY;
   if( 1 == mode ) dev_open++;
   if( 2 == mode ) {
      f->private_data = kmalloc( LEN_MSG + 1, GFP_KERNEL );
      if( NULL == f->private_data ) return -ENOMEM;
      strcpy( f->private_data, "dynamic: not initialized!" );  // динамический буфер
   }
   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 char* get_buffer( struct file *f ) {
   static char static_buf[ LEN_MSG + 1 ] = "static: not initialized!"; // статический буфер :
   switch( mode ) {
      case 0:
      case 1:
      default:
         return static_buf;
      case 2:
         return (char*)f->private_data;
   }
}

// чтение из /dev/mopen :
static ssize_t mopen_read( struct file *f, char *buf, size_t count, loff_t *pos ) {
   static int odd = 0;
   char *buf_msg = get_buffer( f );
   LOG( "read - file: %p, read from %p bytes %d; refcount: %d",
        f, buf_msg, count, module_refcount( THIS_MODULE ) );
   if( 0 == odd ) {
      int res = copy_to_user( (void*)buf, buf_msg, strlen( buf_msg ) );
      odd = 1;
      put_user( '\n', buf + strlen( buf_msg ) );
      res = strlen( buf_msg ) + 1;
      LOG( "return bytes :  %d", res );
      return res;
   }
   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;
   char *buf_msg = get_buffer( f );
   LOG( "write - file: %p, write to %p bytes %d; refcount: %d",
        f, buf_msg, count, module_refcount( THIS_MODULE ) );
   res = copy_from_user( buf_msg, (void*)buf, len );
   if( '\n' == buf_msg[ len -1 ] ) buf_msg[ len -1 ] = '\0';
   else buf_msg[ 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 ) printk( KERN_ERR "unable to register %s misc device", DEVNAM );
   return ret;
}

static void __exit mopen_exit( void ) {
   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 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 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, что при второй попытке открытия устройства возникла ошибка «устройство занято».

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

```
$ sudo insmod ./mmopen.ko 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`) 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 этом мы поговорим детально чуть ниже):

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

Хотя именно то, для чего мы готовили драйвер — 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 реализовываться 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

Вернёмся ещё раз к вопросу счётчика ссылок использования модуля, и, на примере только что спроектированного модуля, внесём для себя окончательную ясность в вопрос. Но 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;

- устройство 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 (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;
   }
   if( par > 0 && ( argc - optind ) < par ) 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[ 40 ] = "/dev/";
   int dfd;                                // дескриптор устройства
   strcat( name, DEVNAME );
   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.

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, чем для предыдущего, но они почти те же:

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

- отличие только в необязательном параметре, который на этот раз несёт смысл: размер блока (в байтах), который читать за одну операцию чтения (если он не указан то читается максимальный размер буфера).

И окончательно наблюдаем как это всё работает...

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.

```
$ 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
```

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
```

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 фаловым именам /proc (procfs) и более поздний 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. 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 «изоретением» 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 C: int, long, ...

Сравним:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 14
model name      : Genuine Intel(R) CPU           T2300  @ 1.66GHz
stepping        : 8
cpu MHz         : 1000.000
...
$ wc -l 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.

Теперь, когда мы кратно пробежались на качественном уровне по свойствам интерфейсов, можно перейти к примерам кода модулей, реализующих первый из этих интерфейсов. Интерфейс /proc рассматривается на примерах из архива proc.tgz. Мы будем собирать несколько однотипных модулей, поэтому общую часть определений снесём в отдельный файл:

mod_proc.h :

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

```
#include <linux/proc_fs.h>
#include <linux/stat.h>
#include <asm/uaccess.h>

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

static int  __init proc_init( void );      // предварительные определения
static void __exit proc_exit( void );
module_init( proc_init );
module_exit( proc_exit );

#define NAME_DIR  "mod_dir"
#define NAME_NODE "mod_node"
#define LEN_MSG 160                        // длина буфера и сам буфер обмена
static char buf_msg[ LEN_MSG + 1 ] = "Hello from module!";
```

Файл сборки общий для всех модулей :

Makefile :

```
CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)
DEST = /lib/modules/$(CURRENT)/misc
EXTRA_CFLAGS += -std=gnu99

TARGET1 = mod_procr
TARGET2 = mod_procr2
TARGET3 = mod_proc
TARGET4 = mod_proct
obj-m   := $(TARGET1).o $(TARGET2).o $(TARGET3).o $(TARGET4).o

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

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 );
```

В результате создаётся изрядно сложная структура, в которой нас могут интересовать, в первую очередь, поля:

```
struct proc_dir_entry {
...
        const char *name;
        mode_t mode;
...
        uid_t uid;
        gid_t gid;
...
        const struct file_operations *proc_fops;
...
        read_proc_t *read_proc;
        write_proc_t *write_proc;
...
};
```

Смысл всех этих полей станет понятным без объяснений из рассмотрения примеров построения модулей.

Первый пример (архив `proc.tgz`) показывает создание интерфейса к модулю в `/proc` доступного только для чтения из пользовательских программ (наиболее частый случай):

mod_procr.c :

```
#include "mod_proc.h"

// в точности списан прототип read_proc_t из <linux/proc_fs.h> :
ssize_t proc_node_read( char *buffer, char **start, off_t off,
                        int count, int *eof, void *data ) {
   static int offset = 0, i;
   printk( KERN_INFO "read: %d\n", count );
   for( i = 0; offset <= LEN_MSG && '\0' != buf_msg[ offset ]; offset++, i++ )
      *( buffer + i ) = buf_msg[ offset ];       // buffer не в пространстве пользователя!
   *( buffer + i ) = '\n';                       // дополним переводом строки
   i++;
   if( offset >= LEN_MSG || '\0' == buf_msg[ offset ] ) {
      offset = 0;
      *eof = 1;                                  // возвращаем признак EOF
   }
   else *eof = 0;
   printk( KERN_INFO "return bytes: %d\n", i );
   if( *eof != 0 ) printk( KERN_INFO "EOF\n" );
   return i;
};
// в литературе утверждается, что для /proc нет API записи, аналогично API чтения,
// но сейчас в <linux/proc_fs.h> есть описание типа (аналогичного типу read_proc_t)
// typedef int (write_proc_t)( struct file *file, const char __user *buffer,
//                             unsigned long count, void *data );

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 = -ENOMEM;
      printk( KERN_ERR "can't create /proc/%s\n", NAME_NODE );
      goto err_node;
   }
   own_proc_node->uid = 0;
   own_proc_node->gid = 0;
   own_proc_node->read_proc = proc_node_read;
   printk( KERN_INFO "module : success!\n");
   return 0;
err_node:                      // обычная для модулей практика использования goto по ошибке
   return ret;
}

static void __exit proc_exit( void ) {
   remove_proc_entry( NAME_NODE, NULL );
   printk( KERN_INFO "/proc/%s removed\n", NAME_NODE );
}
```

Здесь и далее, флаги прав доступа к файлу вида `S_I*` - ищите и заимствуйте в `<linux/stat.h>`.

Испытания:

```
$ make
...
$ sudo insmod ./mod_procr.ko
$ dmesg | tail -n1
module : success!
$ ls -l /proc/mod_*
-r--r--r-- 1 root root 0 Map 26 18:14 /proc/mod_node
$ cat /proc/mod_node
Hello from module!
$ dmesg | tail -n7
module : success!
read: 3072
return bytes: 19
EOF
read: 3072
return bytes: 19
EOF
```

Примечание: Обратите внимание на характерную длину блока чтения в этой реализации, она будет отличаться в последующих реализациях.

Несколько последовательно выполняемых операций:

```
$ cat /proc/mod_node
Hello from module!
$ cat /proc/mod_node
Hello from module!
$ cat /proc/mod_node
Hello from module!
$ sudo rmmod mod_procr
$ ls -l /proc/mod_*
ls: невозможно получить доступ к /proc/mod_*: Нет такого файла или каталога
```

Второй пример делает то же самое, но более простым и более описанным в литературе способом `create_proc_read_entry()` (но этот способ просто скрывает суть происходящего, но делает в точности то же самое):

mod_procr2.c :

```
#include "mod_proc.h"

ssize_t proc_node_read( char *buffer, char **start, off_t off,
                        int count, int *eof, void *data ) {
// ... в точности то, что и в предыдущем случае ...
};

static int __init proc_init( void ) {
   if( create_proc_read_entry( NAME_NODE, 0, NULL, proc_node_read, NULL ) == 0 ) {
      printk( KERN_ERR "can't create /proc/%s\n", NAME_NODE );
      return -ENOMEM;
   }
   printk( KERN_INFO "module : success!\n");
   return 0;
}

static void __exit proc_exit( void ) {
   remove_proc_entry( NAME_NODE, NULL );
   printk( KERN_INFO "/proc/%s removed\n", NAME_NODE );
}
```

Примечание (важно!): `create_proc_read_entry()` пример того, что API ядра, доступный программисту, **намного** шире, чем список экспортируемых имён в `/proc/kallsyms` или `/boot/System.map-2.6.*`, это происходит за счёт множества `inline` определений (как и в этом случае):

```
$ 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 полученного модуля:

```
$ sudo insmod ./mod_procr2.ko
$ echo $?
0
$ cat /proc/mod_node
Hello from module!
$ cat /proc/mod_node
Hello from module!
$ sudo rmmod mod_procr2
$ cat /proc/mod_node
cat: /proc/mod_node: Нет такого файла или каталога
```

Третий пример показывает модуль, который создаёт имя в `/proc`, которое может и читаться и писаться; для этого используется не специальный вызов (типа `read_proc_t`), а структура указатели файловых операций в таблице операций (аналогично тому, как это делалось в драйверах интерфейса `/dev`):

mod_proc.c :

```
#include "mod_proc.h"

static ssize_t node_read( struct file *file, char *buf,
                          size_t count, loff_t *ppos ) {
   static int odd = 0;
   printk( KERN_INFO "read: %d\n", count );
   if( 0 == odd ) {
      int res = copy_to_user( (void*)buf, &buf_msg, strlen( buf_msg ) );
      odd = 1;
      put_user( '\n', buf + strlen( buf_msg ) );   // buf — это адресное пространство пользователя
      res = strlen( buf_msg ) + 1;
      printk( KERN_INFO "return bytes :  %d\n", res );
      return res;
   }
   odd = 0;
   printk( KERN_INFO "EOF\n" );
   return 0;
}

static ssize_t node_write( struct file *file, const char *buf,
                           size_t count, loff_t *ppos ) {
   int res, len = count < LEN_MSG ? count : LEN_MSG;
   printk( KERN_INFO "write: %d\n", count );
   res = copy_from_user( &buf_msg, (void*)buf, len );
```

```
   if( '\n' == buf_msg[ len -1 ] ) buf_msg[ len -1 ] = '\0';
   else buf_msg[ len ] = '\0';
   printk( KERN_INFO "put bytes = %d\n", len );
   return len;
}

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 = -ENOMEM;
      printk( KERN_ERR "can't create /proc/%s\n", NAME_NODE );
      goto err_node;
   }
   own_proc_node->uid = 0;
   own_proc_node->gid = 0;
   own_proc_node->proc_fops = &node_fops;
   printk( KERN_INFO "module : success!\n");
   return 0;
err_node:
   return ret;
}

static void __exit proc_exit( void ) {
   remove_proc_entry( NAME_NODE, NULL );
   printk(KERN_INFO "/proc/%s removed\n", NAME_NODE );
}
```

Обратите внимание, функция чтения `node_read()` в этом случае принципиально отличается от аналогичной функции с тем же именем в предыдущих примерах: не только своей реализацией, но и прототипом вызова, и тем, как она возвращает свои результаты.

Испытания 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
$ dmesg | tail -n1
module : success!
$ cat /proc/mod_node
Hello from module!
$ echo новая строка >  /proc/mod_node
$ cat /proc/mod_node
новая строка
$ cat /proc/mod_node
новая строка
$ dmesg | tail -n10
write: 24
put bytes = 24
read: 32768
return bytes :  24
read: 32768
```

```
EOF
read: 32768
return bytes :  24
read: 32768
EOF
$ sudo rmmod mod_proc
$ cat  /proc/mod_node
cat: /proc/mod_node: Нет такого файла или каталога
```

Примечание: 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 иерархию имён? Как мы наблюдаем это, например, для 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 (показана только изменённая часть предыдущего примера):

mod_proct.c :

```
...
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 = -ENOMEM;
      printk( KERN_ERR "can't create /proc/%s\n", NAME_DIR );
      goto err_dir;
   }
   own_proc_dir->uid = own_proc_dir->gid = 0;
   own_proc_node = create_proc_entry( NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO, own_proc_dir );
   if( NULL == own_proc_node ) {
      ret = -ENOMEM;
      printk( KERN_ERR "can't create /proc/%s\n", NAME_NODE );
      goto err_node;
   }
   own_proc_node->uid = own_proc_node->gid = 0;
   own_proc_node->proc_fops = &node_fops;
   printk( KERN_INFO "module : success!\n");
   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  );
    printk(KERN_INFO "/proc/%s removed\n", NAME_NODE );
}
```

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`.

Теперь смотрим что у нас получилось:

```
$ 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
$ cat /proc/mod_dir/mod_node
Hello from module!
$ echo 'new string' > /proc/mod_dir/mod_node
$ cat /proc/mod_dir/mod_node
new string
$ sudo rmmod mod_proct
```

Интерфейс `/sys`

Одно из главных «приобретений» ядра, начинающееся от версий 2.6 — это появление единой унифицированной модель 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, 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 нас структуры представления символьного устройства:

```
struct cdev {
   struct kobject           kobj;
   struct module           *owner;
   struct file_operations  *ops;
   ...
};
```

Во внешнем представлении в /sys, в интересующем нас смысле, каждому объекту struct kobject соответствует каталог, что видно и из самого определения:

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

Но это вовсе не означает, что каждый инициализированный объект 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. 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;
}
```

Вот каждая такая структура (определена в <linux/sysfs.h>) и является определением одного файла, содержащегося в рассматриваемом каталоге:

```
struct attribute {
...
   char   *name        /* имя атрибута-файла */;
   mode_t  mode struct /* права доступа к файлу */;
}
```

А 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 C (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" );
```

Примечание: В первых строках кода (в виде комментариев) приведены варианты определений (взято из хэдер-файлов), отличающихся даже не между версиями ядра, а между близкими подверсиями ядра: код проверялся в версиях 2.6.32 и 2.6.35 - это лишний раз говорит о волатильности API ядра, и, особенно, ещё не устоявшейся подсистемы `sysfs`.

Тестируем код:

```
$ sudo insmod ./xxx.ko
$ ls -l /sys/class/x-class
-rw-rw-rw- 1 root root 4096 Map 30 21:54 xxx
$ ls -lR /sys/class/x-class
/sys/class/x-class:
-rw-rw-rw- 1 root root 4096 Map 30 21:54 xxx
$ cat /sys/class/x-class/xxx
Hello from module!
$ echo 12345 >  /sys/class/x-class/xxx
$ cat /sys/class/x-class/xxx
12345
$ cat /sys/class/x-class/xxx
12345
$ ls /sys/module/xxx/
holders  initstate  notes  refcnt  sections  srcversion
$ sudo rmmod xxx
$ cat /sys/class/x-class/xxx
cat: /sys/class/x-class/xxx: Нет такого файла или каталога
```

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, весьма условно вписывается в 7-уровневую модель 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, что:

- всё, что относится к поддержке оборудования и канальному уровню — описывается как сетевые интерфейсы;
- протоколы сетевого уровня OSI (IP/IPv4/IPv6 , IPX, ICMP, RIP, OSPF, ARP, ...) — как сетевой уровень стека протоколов (или L2);
- всё, что выше (UDP, TCP, SCTP ...) - как протоколы транспортного уровня (или L3);
- 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.

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.

==========

здесь Рис. 4: сетевые уровни и уровни стека протоколов.

==========

А вот основной структурой обмениваемых данных (между сетевыми уровнями), на движении которой построена работа всех сетевых уровней — 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 в точности соответствует структуре инкапсуляции сетевых протоколов протоколов внутри друг друга, это позволяет обрабатывающему слою доступаться до информации, относящейся только к данному слою.

Драйверы: сетевой интерфейс

Примеры этого раздела заимствованы из [6] и находятся в архиве net.tgz.

lab1_network.c :

```
#include <linux/module.h>
#include <linux/netdevice.h>
#include <linux/init.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;
}

#ifdef HAVE_NET_DEVICE_OPS
static struct net_device_ops ndo = {
   .ndo_open = my_open,
   .ndo_stop = my_close,
   .ndo_start_xmit = stub_start_xmit,
};
#endif

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 );
#ifdef HAVE_NET_DEVICE_OPS
   dev->netdev_ops = &ndo;
#else
   dev->open = my_open;
   dev->stop = my_close;
   dev->hard_start_xmit = stub_start_xmit;
#endif
}

static int __init my_init( void ) {
   printk( KERN_INFO "Loading stub network module:...." );
   dev = alloc_netdev( 0, "mynet%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\n", dev_name( &dev->dev ) );
   return 0;
}

static void __exit my_exit( void ) {
   printk( KERN_INFO "Unloading stub network module\n\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" );
```

Это сетевое устройство уже можно установить:

```
$ sudo ifconfig mynet0
mynet0: error fetching interface information: Device not found
$ sudo insmod lab1_network.ko
$ lsmod | head -n2
Module                  Size  Used by
lab1_network            1172  0
$ dmesg | tail -n6
Loading stub network module:....
my_setup()
Succeeded in loading mynet0!
Hit: my_open(mynet0)
mynet0: no IPv6 routers present
```

И теперь мы можем с ним немного поработать:

```
$ sudo ifconfig mynet0
mynet0    Link encap:Ethernet  HWaddr 00:01:02:03:04:05
          BROADCAST 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)
$ sudo ifconfig mynet0 up 192.168.1.200
$ sudo ifconfig mynet0
mynet0    Link encap:Ethernet  HWaddr 00:01:02:03:04:05
          inet addr:192.168.1.200  Bcast:192.168.1.255  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)
$ ping 192.168.1.200
PING 192.168.1.200 (192.168.1.200) 56(84) bytes of data.
64 bytes from 192.168.1.200: icmp_seq=1 ttl=64 time=0.056 ms
64 bytes from 192.168.1.200: icmp_seq=2 ttl=64 time=0.051 ms
64 bytes from 192.168.1.200: icmp_seq=3 ttl=64 time=0.055 ms
64 bytes from 192.168.1.200: icmp_seq=4 ttl=64 time=0.052 ms
64 bytes from 192.168.1.200: icmp_seq=5 ttl=64 time=0.054 ms
64 bytes from 192.168.1.200: icmp_seq=6 ttl=64 time=0.052 ms
^C
--- 192.168.1.200 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5440ms
rtt min/avg/max/mdev = 0.051/0.053/0.056/0.006 ms
```

Удаляем сетевой интерфейс:

```
$ sudo rmmod lab1_network
$ dmesg | tail -n3
Unloading stub network module
Hit: my_close(mynet0)
$ sudo ifconfig mynet0
mynet0: error fetching interface information: Device not found
```

Как уже отмечалось выше, основу структуры описания сетевого интерфейса составляет структура `struct net_device`, описанная в `<linux/netdevice.h>`. Это очень крупная структура, содержащая не только описание аппаратных средств, но и конфигурационные параметры сетевого интерфейса по отношению к выше лежащим протоколам, например:

```
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 */
...
   /* 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».

Все структуры, описывающие доступные сетевые интерфейсы в системе, увязаны в связный список. Следующий пример диагностирует такой список:

lab1_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 0;
```

```
}
static void __exit my_exit( void ) {
   printk( KERN_INFO "Module Unloading\n" );
}

module_init( my_init );
module_exit( my_exit );

MODULE_AUTHOR( "Jerry Cooperstein" );
MODULE_DESCRIPTION( "LDD:1.0 s_25/lab1_devices.c" );
MODULE_LICENSE( "GPL v2" );
```

Выполнение (предварительно для убедительности загрузим ранее созданный модуль `lab1_network.ko`):

```
$ sudo insmod lab1_network.ko
$ sudo insmod lab1_devices.ko
$ 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:

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 (см. ранее) для этой линии. Последним действием является передача заполненного сокетного буфера вызову netif_rx(), который и запустит процесс движения его (буфера) вверх по структуре сетевого стека.

Этим обеспечивается движение сокетного буфера вверх по стеку. Движение вниз (при отправке в сеть) обеспечивается по цепочке.

4. При инициализации сетевого интерфейса (это момент, который уже был назван 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,
};
```

5. 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, 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 API сокетов прикладного уровня.

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 );
```

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" );
```

Выполнение 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 на 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 ) {
    if( inet_add_protocol( &test_proto, PROTO ) < 0 ) {
        printk( KERN_INFO "proto init: can't add protocol\n");
        return -EAGAIN;
    };
    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" );
```

```
$ sudo insmod trn_proto.ko
$ lsmod | head  -n2
Module                  Size  Used by
trn_proto               780  0
$ dmesg | tail -n2
proto init: can't add protocol
proto module loaded
$ 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
...
```

Статистики

Для накопления статистики работы сетевого интерфейса описана структура (весьма большая, определена 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  */
...
}
```

Такая структура должна заполняться кодом модуля статистическими данными проходящих пакетов. Это делается, если вы хотите получать статистики сетевого интерфейса пользователем через интерфейс файловой системы /proc, как это происходит для других сетевых интерфейсов.

Внутренние механизмы ядра

«Очень трудно видеть и понимать неизбежное в хаосе вероятного»

Андрей Ваджра (псевдоним украинского 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 исключительно разработчику модуля в качестве строительных конструкций для реализации своих замыслов. Большинство понятий этой части описания уже знакомо по API пользовательского пространства, и имеют там прямые аналогии. Но существуют и некоторые принципиальные расхождения.

Механизмы управление памятью

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 системными вызовами, которые ретранслируются в рассматриваемые здесь механизмы. Исключение составляет один `alloca()`, который распределяет память непосредственно из стека выполняемой функции (что имеет свою опасность в использовании). Таким образом, рассматриваемые вопросы имеют прямой практический интерес и для прикладного программирования (пространства пользователя).

Механизмы динамического управления памятью в коде модулей (ядра) имеют два главных направления использования:

1. Однократное распределение буферов данных (иногда достаточно и объёмных и сложно структурированных), которое выполняется, как правило, при начальной инициализации модуля (в сетевых драйверах часто при активизации интерфейса командой ifconfig);

2. Многократное динамическое создание-уничтожение временных структур, организованных в некоторые списочные структуры;

Первоначально мы рассматриваем механизмы первой названной группы (которые, собственно, и являются механизмами динамического управления памятью), но к концу раздела отклонимся и рассмотрим использование циклических двусвязных списков, ввиду их максимально широкого использования в ядре Linux (и призывов разработчиков ядра использовать только эти, или подобные им, там же описанные, структуры).

Динамическое выделение участка

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. Только некоторые (наиболее используемые флаги):

- `GFP_KERNEL` - выделение производится от имени процесса, который выполняет системный запрос в пространстве ядра — такой запрос может быть временно переводиться в пассивное состояние (блокирован).

- `GFP_ATOMIC` - 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.

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 диспетчированием 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 в heap блок, превышающий запрошенный размер), либо best-fit (блок в heap, 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.

Ещё один подход, применявшийся в Linux (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.

Последний алгоритм распределителя, использующийся как основной в версиях ядра 2.6, это слаб алокатор (slab allocation). Слабовый распределитель впервые предложен Джефом Бонвиком (Jeff Bonwick), реализован и описан в SunOS (в середине 90-х годов). Идея такого распределителя состоит в том, что запросы на выделение памяти под объекты равного размера удовлетворяются из области одного кэша (слаба), а запросы на объекты другого размера (пусть отличающиеся от первого случая самым незначительным образом) - удовлетворяются из совершенно другого такого же кэша.

Примечание: Сам термин слаб переводится близко к «облицовочная плитка», и принцип очень похож: любую вынутую из плоскости плитку можно заменить другой такой же, потому, что их размеры в точности совпадают.

Использование слаб алокатора может быть отменено при сборке ядра (параметр `CONFIG_SLAB`). Это имеет смысл и используется для небольших и встроенных систем. При таком решении включается алокатор, который называют SLOB, он может экономить до 512KB памяти, но страдает названными уже недостатками, главный из которых - фрагментация. Дальше детально мы будем рассматривать только слабовый распределитель.

Слабовый распределитель[5]

Текущее состояние слабового распределителя можем рассмотреть в файловой системе `/proc` (что даёт достаточно много для понимания самого принципа слабового 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 слаб должен быть создан (зарегистрирован) вызовом `kmem_cache_create()`, а

5 В литературе (публикациях) мне встречалось русскоязычное наименование такого распределителя как: «слабовый», «слябовый», «слэбовый»... Поскольку термин нужно как-то именовать, а ни одна из транскрипций не лучше других, то я буду пользоваться именно первым произношением из перечисленных.

потом из него можно «черпать» элементы фиксированного размера (под который и был создан слаб) вызовами `kmem_cache_alloc()` (это и есть тот вызов, в который, в конечном итоге, с наибольшей вероятностью ретранслируется ваш `kmalloc()`). Все сопутствующие описания ищите в `<linux/slab.h>`. Так это выглядит на качественном уровне. А вот при переходе к деталям начинается цирк, который состоит в том, что прототип функции `kmem_cache_create()` меняется от версии к версии.

В версии 2.6.18 и **практически во всей литературе** этот вызов описан так:

```
kmem_cache_t *kmem_cache_create( const char *name, size_t size,
                  size_t offset, unsigned long flags,
                  void (*ctor)( void*, kmem_cache_t*, unsigned long flags ),
                  void (*dtor)( void*, kmem_cache_t*, unsigned long flags ) );
```

`name` — строка имени кэша;

`size` — размер элементов кэша (единый и общий для всех элементов);

`offset` — смещение первого элемента от начала кэша (для обеспечения соответствующего выравнивания по границам страниц, достаточно указать 0, что означает выравнивание по умолчанию);

`flags` — опциональные параметры (может быть 0);

`ctor, dtor` — **конструктор** и **деструктор**, соответственно, вызываются при размещении-освобождении каждого элемента, но с некоторыми ограничениями ... например, деструктор будет вызываться (финализация), но не гарантируется, что это будет поисходить сразу непосредственно после удаления объекта.

К версии 2.6.24 [5, 6] он становится другим (деструктор исчезает из описания):

```
struct kmem_cache *kmem_cache_create( const char *name, size_t size,
                        size_t offset, unsigned long flags,
                        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 слабе должно выравниваться по строкам процессорного кэша, это может существенно поднять производительность, но непродуктивно расходуется память;

`SLAB_POISON` — начально заполняет слаб 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 слаба:

```
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 слаб 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 слабе 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 слаба никаким образом не указывается реальный или максимальный объём памяти, находящейся под управлением этого слаба: 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 слаб, 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 слаб размечено некоторое фиксированное количество фиксированных объекто-мест (339 в последнем примере), которые укладываются в некоторый начальный объём слаба 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[6].

Ещё одна вариация на тему распределителя памяти, в тос числе и слаб-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 );
```

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

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 слаб 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 слаб алокатор для управления памятью, часто выглядит следующим образом:

```
// создание нового слаба
kmem_cache_t *cache = kmem_cache_create( ... );
// создание пула, который будет распределять память из этого слаба
mempool_t *pool = mempool_create( MY_POOL_MINIMUM, mempool_alloc_slab, mempool_free_slab, cache );
```

После того, как пул был создан, объекты могут быть выделены и освобождены с помощью:

```
void *mempool_alloc_slab( gfp_t gfp_mask, void *pool_data );
void mempool_free_slab( void *element, void *pool_data );
```

После создания пула памяти функция выделения будет вызвана достаточное число раз для создания пула предопределённых объектов. После этого вызовы `mempool_alloc_slab()` пытаются получить новые объекты от функции выделения - возвращается один из предопределённых объектов (если таковые сохранились). Когда объект освобождён `mempool_free_slab()`, он сохраняется в пуле если количество предопределённых объектов в настоящее время ниже минимального, в противном случае он будет возвращён в систему.

Примечание: Такие же группы API есть для использования в качестве распределителя памяти для пула `kmalloc()` (`mempool_kmalloc()`) и страничного распределителя памяти (`mempool_alloc_pages()`).

Размер пула памяти может быть динамически изменён:

```
int mempool_resize( mempool_t *pool, int new_min_nr, int gfp_mask );
```

- в случае успеха этот вызов изменяет размеры пула так, чтобы иметь по крайней мере `new_min_nr` объектов.

Когда пул памяти больше не нужен он возвращается системе:

```
void mempool_destroy( mempool_t *pool );
```

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 /* наносекунды */ };
```

- 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 числел: 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»

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 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, 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). И на это обстоятельство нужно обращать особое внимание — оно может стать предметом очень серьёзных шибок!

Потоки 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 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);
```

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.

Синхронизации

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 виде, как она была впервые описана Э. Дейкстрой в его знаменитой работе 1968 г. «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 блокирумости кода, работающего с локальными данными процесса, остаётся актуальным и в этом случае.

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 последовательность выполнения, 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 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, определёно ли поле владельца захватившего мютекс (в различных POSIX ОС, и в мютексах пространства ядра и пространства пользователя), 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.
```

Для определения и инициализации используем макросы (эквивалентные по смыслу записанным выше) вида:

```
DEFINE_SPINLOCK( lock );
DEFINE_RWLOCK( lock );
```

Основной интерфейс `spinlock_t` :

```
spin_lock ( spinlock_t *sl );
spin_unlock( spinlock_t *sl );
```

Примечание: Если при компиляции ядра не установлено SMP и не конфигурировано вытеснение кода в ядре (наличие 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 (скорее всего, вместе с процессором его выполняющим, то есть происходит деградация системы — число доступных 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,

`void spin_lock_init( spinlock_t *sl );` - динамическая инициализация спин-блокировки

`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.

В случае семафоров, вместо структуры `struct semaphore` вводится `struct rw_semaphore`, а набор интерфейсных функций захвата освобождения (`down()/up()`) расширяется до:

`down_read( &mr_rwsem );` - попытка захватить семафор для чтения

`up_read( &rar_rwsem );` - освобождение семафора для чтения

`down_write(&mr_rwsem);` - попытка захватить семафор для записи

`up write(&mr 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 фактически не семафорами, а мютексами.

Пример того, как могут быть использованы семафоры чтения-записи:

```
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 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 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.

Сериальные (последовательные) блокировки

Это пример одного только из нескольких механизмов синхронизации, которые и блокировками по существу 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 динамически:

```
seqlock_t lock;
seqlock_init( &lock );
```

Доступ на чтение работает получая беззнаковое целочисленное значение последовательности на входе в защищаемую критическую секцию. На выходе из этой секции это значение должно сравниваться с текущим таким значением; 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 блокировку, чтобы войти в критическую секцию, защищаемую последовательной блокировкой. Чтобы это сделать, код писателя делает вызов функции:

```
void write_seqlock( seqlock_t *lock );
```

Снятие блокировки записи:

```
void write_sequnlock( seqlock_t *lock );
```

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).

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?).

Одним из таких механизмов являются определённые в `<linux/compiler.h>` макросы `likely()` и `unlikely()`, например:

```
if( unlikely() ) {
   /* сделать нечто редкостное */
};
```

Или:

```
if( likely() ) {
```

```
    /* обычное прохождение вычислений */
}
else {
    /* что-то нетрадиционное */
};
```

Такие предписания а). имеют целью оптимизацию компилированного кода, 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.

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`).

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 точнее, то показано 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 и вопросы

При рассмотрении техники обработки прерываний возникает ряд тонких вопросов, на которые меня натолкнули участники проводимых тренингов, но на которые я пока не знаю ответа. Но такие вопросы должны быть названы, а получить на них ответы достаточно несложно, подготовив соответствующие тестовые примеры, и в ближайшее время я надеюсь представить такие примеры. Итак:

1. 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?

2. При регистрации нескольких обработчиков прерываний, разделяющих одну линию IRQ, есть ли способы изменения последовательности срабатывания этих нескольких обработчиков?

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.

В отношении анализа всего установленного в системе оборудования, начиная с анализа изготовителя и 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. Информация от этих команд в какой-то мере дублирует друг друга. Но сбор такой 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 вывода, для идентификации того, что это именно та команда) — более детальное обсуждение увело бы нас слишком далеко от наших целей. Вот некоторые такие команды:

```
# sudo lshw
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
...
$ 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 памяти куда установлены.

Устройства на шине PCI

Архитектура PCI был разработана в качестве замены стандарту ISA с тремя основными целями: получить лучшую производительность при передаче данных между компьютером и его периферией, быть независимой от платформы, насколько это возможно, и упростить добавление и удаление периферийных устройств в системе. В настоящее время PCI широко используется в разных архитектурах: IA-32 / IA-64, Alpha, PowerPC, SPARC64 ... Самой актуальной для автора драйвера является поддержка PCI автоопределения интерфейса плат: PCI

устройства настраивается автоматически во время загрузки. Затем драйвер устройства получает доступ к информации о конфигурации устройства, и производит инициализацию. Это происходит без необходимости совершать какое-либо тестирование.

Каждое периферийное устройство PCI идентифицируется 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.

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)
```

Другое представление той же информации (тот же хост) 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: `0000:00:02.0` - выделены домен (16 бит), шина (8 бит), устройство (5 бит) и функция (3 бита). Поэтому, когда мы говорим об устройстве (далее), мы имеем в виду набор: номера домена + номер шины + номер устройства + номер функции.

С другой стороны, 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. 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 `PCI_ANY_ID=-1`, то предполагается выбор 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 );
```

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 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)
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" );
```

Небольшой фрагмент результата выполнения этого 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
$
```

Для использования некоторой группы устройства PCI, код модуля определяет массив 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, },
};
```

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 может 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>; некоторые из них:

IORESOURCE_IO, IORESOURCE_MEM — только один из этих флагов может быть установлен.

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 удаления устройства;
- ... 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 );
```

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 здесь указывалось фиксированное значение, устанавливаемое механически на плате устройства (переключателями, джамперами, ...). 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`, 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 адреса, 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 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, отображённая 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 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). 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. Буфера DMA могут выделяться только в строго определённых областях памяти:

- эта память должна распределяться в **физически** непрерывной области памяти, 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 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, 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:

```
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 );
```

Примеры использования API однотипны и достаточно громоздки. Некоторые примеры использования показаны в архиве `dma.tgz`, результаты выполнения показаны там же в файле `dma.hist`.

Устройства USB

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 003: ID 0461:4d17 Primax Electronics, Ltd Optical Mouse
Bus 004 Device 002: ID 0458:0708 KYE Systems Corp. (Mouse Systems)
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 006: ID 08ff:2580 AuthenTec, Inc. AES2501 Fingerprint Sensor
Bus 001 Device 003: 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
```

Список идентификаторов USB (производитель:устройство) 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 . Для одного из приведенных выше устройств (WEB-камеры), для которого мы будем проводить тест:

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

Подключение (отключение) 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):

lab1_usb.c :

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

struct my_usb_info {
   int connect_count;
};

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( KERN_INFO "\nmy_usb_probe\n" );
   printk( KERN_INFO "devnum=%d, speed=%d\n", dev->devnum, (int)dev->speed );
   printk( KERN_INFO "idVendor=0x%hX, idProduct=0x%hX, bcdDevice=0x%hX\n",
           dev->descriptor.idVendor,
           dev->descriptor.idProduct, dev->descriptor.bcdDevice );
   printk( KERN_INFO "class=0x%hX, subclass=0x%hX\n",
           dev->descriptor.bDeviceClass, dev->descriptor.bDeviceSubClass );
   printk( KERN_INFO "protocol=0x%hX, packetsize=%hu\n",
           dev->descriptor.bDeviceProtocol,
           dev->descriptor.bMaxPacketSize0 );
   printk( KERN_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( KERN_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( KERN_INFO "\nmy_usb_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-hotplug",
   .probe = my_usb_probe,
   .disconnect = my_usb_disconnect,
   .id_table = my_usb_table,
};

static int __init my_init_module( void ) {
   int err;
   printk( KERN_INFO "Hello USB\n" );
   err = usb_register( &my_usb_driver );
   return err;
}

static void my_cleanup_module( void ) {
   printk( KERN_INFO "Goodbye USB\n" );
   usb_deregister( &my_usb_driver );
}

module_init( my_init_module );
module_exit( my_cleanup_module );

MODULE_AUTHOR( "Terry Griffin" );
MODULE_DESCRIPTION( "LDD:1.0 s_28/lab1_usb.c" );
MODULE_LICENSE( "GPL v2" );
```

Работа полученного модуля:

```
$ sudo insmod lab1_usb.ko.
$ lsmod | grep lab
lab1_usb                1546  0.
$ dmesg | tail -n 10
Hello USB
usbcore: registered new interface driver usb-hotplug
```

... размыкаем кабель USB-камеры :

```
$ dmesg | tail -n 3
Hello USB
usbcore: registered new interface driver usb-hotplug
usb 1-3: USB disconnect, address 3
```

... снова подключаем кабель USB-камеры :

```
$ dmesg | tail -n 10
Hello USB
usbcore: registered new interface driver usb-hotplug
usb 1-3: USB disconnect, address 3
usb 1-3: new high speed USB device using ehci_hcd and address 7
usb 1-3: New USB device found, idVendor=046d, idProduct=080f
usb 1-3: New USB device strings: Mfr=0, Product=0, SerialNumber=2
usb 1-3: SerialNumber: 1DC23270
usb 1-3: 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/usb1/1-3/1-3:1.0/input/input13
$ sudo rmmod lab1_usb
$ dmesg | tail -n 2
Goodbye USB
usbcore: deregistering interface driver usb-hotplug
```

Более экзотические возможности

Существует ещё большое разнообразие возможностей для программиста, пишущего в адресном пространстве ядра (или для поддержки собственных операций в пространстве ядра). Они используются гораздо реже, чем типовые средства, разбираемые выше, но они весьма значимы, и некоторые из них должны быть хотя бы коротко обозначены.

Примечание: Многие из таких возможностей, реализующие действия, аналогичные таким же в пространстве пользователя (в более привычном контексте), в литературе и обсуждениях относят к общей группе «хелперы», где информацию о них и следует искать.

Запуск процессов из ядра

Новые процессы пользовательского пространства могут создаваться кодом ядра, так же, как они создаются и пользовательским кодом вызовами группы `exec*()`.

Примечание: Процессы из пользовательского кода создаются в два шага: выполнением `fork()` создаётся новое адресное пространство, которое и станет пространством нового процесса, после чего вызывается функция семейства `exec()`.

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", "this is wall message", NULL };
   static char *envp[] = { NULL };
   if( str ) argv[ 1 ] = str;
   rc = call_usermodehelper( "/usr/bin/wall", argv, envp, 0 );
   if( rc )
      printk( KERN_INFO "failed to execute : %s %s\n", argv[ 0 ], argv[ 1 ] );
   else
      printk( KERN_INFO "execute : %s %s\n", argv[ 0 ], argv[ 1 ] );
   return -1;
}

module_init( exec_init );
MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "Oleg Tsiliuric <olej@front.ru>" );
MODULE_VERSION( "1.1" );
```

Вызов `call_usermodehelper()` получает параметры точно так же, как вызов пользовательского пространства `execve()` (через который выполняются все вызовы семейства `exec*()`), детали смотрите в справочной странице :

```
$ man 2 execve
```

Срабатывание созданного модуля:

```
$ sudo insmod mod_exec.ko
this is wall messageom root@notebook.localdomain (Tue May  3 22:32:00 2011):
insmod: error inserting 'mod_exec.ko': -1 Operation not permitted
$ dmesg | tail -n1
execute : wall this is wall message
```

- модуль успешно загружается, видно нормальный запуск автономного пользовательского приложения... некоторый диссонанс вносит нарушение целостности текстового сообщения, посланного терминалу, в типовом запуске этой программы это должно выглядеть так:

```
$ wall this is wall message
Broadcast message from olej@notebook.localdomain (pts/2) (Tue May  3 22:33:57this is wall message
```

В этом и есть особенность и ограниченность метода: процесс запускается без управляющего терминала и с нестандартным для него окружением! Если в качестве пользовательского процесса, заменив строку запуска, использовать:

```
char *argv[] = { "/bin/echo", "this is wall message", NULL };
```

То результатом будет:

```
$ sudo insmod mod_exec.ko str=XXX
insmod: error inserting 'mod_exec.ko': -1 Operation not permitted
$ dmesg | tail -n1
execute : /bin/echo XXX
```

- мы имеем совершенно нормальный запуск пользовательского процесса ... но в глухо-немом варианте, без ожидаемого вывода в терминал, поскольку не существует терминала, в который надлежало бы производить такой вывод.

Всю основную работу по созданию и запуску процесса, как легко видеть, выполняет вызов (<linux/kmod.h>):

```
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

Точно так же, как запуск процесса, по аналогии с пользовательским пространством, можно посылать из ядра сигналы UNIX как пользовательским процессам, так и потокам пространства ядра. Для уяснения возможностей использования сигналов из ядра (и в ядре) я воспользовался несколько видоизменённым (архив signal.tgz) проектом из [6]. Идея теста проста:

- имеем пользовательское приложение sigreq («мишень» на которую направляются сигналы), и которое регистрирует полученные сигналы;
- имеем модуль ядра lab3_ioctl_signal.ko, которому можно «заказать»: какому процессу (PID) отсылать сигнал и какой сигнал, пользовательское приложение, в качестве целеуказания мы и будем указывать sigreq;
- и имеется диалоговый пользовательский процесс ioctl, который казывает модулю ядра: какой сигнал отсылать и кому.

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:

lab3_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_DESCRIPTION("LDD:1.0 s_13/lab3_ioctl_signal.c");
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:

```
#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 последовательность команда `ioctl()` над модулем: 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:

```
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 );
}
```

Примечание: 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 выберем безобидный 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 lab3_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 устройством может быть выполнен в пространстве пользователя. Случаем, в котором работа в пространстве пользователя может иметь смысл, является тот, когда вы начинаете иметь дело с новым и необычным оборудованием. 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;

Модификация 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). Иногда хотелось бы подменить или добавить позицию (адрес обработчика) в таблице (это техника, благополучно известная программистам ещё со времён MS-DOS). 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, используемого прикладными программами пакета;
- Так делают программы-вирусы или недоброжелательные программы, пытающиеся перехватить контроль над компьютером;

До определённого времени (до версии 2.6) ядро экспортировало адрес таблицы переходов системных вызовов `sys_call_table[]`. На сейчас, этот символ может присутствовать в таблице имён ядра (`/proc/kallsyms`), но не экспортируется для использования модулями (нужен тип T):

```
$ 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`, находящийся в начальных позициях таблицы `sys_call_table[]`:

```
$ cat /proc/kallsyms | grep sys_close
c047047a T sys_close
```

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). Есть положительный опыт, полученный с использованием, например, динамично развивающихся проектов виртуальных машин QEMU (свободных проект http://wiki.qemu.org) и VirtualBox (основанный на QEMU проект от Sun Microsystems, ныне от Oracle). Отладка в среде виртуальной машины (с учётом минусов, привносимых всяким моделированием) создаёт целый ряд дополнительных преимуществ, по сравнению, например, с отработкой проектов пространства пользователя:

- 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.

Отдельные отладочные приёмы и трюки

Здесь мы перечислим некоторые 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 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 и не навредит) становится интересным средством отладки, особенно на начальных этапах отработки, когда можно проверить все инициализированные значения модуля.

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
```

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 аномалии в поведении системы. Особенно часто это наблюдается, например, при отработке интерфейсов драйвера в файловую систему `/proc`.

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.

Очень многое из того, что полезно бы для разработчика драйверов для Linux - не попало в предмет нашего рассмотрения. Но всякое изложение нужно на какой-то момент завершать, в том текущем виде, какой оно имеет на сегодня (хотя бы потому, что по поводу некоторых вещей, которые хотелось бы обсудить, у меня нет примеров программных кодов, подтверждающих своим выполнением сказанное). Будем надеяться, что эти дополнительные стороны программирования модулей ядра Linux мне удастся дополнить в будущих редакциях текста.

Приложения

Приложение А : 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, 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, например с расширенными отладочными функциями.

Выбор ядра

Берём архив исходных кодов ядра из официального источника: http://www.kernel.org/ . Помещаем архив в `/usr/src` и 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 ссылку (как это и рекомендуют) на каталог рабочих исходных кодов, при изменении версии ядра мы сможем только переставлять ссылку:

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

- это объём исходных кодов ядра, мы к нему ещё вернёмся...

Конфигурация

Теперь нам предстоит провести конфигурирование ядра, что должно закончиться созданием файла `./confugure` в каталоге исходных кодов. Хорошей идеей будет использовать в качестве начального приближения тот файл `./confugure`, 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) и файл может быть скопирован под именем `./confugure` 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:

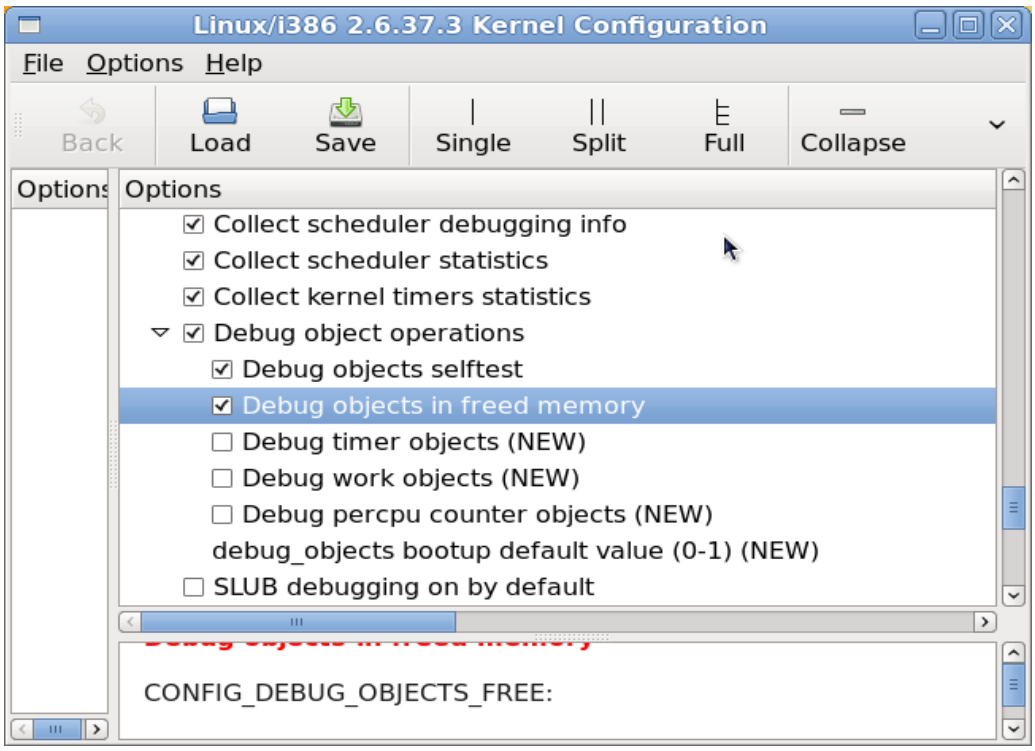
```
$ make gconfig
  HOSTCC  scripts/kconfig/gconf.o
...
```

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
```



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.

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 репозиториями, указанными 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.

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»:

http://dmilvdv.narod.ru/Translate/LDD3/index.html?linux_device_drivers.html

[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