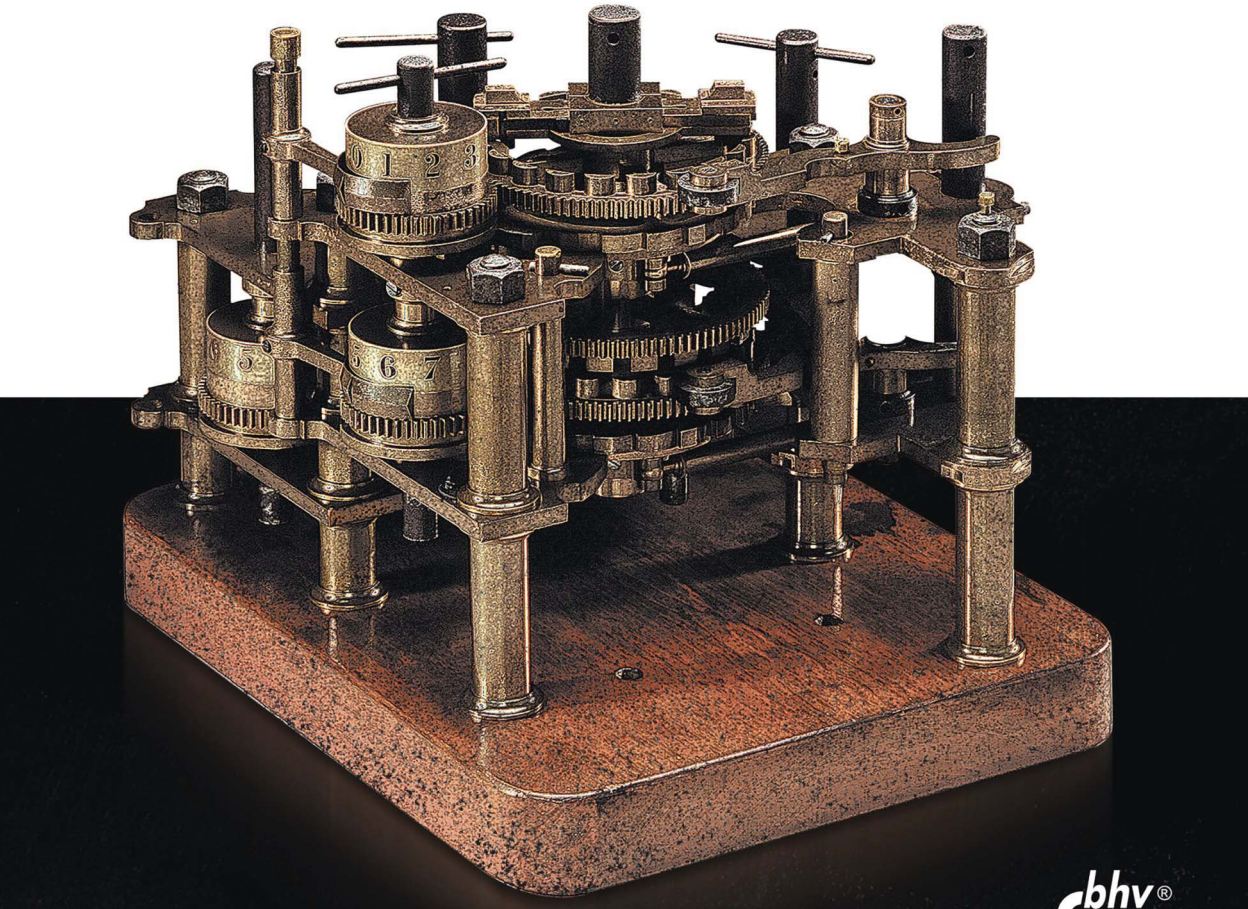
Олег Цилюрик

РАСШИРЕНИЯ ЯДРА LINUX: ДРАЙВЕРЫ И МОДУЛИ



bhv®

Олег Цилюрик

РАСШИРЕНИЯ ЯДРА LINUX: ДРАЙВЕРЫ И МОДУЛИ

Санкт-Петербург
«БХВ-Петербург»
2023

УДК 004.4
ББК 32.973.26-018.2
Ц60

Цилюрик О. И.

Ц60 Расширения ядра Linux: драйверы и модули. — СПб.: БХВ-Петербург, 2023. — 688 с.: ил.

ISBN 978-5-9775-1719-5

В книге подробно рассмотрено программирование драйверов ядра Linux, исследованы возможности расширяемости ядра при помощи модулей. Основная версия ядра — 5.15. Код примеров отработан и проверен на десятках различных инсталляций Linux, установленных из различных дистрибутивов и разных семейств дистрибутивов: Fedora, CentOS, Debian, Ubuntu, Mint. Уделено внимание архитектурам x_64, x_86, ARM, а также одноплатному компьютеру Raspberry Pi и драйверам устройств, подключаемых по USB. Затронут стандарт POSIX, разобраны API ядра, работа с Raspberry Pi, системные вызовы и подключение разнообразных периферийных устройств.

Для программистов и системных администраторов

УДК 004.4
ББК 32.973.26-018.2

Группа подготовки издания:

Руководитель проекта	*Олег Сивченко*
Зав. редакцией	*Людмила Гауль*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Зои Канторович*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

ISBN 978-5-9775-1719-5

Оглавление

Памяти моих родителей, благодаря которым я усвоил,
что одним из немногих достойных занятий,
которым посвящают себя люди,
есть написание книг

Когда меня не станет, я буду петь голосами
Моих детей и голосами их детей.
Нас просто меняют местами,
Таков закон Сансары, круговорот людей.

Баста (Василий Вакуленко), «Сансара»

От автора

Ómne tulít punctúm, qui míscuit útile dúlci,
Léctorém deléctandó paritérque monéndo.
(Всех соберет голоса, кто смешает приятное с пользой,
И услаждая людей, и на истинный путь наставляя.)
Гораций, «Ars Poetica»

Предыстория

Исходный прототип этой книги в очень архаичном виде появился как итог подготовки и проведения курса тренингов, которые мне предложила организовать компания Global Logic (**http://www.globallogic.com/**) для сотрудников своих украинских отделений (в разных городах Украины в филиалах компании их на то время работало свыше двух тысяч человек). Ровно перед этим я как раз закончил участие в очень крупном проекте в области IP-телефонии (да и просто цифровой телефонии вообще), затянувшееся на более чем пять лет и связанное как раз с разработкой драйверов для проприетарного оборудования и его окружения. У меня не было, конечно, и мысли создавать какой-то развернутый текст описания к черновикам кода, которые скопились за эти годы. А тут предлагалось обобщить практический опыт такой разработки и на его основе создать вводный курс для программистов компании... Сказано — сделано!

Первоначальный курс, начитанный в тренинговом[1] цикле весны-лета 2011 года в Харькове, и составил основную часть этого текста. Второй «подход» к курсу состоялся в тренинговом курсе лета-осени 2012 года, проводимом для Львовского отделения компании, специализирующегося больше на разработках для встраиваемого оборудования, — это потребовало существенной детализации материала. К завершению подготовки курса стало ясно, что большую проделанную работу, главным образом по написанию и отладке *примеров кода*, жаль терять бессмысленно, расценивая ее только как иллюстративный материал к тренингам. Более того, несколько моих коллег прошлых лет на протяжении этой работы обращались ко мне с просьбой переслать им материал в том «сыром» виде, как он находился

[1] Так процесс обучения принято называть в компании, поэтому я придерживаюсь принятой в ней терминологии... Хотя я, скорее, называл бы это практическим семинаром, когда профессиональные специалисты, каждый из которых обладает изрядным опытом завершенных проектов, обмениваются мнениями и формируют единую точку зрения на состояние вещей.

в процессе создания (as is), и утверждали позже, что он им заметно помог. Все это подвигло меня на намерение довести лекционный материал до вида читаемого издания. Исходный текст был значительно дополнен и переработан, итогом чего и стала базовая часть этой книги. Следующая серьезная ревизия ее произошла после марта-апреля 2014 года, когда организовывался новый цикл обучения сотрудников Global Logic, что потребовало и нового существенного пересмотра всего материала.

Последующее обновление всех приведенных в нем кодов делалось в 2015–2016 годах в качестве ревизии под изменившиеся версии ядра (примерно 3.16). К этому времени текст широко разошелся по Интернету и стал известен как «Конспект...». Последняя ревизия текста и кодов — радикальная — относится к 2022 году и связана снова с «погоней» за изменениями в API ядра. Она и привела меня к осознанию того, что нужно демонстрировать не конкретные реализации кода под последние версии ядра, а показывать идеологию на уровне понимания и то, как искать и модифицировать конкретику реализации под постоянно «убегающие» изменения в ядре. О чем будет еще не один раз сказано подробно далее...

Вся эта предыстория протяженностью больше 10 лет рассказана с той единственной целью, чтобы сразу создать понимание — насколько это возможно — того, что есть в этой книге и чего в ней не может быть...

Литература по программированию модулей ядра Linux хотя и малочисленна, но есть. В конце книги приведено достаточно много обстоятельных источников информации по этому предмету. Они весьма хороши, а отдельные из них — так и просто замечательные... Но актуальность (по моему мнению) дополнительной систематизации информации, попытка которой вылилась в создание этой книги, на момент ее подготовки стимулируется еще и некоторыми привходящими обстоятельствами:

- всплеском интереса к операционным системам, базирующимся на ядре Linux, для самых различных классов мобильных устройств. Примером того может служить в высшей степени динамично развивающаяся и массово используемая система Android. В глубине Android работает то же ядро Linux. И 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 специфические модели оборудования заметно (или сознательно?) вуалируются (лицензия GPL обязывает, но разработчики не особенно с этим торопятся...);
- тенденцией роста числа процессоров в единице оборудования — на сегодня уже не являются экзотикой компьютеры SMP[2] с 4–10 ядрами. Плюс каждое ядро может быть дополнено гипертрейдингом. Но и это только начало: большую активность приобрел поиск технологий параллельного использования десятков, сотен, а то и тысяч параллельно работающих процессоров — в этом плане обратим

[2] SMP (symmetric multiprocessing) — симметричная многопроцессорная архитектура. Главной особенностью систем с архитектурой SMP является наличие общей физической памяти, разделяемой всеми процессорами.

внимание на модель программирования CUDA[3] от компании Nvidia. Все эти архитектуры используются эффективно только в том случае, если параллелизм адекватно поддерживается со стороны ядра;

- очень серьезное расширение (в последние 10–15 лет) аппаратных платформ, *на практике* используемых IT-рынком. Если еще 15 лет назад многочисленные процессорные платформы: SPARC, ARM, MIPS, PPC и пр., существовали как маргинальные линии развития в стороне от доминирующей линии *x*86 от Intel, то на сегодня картина разительно изменилась. И все эти архитектуры поддерживаются единой операционной системой Linux (это вам не дружный альянс Intel & Microsoft). И более того, на многих аппаратных платформах, «покувыркавшись» со своими оригинальными проприетарными системами программного обеспечения, начинают рассматривать Linux как *основную* программную платформу.

Эти наметившиеся тенденции, если и не подвигают к немедленному написанию собственных компонентов ядра (что, пожалуй, и совершенно не обязательно), то, по крайней мере, подталкивают интерес к более точному пониманию процессов, которые происходят в ядре, и использованию их возможностей в прикладных проектах.

Кому адресована книга?

Хорошая книга не дарит тебе откровение,
хорошая книга укрепляет тебя в твоих самостоятельных догадках.
Андрей Рубанов, «Хлорофилия»

В занятиях «копаться» с ядром операционной системы (и Linux здесь не исключение) могут быть, собственно, две основные категории интереса, одинаково похвальные, но разительно отличающиеся мотивацией:

- желание глубоко *разобраться* с механизмами, оперирующими внутри ядра. Часто такой интерес подталкивается намерением писать код для каких-то компонентов ядра и, возможно, желанием войти в команду, работающую над совершенствованием и развитием кода ядра;
- намерение (а часто и безальтернативная необходимость в силу служебных обязанностей) *использовать* механизмы ядра в своих собственных проектах прикладного характера — потребность совершенно утилитарного свойства.

Для потенциальных читателей первой категории эта книга практически бесполезна — я не располагал никакой инсайдерской информацией относительно механизмов ядра и не состоял в команде разработчиков. В материалах книги использованы только информация из публичных общедоступных источников и собственный экс-

[3] CUDA (аббревиатура от Compute Unified Device Architecture) — параллельная вычислительная платформа и модель интерфейса прикладного программирования (API), созданная компанией Nvidia.

периментальный опыт разработки в реальных проектах, опыт, который растянулся на десятилетие... Я не знаю и ничего не могу сказать о тех многих других механизмах ядра, которые выходили за круг моих непосредственных служебных интересов. Для тех, кто относит себя к первой группе специалистов, существует определенное число (хотя и весьма ограниченное) изданий (англоязычных, естественно), со знанием дела излагающих предмет. Значительная часть из них приведена в списке литературы в конце книги.

Эта книга рассчитана на достаточно *опытных* разработчиков системного программного обеспечения. Предполагается, что, возможно, у читателя и отсутствует какой-либо опыт в программировании именно для ядра Linux или даже вообще в программировании для этой системы — но предполагаются знания и какой бы то ни было опыт в системном программировании для других операционных систем, которые станут хорошей базой для построения аналогий. В высшей степени плодотворно любое знакомство с одной или несколькими POSIX-системами: Open Solaris, QNX, FreeBSD, NetBSD, MINIX3, ... — с любой из них в равной степени.

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

Естественно, я предполагаю, что вы «на дружеской ноге» с UNIX/POSIX консольными утилитами — такими как: `ls`, `rm`, `grep`, `tar` и другими. В Linux используются наибольшим образом GNU (FSF) реализации этих утилит, которые набором опций часто отличаются (чаще в сторону расширения) от непосредственных предписаний стандарта POSIX, а также, порой, и от своих собратьев в других операционных системах (Solaris, QNX, ...). Но эти различия не столь значительны, я думаю, чтобы вызвать у вас какие-либо затруднения.

Некоторые главы книги предполагают совсем начальный уровень знакомства (или не знакомства) читателей с инструментарием, традициями и привычками Linux — в большинстве случаев такие вводные обзоры включены по прямым пожеланиям читателей рукописи. Более умудренные знатоки могут их безболезненно пропускать, но выбросить их, по пожеланиям «экспертов» и в ущерб нуждающимся в них, я не счел уместным.

Структура книги

Никто не обнимет необъятного!
Козьма Прутков, «Плоды раздумья», 1854, 67-й афоризм

Я предполагаю, что вы вряд ли ставите своей целью писать код непосредственно для ядра Linux. В таком случае вам предстоит совсем другой предпочтительный «путь джедая» — установить контакт непосредственно с достаточно многочисленной на сегодня командой разработчиков ядра, формально возглавляемой Линусом Торвальдсом, работающей по строгим регламентам и распределению ролей, влиться в эту команду и найти в ней свое место. Там есть свои правила организации работы...

Я полагаю, что техника написания модулей ядра вам нужна в *крупных реальных прикладных проектах* — как средство некоторой модификации функциональности ядра под потребности своего проекта. Начиная с потребности написания *драйверов* под некоторое специфическое оборудование или протоколы, использующиеся в проектах. Потому что термины «модуль ядра» и «драйвер» — это в некотором смысле синонимы... что будет рассмотрено позже. Но модули ядра — это не только драйверы, их функциональность может быть гораздо шире.

Я не только так полагаю, но и использовал такое свое понимание в практической разработке в проектах на протяжении ряда лет. А поэтому оно и определяет направленность книги. И эта нацеленность на *написание* модулей ядра дополняется по тексту только *минимальным* описанием внутренних механизмов и API ядра, которые необходимы для такой работы (что не является предметом моей заинтересованности). В компенсацию этого минимума текст дополнен довольно пространными *функциональными* (по принципу действия) описаниями того, как прикладное программное обеспечение взаимодействует с ядром, — что, собственно, не относится к темам ядра, но крайне необходимо в прикладных разработках.

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

Все, что описывается далее, — это разные стороны техники написания модулей ядра: управление памятью, взаимопонимание со службой времени, обработка прерываний, программные интерфейсы символьных и сетевых устройств... и многое другое. Но укрупненно эти вопросы отсортированы примерно по следующему порядку, несколько отличающемуся от часто принятого:

- беглый обзор базовых понятий и возможностей, как уже было сказано;
- вопросы архитектуры операционной системы;
- краткий обзор доступных инструментов программирования;

- обзор API и механизмов, представляющих интерфейсы модуля *во внешний мир* — в пространство пользователя;
- обзор API и механизмов, реализующих «внутреннюю механику ядра» — это инструментарий программиста для *внутренней реализации* алгоритмики модуля;
- отдельные тонкие вопросы и возможности (и даже трюки), не так часто требующиеся при отработке непосредственно драйверов, но представляющие большой интерес в технике модулей ядра вообще.

Текст этой книги ориентирован в первую очередь не столько на чтение или подробное изучение, сколько на использование в качестве отправного справочника при последующей работе по программированию в затронутой области. Это накладывает отпечаток на текст (и обязательства на автора):

- перечисление *альтернатив* — например, символьных констант для выражения значений некоторого параметра — в случае их многочисленности приводится *не полностью*. Разъясняются только наиболее употребимые, акцент делается на понимание (все остальное может быть найдено в заголовочных файлах ядра) — вместо путанного перечисления 30 альтернатив лучше указать две, использующиеся в 95% случаев;
- обязательно указываются те места для поиска (имена заголовочных файлов, файлы описаний) где можно (попытаться) разыскать ту же информацию, но актуальную в требуемой вами версии ядра. Это связано с постоянными *изменениями*, происходящими от версии к версии;
- подготовлено максимальное (в меру моих сил возможное) число *примеров*, иллюстрирующих изложение. Это примеры не гипотетические, описывающие фрагментарно, как должно быть, — все примеры законченные, исполнимые и проверенные. Они оформлены как мини-проекты, которые собираются достаточно тривиально, причем многие проекты содержат файл *.hist — протокол с терминала тех действий, которые выполнялись по созданию, сборке и тестированию проекта. Это многое разъясняет: зачем сделан пример и что он должен показать;
- с другой стороны, все примеры максимально упрощены и из них исключено все лишнее, что могло бы усложнять их прозрачность. Хотелось бы предполагать, что такие примеры могут быть не только иллюстрацией, но и *стартовыми шаблонами*, от которых можно оттолкнуться при написании своего кода подобного же целевого предназначения: берем такой шаблон и начинаем редактированием наполнять его спецификой разрабатываемого проекта...;
- именно с этой целью (возможность последующего справочного использования) *в тексте* показаны протоколы того, как выполняются команды: 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 разработ-

чиков программных проектов общего назначения. Максимально сокращены и те разделы, по которым трудно (или мне не удалось) показать действующие характерные программные *примеры* минимально разумного объема. Рассказа на качественном уровне — «на пальцах» — я старался всячески избегать: те аспекты, которые я не могу показать в коде, я просто безмолвно опускаю (возможно, в более поздних редакциях книги они найдут свое место).

Во многих местах по тексту разбросаны врезки, предваряемые словом **Примечание** или **Пояснение**. Иногда это одна фраза, иногда полстраницы и более... Они содержат необязательные уточняющие детали, обсуждение (или, точнее, указание) *непонятных мне* на сегодня деталей, а где-то они представляют собой «лирическое отступление» в сторону от основной линии изложения, но навеянное попутно... Отдельные моменты в обсуждениях будут приводиться в форме «вопрос-ответ» (FAQ) — это те вопросы, которые *реально* неоднократно задавали участники семинаров и читатели рукописи. Если кого-то такие отступления станут утомлять — их можно безболезненно опускать безо всякого ущерба для дальнейшего чтения.

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

- краткая справка по утилите `make`, с которой приходится работать постоянно (*приложение 1*), — все, что там сказано, большинству хорошо известно, но в качестве памятки не помешает, — там есть специфические детали, полезные именно при работе с модулями ядра;
- исходный код *тестов* (примеры кода) динамического выделения памяти в ядре, обсуждение и результаты этих тестов. Я вынес все это из основного текста книги в отдельное *приложение 2* из-за объемности и детальности материала;
- описание *нескольких* способов записи в аппаратно защищенные страницы памяти архитектуры *x*86 (*приложение 3*) — это необходимо для многих тонких возможностей работы с таблицей системных вызовов ядра.

В любом случае, как уже было сказано, текст этой книги — производное от первоначального конспекта тренингового курса, проведенного с совершенно прагматическими намерениями для контингента профессиональных разработчиков программного обеспечения, в котором я только систематизировал и обобщал свой опыт предыдущих лет. Таким конспектом (для себя в первую очередь) он и остается — никаких излишних подробных разъяснений, обсуждений... Хотелось бы надеяться, что он совместно с прилагаемыми примерами кода может стать отправной точкой, шаблоном, оттолкнувшись от которого вы сможете продолжать развитие реального проекта, затрагивающего область ядра Linux.

При подготовке книги я не располагал никакой инсайдерской информацией и не контактировал напрямую с командой разработчиков ядра Linux, поэтому вся излагаемая в ней информация получена из открытых публикуемых источников или из целенаправленных экспериментов. Совершенно не исключено, что некоторые выводы из таких экспериментов интерпретированы мной неверно или неполно... Но основной принцип, предлагаемый книгой для ее читателей: ищите и вы найдете

всю недостающую вам информацию по ядру Linux. Моя же цель — только подсказать, где и как ее искать.

В конечном счете эта книга — отображение моих собственных мытарств в поисках ответов на вопросы, которые вставали передо мной по ходу работы в крупном проекте на протяжении нескольких лет, проб и ошибок. И описываются в ней те нюансы, которые возникали в процессе работ, и то, как они разрешались. Возможно, у вас возникнут совсем другие вопросы, которые меня не занимали. Но я стараюсь максимально указывать на источники и алгоритмы поиска ответов на возникающие вопросы.

Текст книги насыщен и даже *перенасыщен* сверх необходимого минимума примерами консольных команд (CLI) и листингами их выполнения. Это сделано сознательно и связано с тем, чтобы максимально показать, *как* мы добываем нужную информацию из системы и *где* мы ее добываем. По ходу изложения будет неоднократно отмечено, что совместимость API и механизмов ядра от версии к версии — низкая («никакая»). Поэтому невозможно показать и научить, *как* реализовать тот или иной модуль ядра (его код устареет уже ко времени издания книги). Можно только показать, *где* мы находим информацию, подсказывающую, как реализовать возможность.

Еще один мотив такой перенасыщенности командами — эти многочисленные команды являются инструментарием экспериментирования со множеством разнообразных и замысловатых опций и параметров. И их текст может явиться для вас (и для меня самого) *справочником* для последующих экспериментов.

То же самое относится и архитектурным различиям систем Linux — в примерах команд будут, чередуясь, показываться примеры из 32- и 64-битных систем, из Intel *x*86 и ARM... Это делается сознательно — нужно видеть, как выглядят эти различия, и то общее, что скрывается за ними.

Соглашения, принятые в тексте

Для ясности чтения текста он — по функциональному назначению его фрагментов — размечен различными шрифтами. В принципе, такая или подобная разметка в публикациях употребляется практически повсеместно. Итак, для выделения фрагментов текста по назначению используется следующая разметка:

- некоторые ключевые понятия и термины в тексте, на которые нужно обратить особое внимание, будут выделены *курсивом*;
- `моноширинным шрифтом` (прямо в тексте) будет выделяться написание имен команд и программ — всех тех терминов, которые должны оставаться неизменяемыми, например: `/proc`, `mkdir`, `./myprog` и т. п.;
- ввод пользователем в консольных командах (сами выполняемые команды или ответы в диалоге), кроме того, могут быть выделены **`полужирным моноширинным шрифтом`**, чтобы отличать их от ответного вывода системы в диалогах;

- имена файлов и путей к ним в тексте будут выделены рубленым шрифтом типа Arial;
- имена файлов программных листингов (как они подаются в архиве примеров) оформлены специальной шапкой — например, так:

hello_printk.c

- в некоторых примерах команды работы с модулем будут записываться так:

```
# insmod module.ko
```

в других — вот так:

```
$ sudo insmod module.ko
```

что будет означать то же самое: символ `#` предваряет команду с правами root, а символ `$` — команду в правами ординарного пользователя.

Причем в ряде случаев это будет записано так (что, опять же, то же самое):

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

Пояснение

Последний случай обусловлен тем, что примеры проверялись на нескольких различных инсталляциях и дистрибутивах Linux — для большей адекватности и в силу продолжительности написания, что было под рукой, на том и проверялось). При этом обнаружилось, что в некоторых инсталляциях каталог `/sbin` не входит в переменную `$PATH` для ординарного пользователя, а править скопированный с терминала протокол выполнения команд произвольно (как «должно быть», а не «как есть») я не хотел во избежание непроизвольного внесения несоответствий.

Код примеров и замеченные опечатки

Книги — только одно из вместилищ,
где мы храним то, что боимся забыть.
Рэй Брэдбери, «451 градус по Фаренгейту»

Те, кто читал предварительные редакции текста, временами спрашивали: «Зачем так много детальных команд, протоколов выполнения команд и даже альтернативных вариантов команд, которые выполняют если и не эквивалентные, то весьма близкие действия?». Ответ простой: потому что этот текст делался (годами) как справочное пособие, напоминание (часто самому себе) по выполнению самых разнообразных действий «вблизи ядра». Для меня это — памятка на каждый день, а для вас — самоучитель игры с ядром.

Но самое ценное, что прилагается к этой книге, — это архив примеров кода. Он представляет собой результат не одного года экспериментов автора и его накапливающийся справочник, памятка заготовок (шаблонов) для дальнейших конкретных реализаций. Все остальное — сама книга — это, собственно, пространный комментарий к архиву кодов.

Все листинги, приводимые в качестве примеров, были опробованы и испытаны. Более того, они перепроверяются из года в год, с изменениями в версиях ядра, и при необходимости по возможности корректируются (об этом нужно и будет сказано далее). Архив организован в виде тематического дерева, на верхнем уровне соответствующего структуре разделов книги. В тексте, где обсуждаются коды примеров, везде по возможности будет приведено в скобках имя архива в этом источнике — например: архив tools/export.tgz, или это может быть указано просто как каталог tools). В зависимости от того, в каком виде (свернутом или развернутом) вам достались файлы примеров, то, что названо по тексту «архив», может быть представлено на самом деле деревом каталогов, содержащих файлы этого примера, поэтому относительно «целеуказания» примеров термины *архив* и *каталог* (*папка*) 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 имен объектных файлов, и т. д. Листинги поэтому специально не нумерованы (нумерация могла бы «сползти» при многолетних правках), но указаны каталоги по имени, где их можно найти. В архивах примеров *могут* содержаться (в большинстве) файлы вида *.hist (расширение от history) — текстовые файлы протоколов выполнения примеров: порядок запуска приложений, каких результатов следует ожидать и на что обратить внимание. В тех случаях, когда сборка (`make`) примеров требует каких-то специальных приемов, протокол сборки также может быть показан в этом файле. Файлы history накапливались как совершенно черновой и сырой материал, они не подлежат никакой последующей правке, да и разобраться в них трудно, но они могут дать краткие подсказки о том, как поступать с соответствующим им примером.

Очень серьезную проблему при создании и работе с архивом, как стало понятно со временем (а работа с примерами кода продолжается больше 10 лет), составляет *вариабельность ядра* от версии к версии: то, что вчера замечательно выполнялось в точности, как и замышлялось, — сегодня перестает просто даже компилироваться. Но и тупо вносить изменения в код простым редактированием тоже не выход — код работоспособен в последних версиях ядра, но перестает компилироваться в предыдущих. А в обиходе для разных областей применения находится широчайший спектр версий ядра одновременно: от 2.6.32 и до 5.18... В этом есть проблема! И в итоге *опробования* в работе разных вариантов принято решение:

- в каждом каталоге архива *может* (в тех каталогах, где это нужно) находиться подкаталог old_vers (иногда ?old_vers — знак ? включен в имя с тем, чтобы это имя всегда находилось «выше» других, бросалось в глаза и не «путалось под ногами» при работе со значащими подкаталогами);
- в этом подкаталоге может содержаться один или несколько подкаталогов с *числовым* именем, соответствующим версии ядра, — это *тот же* проект, но в том неизменном виде (без всяких правок), в котором он успешно использовался в указанной версии ядра (и, как правило, в других версиях, ниже указанной);
- в самом подкаталоге old_vers находится файл Makefile, который в процессе рекурсивной сборки всего дерева (о чем будет рассказано отдельно) не собирает вло-

женные подкаталоги версий (это привело бы к ошибкам сборки), а только может очищать их;

- для сборки в такие подкаталоги версий нужно зайти внутрь и уже там выполнить `make`.

  Например, в каталоге network:

```
$ ls -l ?old_vers/
итого 12
drwxrwxr-x 7 olej olej 4096 июн 18 01:45 3.14
drwxrwxr-x 7 olej olej 4096 июн 18 01:46 3.17
-rw-r--r-- 1 olej olej  342 апр 25  2014 Makefile
```

  Здесь каталоги 3.14 и 3.17 будут содержать такие же подкаталоги, что и текущий (последний) каталог network: net, netproto, virt и другие, но относящиеся к означенной версии *ядра*. Такую структуру можно было бы, наверное, успешно создать с помощью GIT, но «ветви» (branch) в нашем случае относятся не к версиям самого проекта, а к версиям сущности, внешней по отношению к самому проекту — ядру.

Основа некоторых обсуждаемых примеров заимствована из публикаций (приведенных в конце книги в *разд. «Источники информации»*) — в таких случаях я старался везде указать источник заимствования и сохранить авторские комментарии. Некоторые примеры возникли в обсуждениях с коллегами по работе — часть примеров реализована ими или совместно с ними в качестве идеи теста. Во всех случаях я старался сохранить отображение первоначального авторства в кодах (в комментариях, в авторских макросах).

И еще одно немаловажное замечание относительно программных кодов. Я отлаживал и проверял все примеры на не менее чем двух десятках различных инсталляций Linux (реальных и виртуальных, 32- и 64-разрядных архитектурах, *x*86 и ARM, установленных из различных дистрибутивов и разных семейств дистрибутивов: Fedora, CentOS, Debian, Ubuntu, Mint...). И в некоторых случаях *давно работающий* пример вдруг переставал компилироваться с совершенно дикими сообщениями вида:

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

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

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

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

В конечной фазе подготовки книги стало понятно, что архив кодов нужно представить параллельно в двух эквивалентных формах: иерархия по главам книги (каталог kernel.modules_by_chapters) и иерархия по темам обсуждения (каталог kernel.modules_by_topics). Это обусловлено как накопившимся объемом архивов, так и возможным способом работы с ним — при работе с текстом или при изучении конкретного применения. Архив кодов как самая важная часть работы выложен на *нескольких общедоступных* ресурсах. В частности, отправной точкой в поиске ссылок на архив остается форум: **https://linux-ru.ru/viewtopic.php?f=18&t=7075&p=27834#p27834**. Кроме того, архив кодов размещен на FTP-сервере издательства «БХВ», и скачать его оттуда можно по ссылке **https://zip.bhv.ru/9785977517195.zip**. Эта ссылка доступна и со страницы книги на сайте **https://bhv.ru/**.

Конечно, даже при самой тщательной выверке и вычитке в таком объемном тексте не исключены недосмотры и опечатки, могут также проскочить мало внятные стилистические обороты и подобное. Обо всех замеченных такого рода дефектах я прошу сообщать по электронной почте **o.tsiliuric@yandex.ru**, и я был бы признателен за указание на любые недостатки книги, замеченные ошибки или за высказанные пожелания по ее последующей доработке.

Замечания о версиях ядра

Этот текст в нынешнем его виде накапливался, подготавливался и писался, изменялся и дополнялся на протяжении 11 лет. А кроме того, использовались также заделы и следы предыдущих реальных проектов, в которых участвовал автор, еще как минимум пяти предшествующих лет, выполнявшиеся в дистрибутиве CentOS 5.2:

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

Все это время к тексту отбирались, писались, добавлялись, отрабатывались, выверялись и совершенствовались примеры реализации кода. За все это время «в обиходе» сменились версии ядра от 2.6.32 до 5.15 (на момент последней редактуры). Примеры и команды, показанные в тексте, отрабатывались и проверялись на всем этом диапазоне версий и в дополнение на нескольких различных дистрибутивах Linux: Fedora, CentOS, Debian, Ubuntu, Armbian, Raspberry Pi OS...

Кроме того, дополнительное радикальное разнообразие вносит и то, что примеры отрабатывались на 32- и 64-битных инсталляциях, в архитектуре *x*86/*x*86_64 и в архитектуре ARM, на реальных и нескольких виртуальных машинах под управлением Oracle VirtualBox. Как легко видеть, для проверок и сравнений были использованы варианты по возможности широкого спектра различий. Хотя выверить *все* примеры и на *всех* вариантах установки — это за гранью человеческих возможностей.

Как оказалось, *самая сложная* из всех решавшихся задача — поддерживать адекватное (работоспособное) состояние кодов всех примеров относительно вариабельности kernel API (о версионности ядра мы еще будем подробно говорить позже и не раз). Первые 3–4 года от начала работы над материалом я пытался участвовать в «гонках за версиями ядра» — к каждой очередной редактуре текста *приводить* все примеры к более или менее последней версии ядра на это время.

Выполняя ревизию текста и кода на соответствие, особенно к версиям ядра 5.*x*, я задавался вопросом: а нужно ли *сохранять* реализации, сделанные под предыдущие версии — 2.6.*x*, 3.*x*, 4.*x*? Поскольку вариант кода, адаптированный к версии 5.*x*, *не будет* собираться под множество предыдущих версий, а там, где собирается, *не будет* работоспособным! Разработчики ядра нисколько не «заморачиваются» совместимостью kernel API снизу вверх.

И, наблюдая это разнообразие, я приходил к решению, и, более того, это же вытекало из обсуждений с коллегами и переписки с читателями предварительной рукописи: *нужно* сохранять предыдущие реализации одновременно с обновленными. Потому что это важно для *встраиваемых* (embedded) и промышленных реализаций, где все еще широко применяются прошивки прежних версий (например, OpenWRT). Это важно также и для многих *промышленных* проектов, где в качестве базовых часто не гонятся за последними версиями ядра, предпочитая проверенную надежность новизне. И это важно для тех, кто работает с отдельными дистрибутивными проектами, не попадающими в «мейнстрим», но от того не менее интересными: AntiX, GalliumOS, NuTyX, PuppyLinux и др. Ну а еще это становится особенно актуальным — в свете последних событий — для дистрибутивов русскоязычной сборки: Astra Linux, ALT Linux, ROSA и др., которые «не поспевают» за самыми свежими версиями ядра (да это и не нужно). И продолжаться эта тенденция будет, видимо, долгие годы...

Наконец, текст рукописи и объем кодов рос, и в какой-то момент времени приводить *весь* этот объем к состоянию, адекватному свежей версии ядра, стало по трудоемкости выше любых человеческих сил. Поэтому было принято решение, и вы его увидите и в тексте, и в кодах, что:

- по самым актуальным подсистемам, наиболее часто используемым (примеры тому драйверы символьных устройств, система procfs, сетевая подсистема и т. п.), обновление кодов производится до самых последних версий ядра;
- по реже используемым подсистемам (драйверы блочных устройств, неблокирующий ввод/вывод и мультиплексирование) примеры описываются относительно той версии, до которой они еще обновлялись, — идеологически этого достаточно, работа их не меняется, а доведение до свежих версий оставлено на самостоятельную проработку;

Независимо от того или иного решения, *во всех случаях* в архиве кодов сохраняются все реализации, успешно использовавшиеся в предыдущих версиях ядра.

Обновляемость ядра

К версии (ядра) нужно подходить с очень большой осторожностью: ядро — это не пользовательский уровень, и разработчики ядра совершенно не обременяют себя ограничениями совместимости снизу вверх (в отличие от пользовательских API и POSIX-стандартов). Источники информации и обсуждения, в множестве разбросанные по Интернету, чаще всего касаются устаревших версий ядра (а временами и древних) и абсолютно не соответствуют текущему положению дел. Очень показа-

тельно это проявилось, например, в отношении макросов подсчета ссылок использования модулей, которые только до версий 2.4.*x* использовались: `MOD_INC_USE_COUNT` и `MOD_DEC_USE_COUNT`, но их нет, начиная с 2.6.*x* и далее, а они продолжают фигурировать во множестве описаний в Интернете.

Время от времени (в разные годы) Linux Foundation публикует очеты о ходе развития ядра Linux. Вот публикуемый для примера короткий фрагмент хронологии (за 2015 г.) выходов нескольких последовательных версий ядра (рис. О.1).

Kernel Release	Version Date	Days of development
3.11	2013-09-02	64
3.12	2013-11-03	62
3.13	2014-01-19	77
3.14	2014-03-30	70
3.15	2014-06-08	70
3.16	2014-08-03	56
3.17	2014-10-05	63
3.18	2014-12-07	63

Рис. О.1. Фрагмент хронологии (за 2015 г.) выходов нескольких последовательных версий ядра (в последней колонке указано число дней от предыдущей версии до текущей)

И оттуда же:

- объем работы в последнее время вырос: каждая новая версия ядра включает в себя более 10 000 патчей от 1400+ разработчиков;
- количество файлов и строк кода постоянно растет. В момент первоначального релиза в 1991 году ядро содержало около 10 000 строк, а сейчас — 18 997 848. За полтора года база разбухла примерно на 2 млн строк.

Среднее время до выхода очередного ядра на протяжении нескольких последних лет (2005–2012) составляло 80 дней, или около 12 недель (взято там же). Нет оснований полагать, что этот темп снизится в дальнейшем. А разработчики ядра, как отмечалось ранее, не особенно обременяют себя требованиями совместимости снизу вверх. Поэтому единственный действенный способ преуспеть в написании собственных модулей ядра — не накопленные знания и объяснения (в том числе и настоящая книга!), а знание, *как и где найти* актуальную информацию по *интересующей вас* версии ядра.

Уникальными ресурсами, позволяющими изучить и сравнить исходный код ядра для различных его версий и аппаратных платформ, являются ресурсы проекта **Linux Cross Reference** (известного как LXR). Таких ресурсов несколько (дублирующих, но отличающихся внешним видом и оформлением). Я приведу ссылки на некоторые (ссылки со временем могут изменяться или исчезать, ресурсы по ним отличаются использованием, поэтому полезно рассмотреть все):

- **https://elixir.bootlin.com/linux/latest/source**;
- **http://lxr.linux.no/+trees**;
- **https://lxr.missinglinkelectronics.com/linux**.

Уникальность ресурсов проекта LXR при разработке кодов модулей ядра состоит еще и в том, что при отсутствии какого-то необходимого вызова API вы можете «пройтись» по версиям ядра вниз и выяснить, до какой версии он существовал, а с какой отсутствует — поменялся на подобный или вообще исключен.

Это *основной* источник (из известных автору), позволяющий сравнивать изменения в API и реализациях от версии к версии (начиная с самых ранних версий ядра). Часто изучение элементов кода ядра по этим ресурсам гораздо продуктивнее, чем делать то же, но по исходному коду непосредственно вашей инсталлированной системы.

Использованные источники информации

Самая неоценимая помощь при подготовке первоначальной редакции от компании Global Logic, которую только можно было оказать, состояла в том, что на протяжении всей работы компания заказывала и покупала на Amazon (**http://www.amazon.com/**) подлинники всех книг, изданных в мире за последние лет 10, которые я мог найти и указать полезными. Как оказалось, таких изданий не так и много, не более полутора десятков (это если считать даже со смежными с программированием модулей вопросами). Последующие 11 лет обновления текста (когда активного, когда вялого) текст дополнялся библиографией всего, что оказалось полезным в процессе работы над ним. Самой ценной информацией в такой плохо структурированной области, как ядро Linux, является именно перечень актуальных и полезных источников. Большинство из этих источников (которые показались мне самыми полезными) приведено в конце книги в *разд. «Источники информации»*.

Это не библиография в общепринятом смысле... И не только потому, что по тексту не расставлены указатели, где и в каком источнике искать тему. В некоторых случаях это только указание выходных данных книг. Там, где существуют изданные русскоязычные их переводы, я старался указать и переводы тоже. По некоторым источникам, авторы которых решили сделать их публично доступными, показаны ссылки для скачивания их из Сети. Для статей, которые взяты из Сети, я указываю интернет-адрес (URL) и по возможности авторов публикации, но далеко не по всем материалам, разбросанным и кочующим по Интернету, удается установить авторство.

Обновления текущей редакции текста

Текст книги на протяжении ряда лет находился в постоянном развитии, расширении и обновлении. Кроме того, улучшалось его качество за счет исправления неточностей и ошибок, на которые указывали читатели рукописи (опубликованной под лицензией Creative Commons Attribution ShareAlike) в переписке с автором. Текущая редакция достаточно радикально отличается от предыдущих, циркулирующих в Интернете:

- прежде затрагивалась только архитектура Intel *x*86. Сейчас многие примеры проверяются и адаптируются к (малым) ARM-архитектурам — для тестирования

я использую одноплатные микрокомпьютеры, доступные самым широким кругам интересующейся публики: Raspberry Pi и китайское семейство Orange Pi. Использование подобных моделей открывает широкие и доступные возможности для вашего экспериментирования;

- предыдущие редакции «затачивались» преимущественно под 32-битные архитектуры Intel (начальная редакция так и исключительно). На сегодня 32-битная архитектура *x*86 не актуальна и выходит из обращения. Все изложение теперь переориентировано, где не указано явно обратное, на 64-битную архитектуру *x*86_64;
- ранние редакции были существенно ориентированы на RPM-дистрибутивы Linux (Fedora, CentOS), Debian-дистрибутивы затрагивались только отчасти. Теперь картина изменилась в точности до наоборот: RPM-дистрибутивы упоминаются только вскользь и при необходимости.

Но главное обновление — это обновление кодов архива! Оно происходит постоянно в ходе подготовки и редактуры книги и *будет* происходить уже после завершения работы над текущей редакцией текста и изданием книги. Следите за обновлениями архива кодов на тех ресурсах, где он размещен!

- Глава 1 -
Модули с высоты птичьего полёта...

Вы тоскуете о прекрасных днях Minix-1.1,
когда мужчины были мужчинами
и писали свои собственные драйверы устройств?
Линус Торвальдс

Linux и GNU

Поводом к войне послужили следующие обстоятельства. Всеми разделяется убеждение, что варёные яйца при употреблении их в пищу испокон веков разбивались с тупого конца; но дед нынешнего императора, будучи ребёнком, порезал себе палец за завтраком, разбивая яйцо означенным древним способом. Тогда император, отец ребёнка, обнародовал указ, предписывающий всем его подданным под страхом строгого наказания разбивать яйца с острого конца.
Джонатан Свифт, «Путешествия Гулливера»

В самых разных местах Интернета вы встретите утверждение, что «истинно» операционную систему следует называть Linux GNU или GNU Linux, а также бесконечные споры на этот счет. Это сильно напоминает войну между тупоконечниками и остроконечниками у Свифта, но нас такие разногласия занимать не должны, — мы оставим их знатокам Linux. Но в самом предлагаемом ими названии как раз и проявляется дуальность программного обеспечения, и без больших натяжек можно сказать так: Linux — это все, что касаемо ядра, а GNU — это та огромная часть, что относится к программированию пространства пользователя.

Ядро Linux монолитное. Среди многих операционных систем изредка можно встретить микроядерные системы, но это скорее (при их множественных преимуществах) экзотика.

На современных процессорах программный код ядра, в отличие от любого пользовательского кода, выполняется в совершенно другом режиме — это *привилегированный* режим (иногда называемый режимом *супервизора*). В этом режиме коду разрешается выполнять ряд действий, недопустимых в пользовательском режиме: обработку аппаратных прерываний, переразметку памяти, операции ввода/вывода, некоторое число привилегированных команд... Весь остальной программный код всех приложений, GNU и сторонних разработчиков, выполняется только в *пользовательском* режиме (с ограниченными *аппаратными* привилегиями — не путать с *программным* разграничением привилегий: «root и другие»).

Код привилегированного режима (ядра) выполняется в другом, отделенном адресном пространстве (виртуальном). Коды всех пользовательских процессов выполняются, в противоположность, каждый в своем *отдельном* виртуальном адресном пространстве (каждый в своем).

На самых ранних (инициативных) этапах разработки система Linux была не только монолитной, но и цельно компилируемой: все компоненты системы, драйверы и другие функциональности, должны были компилироваться в единый монолит со всем остальным кодом ядра. Такая архитектура не дает шансов любой операционной системе достигнуть уровня промышленного стандарта — это оставляет многие тысячи сторонних разработчиков (аппаратуры в первую очередь) за бортом системы. Да и вкомпилировать поддержку многих десятков, а то и сотен, тысяч альтернативных компонентов (устройств) нереально из-за размеров получаемого монолита. Поэтому с некоторого момента развития Linux была создана система модулей ядра. *Модули* — это некоторый набор функциональных компонентов ядра, на манер детского конструктора, которые альтернативно могут загружаться (или выгружаться) динамически в состав монолитного ядра в процессе загрузки или даже позже, в любой момент выполнения системы. (В некотором смысле система модулей — это своего рода подобие архитектуры приложений пользовательского пространства, называемых *оверлейными*.)

FAQ

Порой на заданные вопросы можно получить неожиданные ответы,
а кропотливая работа преподносит кое-какие сюрпризы.
Даниэла Стил, «Опасные игры», 2017

Как уже было сказано, в рамках моего курса, в его «первозданном» виде, было проведено четыре цикла тренингов программистов-разработчиков. Да и позже, за годы работы с материалом, с вопросами ко мне обращались разнообразные корреспонденты. На некоторые задаваемые вопросы ответы находились мгновенно, другие потребовали изрядно поработать над ними экспериментированием с кодом... И многие из этих вопросов повторяются из раза в раз. Поэтому, еще не приступив к предметному разговору, ответим на такие вопросы — возможно, ответы на них покажут вам, что этот предмет рассмотрения не попадает в круг ваших интересов.

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

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

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

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

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

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

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

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

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

A: Нет, не могут. Драйверы, являющиеся модулями ядра, связываются с экспортируемыми именами ядра (вызываемыми функциями API или объектами данных) по их абсолютным адресам, изменяющимися не только при изменении версии ядра, но даже при пересборке ядра с измененными конфигурационными параметрами. Поэтому драйверы Linux могут предоставляться только в виде *исходных кодов* на языке C, которые требуется *компилировать* для использования.

Модуль в иерархии программных систем

Все мы умеем писать программы для Linux и имеем более или менее приличный опыт написания таких программ, которые при всем их многообразии имеют абсолютно идентичную единую структуру[1]:

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

Такую структуру в коде будут неизменно иметь все приложения-программы, будь то тривиальная программа «Hello, world!», показанная только что, или самая навороченная среда разработки IDE или CAD. Это встречающийся в подавляющем большинстве случай: пользовательское приложение, начинающееся с некоторого `main()`, и завершающее выполнение по `exit()`. Мы говорим о языке программирова-

[1] Это относится в первую очередь к системе Linux или — шире — к любым UNIX-подобным или POSIX-системам и программному коду на языке C, который и был изначально создан для написания таких операционных систем. Но сказанное также имеет отношение и к любым пользовательскими приложениям, написанным на любом из множества языков программирования.

ния C, но примерно то же самое будет и в сотне других используемых языков программирования.

Еще один встречающийся (но гораздо реже) в UNIX случай — демоны: программы, стартующие с `main()`, но никогда не завершающие своей работы. Чаще всего они представляют собой серверы различных служб. Так, в Linux — это сервисы, находящиеся под управлением подсистемы systemd. В этом случае для того, чтобы стать сервером-сервисом, все тот же пользовательский процесс должен выполнить некоторую фиксированную последовательность действий [19, 20], называемую *демонизацией*.

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

Возникает вопрос: а может ли *пользователь* (потребитель) написать и выполнить собственный код, выполняющийся в режиме супервизора, а значит, имеющий все полномочия расширять (или даже изменять) функциональность ядра Linux? Да, может! И эта техника программирования называется *программированием модулей ядра*. И именно она позволяет, в частности, создавать драйверы любого нестандартного оборудования[2].

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

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

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

> ***Примечание***
>
> 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` (Linux — система с абсолютно открытым кодом всех компонентов и подсистем), которая является не чем иным, как заурядным пользовательским приложением, выполните в своем коде те манипуляции с привилегиями, которые проделывает `insmod`, и вы получите желаемое приложение. Естественно, что все это потребует от приложения привилегий root при запуске, но это все то же минимальное требование, которое вообще обязательно при любой работе с модулями ядра.

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

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!***

В самом начале знакомства с техникой написания модулей ядра Linux проще не вдаваться в пространные объяснения, а создать простейший модуль (код такого модуля интуитивно понятен всякому программисту), собрать его и наблюдать его работу. И только потом, ознакомившись с некоторыми основополагающими принципами и приемами работы из мира модулей, перейти к их систематическому изучению.

Вот с такого образца простейшего модуля ядра (см. папку first_hello в сопровождающем книгу файловом архиве) мы и начнем наш экскурс:

hello_printk.c

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>>");

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 повторяться при сборке всех модулей ядра:

Makefile

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

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

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

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

Примечание

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

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

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

Это связано с тем, что в системе, которая специально не готовилась для сборки ядра, по умолчанию *не установлены* те программные пакеты, которые необходимы для этой специфической деятельности, — в частности, заголовочные файлы кода ядра. Более того, в вашем дистрибутиве может быть изначально даже не установлены инструменты компиляции и сборки: `gcc` и `make`. Все это относится к вопросу создания среды разработки модулей ядра, и мы рассмотрим эти вопросы чуть позже. А пока будем предполагать, что мы уже установили все необходимые программные пакеты для сборки модулей ядра:

```
$ make
make -C /lib/modules/5.4.0-113-generic/build
M=/home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello modules
make[1]: вход в каталог "/usr/src/linux-headers-5.4.0-113-generic"
  CC [M]  /home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello/hello_printk.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello/hello_printk.mod.o
  LD [M]  /home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello/hello_printk.ko
make[1]: выход из каталога «/usr/src/linux-headers-5.4.0-113-generic»
```

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

```
$ ls *.ko
hello_printk.ko

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

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

А вот, для сравнения, сборка в точности **того же** *исходного* **кода** модуля, как он собирается на Raspberry Pi, — совершенно отличающаяся архитектура: процессор Cortex ARMv7 32-bit:

```
$ make
make -C /lib/modules/5.15.25-sunxi/build M=/home/olej/2022/kernel/examples/first_hello modules
make[1]: вход в каталог "/usr/src/linux-headers-5.15.25-sunxi"
  CC [M]  /home/olej/2022/kernel/examples/first_hello/hello_printk.o
  MODPOST /home/olej/2022/kernel/examples/first_hello/Module.symvers
  CC [M]  /home/olej/2022/kernel/examples/first_hello/hello_printk.mod.o
  LD [M]  /home/olej/2022/kernel/examples/first_hello/hello_printk.ko
make[1]: выход из каталога «/usr/src/linux-headers-5.15.25-sunxi»
```

Это означает, что модули ядра будут собираться единообразно в любой из множества аппаратных архитектур, поддерживаемых Linux: *x*86, ARM, MIPS, PPC... — 32- и 64-битных. С другой стороны, это совсем не означает, что в любой архитектуре код для сборки окажется абсолютно идентичным: те или иные имена ядра могут оказаться отсутствующими в какой-то из архитектур, определяться как функции или макросы, есть и другие тонкие различия. Эти различия придется отрабатывать препроцессорными определениями (основывающимися на конфигурационных параметрах сборки ядра), о чем мы поговорим позже.

Загрузка и исполнение

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

```
$ sudo insmod hello_printk.ko
$ sudo rmmod hello_printk
$ tail -n2 /var/log/kern.log
Jun  5 15:27:19 R420 kernel: [14377.822570] Hello, world!
Jun  5 15:27:32 R420 kernel: [14390.766363] Goodbye, world!
```

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

```
$ dmesg | tail -n2
[14377.822570] Hello, world!
[14390.766363] Goodbye, world!
```

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

Но даже и в текстовую консоль (при отработке в текстовом режиме) 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 вкратце рассмотрены позже).

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

```
$ modinfo hello_printk.ko
filename:       /home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello/hello_printk.ko
author:         Oleg Tsiliuric <olej.tsil@gmail.com>>
license:        GPL
srcversion:     9526D45F847C0B4EF5BBBDC
depends:
retpoline:      Y
name:           hello_printk
vermagic:       5.4.0-113-generic SMP mod_unload modversions
$ sudo insmod hello_printk.ko
$ lsmod | head -n2
Module                  Size  Used by
hello_printk           16384  0
```

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

Точки входа и завершения

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

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

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

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

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

> ***Примечание***
>
> Обратите внимание: функция завершения по своему прототипу не имеет возвращаемого значения и поэтому не может сообщить о невозможности каких-либо действий, когда она уже начала выполняться. Идея состоит в том, что система при `rmmod` сама проверит требования допустимости вызова функции завершения, и если они не соблюдены, просто не вызовет эту функцию. Это приводит иногда к тому, что при отработке кода модуля он может стать неудаляемым в принципе. В этом случае для продолжения работы над таким модулем вас может спасти только перезагрузка системы!

Показанные здесь соглашения по объявлению функций инициализации и завершения являются общепринятыми. Но существует еще один — недокументированный — способ описания этих функций: воспользоваться непосредственно их *предопределенными* именами, а именно: `init_module()` и `cleanup_module()`. Это может быть записано так:

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

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

Конечно, эта запись никак не способствует улучшению читаемости текста, но иногда может существенно сократить рутину записи, особенно в коротких иллюстративных программных примерах. Мы будем иногда ее использовать именно так. Кроме того, такую запись нужно понимать, поскольку она задействуется кое-где в коде ядра — в литературе и в обсуждениях по ядру.

Внутренний формат модуля

Обращаясь к структуре модуля ядра, мы можем увидеть для начала, что собранный нами модуль является *объектным* файлом ELF-формата (показано для сравнения в архитектуре Intel Xeon и ARMv7 соответственно):

```
$ file hello_printk.ko
hello_printk.ko: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV),
BuildID[sha1]=dc1a94b0bc8019d82aee89ffff3b3e562249f016, not stripped
$ file hello_printk.ko
hello_printk.ko: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV),
BuildID[sha1]=654d8e3a49c4b9eb9d0c0326f7fedb52124c232e, not stripped
```

Всесторонний анализ объектных файлов производится утилитой `objdump`, имеющей множество опций в зависимости от того, что мы хотим посмотреть, из которых только немногие понадобятся для наших целей:

```
$ 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:     формат файла elf64-x86-64

Разделы:
Idx Name          Разм      VMA               LMA               Фа  смещ.  Выр.
  0 .note.gnu.build-id 00000024  0000000000000000  0000000000000000  00000040  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.Linux   00000018  0000000000000000  0000000000000000  00000064  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .text         00000000  0000000000000000  0000000000000000  0000007c  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .init.text    00000019  0000000000000000  0000000000000000  0000007c  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  4 .exit.text    00000012  0000000000000000  0000000000000000  00000095  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
...
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .data         00000000  0000000000000000  0000000000000000  00000260  2**0
                  CONTENTS, ALLOC, LOAD, DATA
...
```

Здесь секции:

- `.text` — код модуля (инструкуции);
- `.init.text`,`.exit.text` — код инициализации модуля и завершения соответственно;

- `.modinfo` — текст макросов модуля;
- `.data` — инициализированные данные;
- `.bss` — неинициализированные данные (Block Started Symbol);

А вот еще один альтернативный инструмент детального анализа объектной структуры модуля (он дает несколько иные срезы информации) — хорошо видна сфера видимости *имен* (в колонке `Связ`: `LOCAL`/`GLOBAL` — имя *экспортируется* или нет за пределы кода модуля):

```
$ readelf -s hello_printk.ko
Таблица символов «.symtab» содержит 33 элемента:
   Чис:    Знач           Разм Тип     Связ   Vis      Индекс имени
...
$ readelf -s hello_printk.ko | grep FUNC
    24: 0000000000000000    25 FUNC    LOCAL  DEFAULT    4 hello_init
    25: 0000000000000000    18 FUNC    LOCAL  DEFAULT    6 hello_exit
    29: 0000000000000000    18 FUNC    GLOBAL DEFAULT    6 cleanup_module
    31: 0000000000000000    25 FUNC    GLOBAL DEFAULT    4 init_module
```

Обратите внимание, как здесь выглядят точки входа и завершения модуля, обсуждавшиеся чуть ранее.

Попутно — чтобы к нему больше не возвращаться — рассмотрим здесь уже поднимавшийся вопрос: чем формат модуля `*.ko` *отличается* от обыкновенного и использовавшегося когда-то ранее для модулей объектного формата `*.o` (тем более что последний появляется в процессе сборки модуля как промежуточный результат):

```
$ ls -l hello_printk.*o
-rw-rw-r-- 1 olej olej 4528 июн  6 12:35 hello_printk.ko
-rw-rw-r-- 1 olej olej 3200 июн  6 12:35 hello_printk.mod.o
-rw-rw-r-- 1 olej olej 2232 июн  6 12:35 hello_printk.o
$ readelf -s hello_printk.o | grep __UNIQUE
    12: 0000000000000000    45 OBJECT  LOCAL  DEFAULT   11 __UNIQUE_ID_author37
    13: 000000000000002d    12 OBJECT  LOCAL  DEFAULT   11 __UNIQUE_ID_license36
$ readelf -s hello_printk.ko | grep __UNIQUE
    16: 0000000000000039    35 OBJECT  LOCAL  DEFAULT   11 __UNIQUE_ID_srcversion40
    17: 000000000000005c     9 OBJECT  LOCAL  DEFAULT   11 __UNIQUE_ID_depends39
    19: 0000000000000065    12 OBJECT  LOCAL  DEFAULT   11 __UNIQUE_ID_retpoline38
    20: 0000000000000071    18 OBJECT  LOCAL  DEFAULT   11 __UNIQUE_ID_name37
    21: 0000000000000083    55 OBJECT  LOCAL  DEFAULT   11 __UNIQUE_ID_vermagic36
    26: 0000000000000000    45 OBJECT  LOCAL  DEFAULT   11 __UNIQUE_ID_author37
    27: 000000000000002d    12 OBJECT  LOCAL  DEFAULT   11 __UNIQUE_ID_license36
```

Как легко видеть, модуль ядра *дополнен* несколькими именами (в таблице имен) со строчными константными значениями, предназначенными для идентификации (автор, лицензия, ...) и контроля целостности, соответствия модуля ядру, в которое он загружается:

```
$ modinfo hello_printk.ko
filename:       /home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello/hello_printk.ko
author:         Oleg Tsiliuric <olej.tsil@gmail.com>>
license:        GPL
srcversion:     9526D45F847C0B4EF5BBBDC
depends:
retpoline:      Y
name:           hello_printk
vermagic:       5.4.0-113-generic SMP mod_unload modversions
$ uname -a
Linux R420 5.4.0-113-generic #127-Ubuntu SMP Wed May 18 14:30:56 UTC 2022 x86_64 x86_64 x86_64
GNU/Linux
```

По причинам, которые будут обсуждаться далее, попытка загрузки модуля в ядро, не соответствующее версии ядра (сигнатуре), в котором модуль собирался (`make`), в общем случае привела бы к катастрофическому краху ядра... Настолько, что ядро зачастую даже не успевает сохранить посмертный дамп состояния. Для предотвращения таких попыток и отменено маркирование модуля сигнатурой.

Как уже *предварительно* должно стать понятно, в Linux *не может быть*, в принципе, бинарных модулей ядра, бинарных драйверов. Это определяется идеологией модульной структуры ядра Linux. Модули могут быть представлены только в исходном коде и при смене или обновлении ядра должны *компилироваться* из исходного кода. Существуют дополнительные (сверх основных системных команд), более поздние инструменты (пакеты) для обслуживания модулей: Dynamic Kernel Module Support (DKMS), Kernel Modules (KMOD), Akmods (в Fedora) — они обеспечивают *динамическую пересборку* определенных модулей ядра при изменениях ядра. Но все они собирают новый экземпляр модуля, ориентируясь на исходный код модуля.

Диагностика модуля

Для диагностического вывода из модуля используем главным образом вызов `printk()`. Он несколько подобен своему собрату `printf()`, по своим правилам и формату общеизвестному из пользовательского пространства, но имеет существенно большее число используемых форматов элементов вывода.

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

```
int trace_printk(const char *fmt, ...)
```

Первому параметру (форматной строке) может *предшествовать* (а может и не предшествовать) константа-квалификатор, определяющая *уровень сообщений*. Определения констант для 8 уровней сообщений, записываемых в вызове `printk()`, вы найдете в файле /lib/modules/`uname -r`/build/include/linux/kern_levels.h:

```
#define KERN_EMERG      KERN_SOH "0"    /* system is unusable */
#define KERN_ALERT      KERN_SOH "1"    /* action must be taken immediately */
```

```
#define KERN_CRIT       KERN_SOH "2"    /* critical conditions */
#define KERN_ERR        KERN_SOH "3"    /* error conditions */
#define KERN_WARNING    KERN_SOH "4"    /* warning conditions */
#define KERN_NOTICE     KERN_SOH "5"    /* normal but significant condition */
#define KERN_INFO       KERN_SOH "6"    /* informational */
#define KERN_DEBUG      KERN_SOH "7"    /* debug-level messages */

#define KERN_DEFAULT    ""              /* the default kernel loglevel */
```

Предшествующая константа `KERN_*` (в нашем случае) не является отдельным параметром (не отделяется запятой) и (как видно из определений) представляет собой символьную строку определенного вида, которая *конкатенируется* с первым параметром, являющимся *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");
printk("целое %ld", 12345);
printk("адрес %px", &printk);
```

Вызов `printk()` сам по себе не производит непосредственно какой-либо вывод, а направляет выводимую строку демону системного журнала, который уже перезаписывает полученную строку: а) на *текстовую консоль* и б) в системный журнал. При работе в графической системе X11 вывод `printk()` в терминал `xterm` (или другой) не попадает (графический терминал *не является* текстовой консолью!), поэтому остается только в системном журнале. Ваш демон системного журнала:

```
$ ps -A | grep logd
   1231 ?        00:00:00 rsyslogd
$ systemctl status rsyslog
● rsyslog.service - System Logging Service
     Loaded: loaded (/lib/systemd/system/rsyslog.service; enabled; vendor preset: enabled)
     Active: active (running) since Mon 2022-06-06 10:33:32 EEST; 9h ago
TriggeredBy: ● syslog.socket
       Docs: man:rsyslogd(8)
             https://www.rsyslog.com/doc/
   Main PID: 1231 (rsyslogd)
      Tasks: 4 (limit: 115828)
     Memory: 4.1M
     CGroup: /system.slice/rsyslog.service
             └─1231 /usr/sbin/rsyslogd -n -iNONE

июн 06 10:33:32 R420 systemd[1]: Starting System Logging Service...
июн 06 10:33:32 R420 rsyslogd[1231]: imuxsock: Acquired UNIX socket
'/run/systemd/journal/syslog' (fd 3) from systemd.  [v8.2001.0]
```

```
июн 06 10:33:32 R420 rsyslogd[1231]: rsyslogd's groupid changed to 110
июн 06 10:33:32 R420 rsyslogd[1231]: rsyslogd's userid changed to 104
июн 06 10:33:32 R420 rsyslogd[1231]: [origin software="rsyslogd" swVersion="8.2001.0"
x-pid="1231" x-info="https://www.rsyslog.com"] start
июн 06 10:33:32 R420 systemd[1]: Started System Logging Service.
```

Вывод системного журнала направляется, как уже сказано, в файл системного журнала и отображается в текстовой консоли, но не отображается в графических терминалах X11. Большой неожиданностью может стать отсутствие вывода `printk()` и в текстовую консоль тоже. Этот вывод обеспечивается демоном системного журнала, и он выводит только сообщения *выше порога*, установленного ему при запуске. Для снижения (или изменения) порога вывода диагностики этого демона, может быть, придется перезапустить его с другими параметрами. Для модульного rsyslogd это определяется в его конфигурационном файле /etc/rsyslog.conf (или в любом файле в каталоге /etc/rsyslog.d/*.conf), где вы найдете строки подобного вида:

```
$ cat /etc/rsyslog.conf
....
kern.*                          /dev/console
kern.*                          /var/log/kern.log
...
```

Вы можете *добавить* любое целеуказание для вывода `printk()` в конфигурационные файлы демона, но целесообразнее добавить в каталог /etc/rsyslog.d *новый файл*, содержащий собственные строки конфигурации демона (после завершения работы вы можете просто удалить этот файл):

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

Этим мы хотим вывод модулей направлять на консоль 12 (любую неинициированную, т. е. стандартную, с номером больше 6), на которую переключаемся комбинацией клавиш: <Ctrl><Alt><F12>. Но для любого изменения конфигурации нужно заставить демон rsyslogd перечитать его конфигурации. Это можно сделать двумя способами... Просто перезапустить демон логирования:

```
$ sudo systemctl restart rsyslog
$ systemctl status rsyslog
● rsyslog.service - System Logging Service
     Loaded: loaded (/lib/systemd/system/rsyslog.service; enabled; vendor preset: enabled)
     Active: active (running) since Mon 2022-06-06 21:54:06 EEST; 2s ago
TriggeredBy: ● syslog.socket
       Docs: man:rsyslogd(8)
             https://www.rsyslog.com/doc/
   Main PID: 71214 (rsyslogd)
      Tasks: 4 (limit: 115828)
     Memory: 1.2M
     CGroup: /system.slice/rsyslog.service
             └─71214 /usr/sbin/rsyslogd -n -iNONE
```

```
июн 06 21:54:06 R420 systemd[1]: Starting System Logging Service...
июн 06 21:54:06 R420 rsyslogd[71214]: imuxsock: Acquired UNIX socket
'/run/systemd/journal/syslog' (fd 3) from systemd.  [v8.2001.0]
июн 06 21:54:06 R420 rsyslogd[71214]: rsyslogd's groupid changed to 110
июн 06 21:54:06 R420 systemd[1]: Started System Logging Service.
июн 06 21:54:06 R420 rsyslogd[71214]: rsyslogd's userid changed to 104
июн 06 21:54:06 R420 rsyslogd[71214]: [origin software="rsyslogd" swVersion="8.2001.0"
x-pid="71214" x-info="https://www.rsyslog.com"] start
```

Или можно поступить проще: направить демону сигнал `SIGHUP` (по такому сигналу многие из демонов Linux перечитывают и обновляют свою конфигурацию):

```
$ pgrep rsyslogd
1248
$ sudo kill -HUP 1248
```

Или даже еще проще:

```
$ sudo kill -HUP `pgrep rsyslogd`
```

Если вы желаете, то можете даже направить вывод из ядра `printk()` на выбранный *графический* терминал (эмулятор терминала), на котором производите отладку (и куда обычно вывод не производится), что может быть удобно в определенных условиях отладки.

Для этого:

1. Определяем выбранный терминал, проделав в нем:

   ```
   $ tty
   /dev/pts/13
   ```

2. Создаем конфигурационный файл демона логирования:

   ```
   # touch /etc/rsyslog.d/70-my.conf
   ```

3. Прописываем выбранный терминал в конфигурции:

   ```
   $ cat /etc/rsyslog.d/70-my.conf
   kern.info;kern.notice;kern.warn              /dev/pts/13
   ```

4. Перечитываем конфигурацию rsyslogd, как было сказано ранее.

5. После чего можем вести отладку модуля в выбранном терминале, не отвлекаясь на системный журнал:

   ```
   $ sudo insmod hello_printk.ko
   Jun  6 22:17:18 R420 kernel: [42234.754863] Hello, world!
   $ sudo rmmod hello_printk
   Jun  6 22:17:22 R420 kernel: [42239.030447] Goodbye, world!
   ```

Функция `printk()`, как видно из ее прототипа, показанного ранее, — это функция с *произвольным числом* параметров. Первый ее *строчный* параметр может включать в свой состав элементы форматирования (шаблоны вывода), начинающиеся со знака `%`. Этим она сильно напоминает по использованию функцию `printf()` из пользовательского пространства. Но будьте осторожны — внешность обманчива!

Символы форматирования printk() отличаются от printf(), и они гораздо разнообразнее. Они стоят того, чтобы их изучить со всей тщательностью [48]. Мы скоро вернемся к этому вопросу — при рассмотрении формата адресов ядра.

Уровни диагностики в */proc*

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

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

Первое число в выходных данных — это уровень журнала консоли, второе — уровень журнала по умолчанию, третье — минимальный уровень журнала, а четвертое — максимальный уровень журнала. Уровень журнала 4, например, соответствует KERN_WARNING. Таким образом, все сообщения с уровнями журнала 3, 2, 1 и 0 будут отображаться на экране, а также регистрироваться, а сообщения с уровнем журнала 4, 5, 6, 7 только регистрируются и могут быть просмотрены с помощью команды dmesg.

Здесь цифры — это установленные пороги *отсечения* вывода: будут разрешены только сообщения модуля со значениями KERN_* *ниже* порога (приоритет сообщений выше), а сообщения со значениями, равными или бóльшими, будут отфильтровываться. Нас интересует в первую очередь первое значение — уровень сообщений printk(), начиная с которого и выше сообщения не будут выводиться на текстовую консоль.

Но эти значения не только диагностируемые, но и (при определенных полномочиях) записываемые, изменяемые:

```
$ ls -l /proc/sys/kernel/printk
-rw-r--r-- 1 root root 0 июн  6 10:33 /proc/sys/kernel/printk
```

Так мы можем поменять уровни логирования:

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

Или даже так[3]:

```
$ echo 8 | sudo tee /proc/sys/kernel/printk
8
$ cat /proc/sys/kernel/printk
8       4       1       7
```

[3] Подсказано одним из читателей рукописи.

После такой переустановки диагностический модуль (каталог log_level, мы его не обсуждаем детально) выведет в *текстовую* консоль (которую мы предварительно только что настроили на графический терминал):

log_level.c

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

static const char* levl[] = {
   KERN_DEBUG,   KERN_INFO, KERN_NOTICE,
   KERN_WARNING, KERN_ERR,  KERN_CRIT,  KERN_ALERT
};

static int __init log_init(void) {
    char out[80];
    int i;
    for(i = 0; i < sizeof(levl) / sizeof(levl[0]); i++) {
        sprintf(out, "message level: %s\n", levl[i] );
        printk(out);
    }
    return -1;
}
```

```
$ sudo insmod log_level.ko
[sudo] пароль для olej:
Jun  7 01:07:50 R420 kernel: [52466.817607] log_level: module license 'unspecified' taints
                                                                                    kernel.
Jun  7 01:07:50 R420 kernel: [52466.817610] Disabling lock debugging due to kernel taint
Jun  7 01:07:50 R420 kernel: [52466.818772] message level: 7
Jun  7 01:07:50 R420 kernel: [52466.818773] message level: 6
Jun  7 01:07:50 R420 kernel: [52466.818774] message level: 5
Jun  7 01:07:50 R420 kernel: [52466.818775] message level: 4
Jun  7 01:07:50 R420 kernel: [52466.818776] message level: 3
Jun  7 01:07:50 R420 kernel: [52466.818776] message level: 2
Jun  7 01:07:50 R420 kernel: [52466.818777] message level: 1
insmod: ERROR: could not insert module log_level.ko: Operation not permitted
```

Представление адресов в Linux

В диагностике ядра в гораздо большей степени, чем в пространстве пользователя, приходится изучать адресные значения. А поэтому, прежде чем говорить о форматах отображения `printk()`, нужно уточнить, что собой представляют адреса в Linux. Картина очень укрупненная, но для наших целей достаточная...

Прежде всего, если явно не оговаривается другое, мы говорим о *виртуальных*, или *логических*, адресах, которые позже сложным образом, через механизм страничного отображения, отображаются в *физические* страницы оперативной памяти. Каждый

процесс пользовательского пространства имеет собственное адресное пространство — от 0 до некоторого значения M. Ядро системы имеет свое собственное адресное пространство, простирающееся от границы M и выше. Таким образом, Linux использует *плоское* адресное пространство, которое позволяет не переключать некоторые сегментные регистры при переключении пользователь/ядро, что повышает производительность системы. Но детали этого процесса нас пока мало занимают...

- 32-битная адресация. Максимальный размер адресного пространства при 32-битных адресах равен 4 Gb. В этом режиме в Linux принято решение разделить указанное адресное пространство между пользователем и ядром в соотношении 3:1. Границу M при этом указывает макрос-указатель `current`, равный по умолчанию `c0000000`. Это константа периода компиляции, и она может быть изменена, но путем пересборки ядра, что может иметь смысл (и имеет место) для некоторых аппаратных архитектур. Понятно, что адреса в приложениях пользователя будут выглядеть ниже этой границы, а адреса выше этой границы будут относиться к ядру. Значение этой границы определяется (при сборке ядра) конфигурационным параметром `CONFIG_PAGE_OFFSET`[4] :

```
$ uname -a
Linux orangepione 5.15.25-sunxi #22.02.1 SMP Sun Feb 27 09:23:25 UTC 2022 armv7l GNU/Linux
$ cat /boot/config-`uname -r` | grep CONFIG_PAGE_OFFSET
CONFIG_PAGE_OFFSET=0xC0000000
```

- 64-битная адресация. Все становится гораздо веселее в Linux 64-бит. Прежде всего, когда мы говорим временами Intel *x*86_64 — это неверно. 64-битную спецификацию опубликовала в 2000 году фирма AMD, а Intel использует только лицензию AMD. Далее, спецификация предполагает, что для адресации используются только 48 (0–47) младших битов из 64, а старшие биты (48–63) расширяются битом 47 — подобно тому, как это делается при расширении знакового разряда отрицательных целых. Такая спецификация адреса в сегодняшних процессорах поддерживается аппаратно. Это позволяет планировать легкость расширения в будущем до адресов 52, 57 и даже 64 (такие цифры называют в планах). В Linux 64-битное разделение виртуальных адресных пространств пользователь/ядро производится поровну — 1:1, по значению бита 47 (1 — ядро, 0 — пользовательский процесс). Поэтому адреса плоского адресного пространства для пользовательских процессов будут выглядеть так: [`0000000000000000` — `00007fffffffffff`], а для ядра — так: [`ffff800000000000` — `ffffffffffffffff`] (скобки квадратные и круглые в обозначении этих диапазонов означают, как это и принято в математике, «включая» и «исключая» соответственно). Как виртуальные 48-битные адреса затем аппаратно, с помощью MMU, преобразуются в физические адреса реальной памяти, нас на этом этапе не интересует. Нас интересует, как они *представляются*. В этой модели памяти параметр конфигурации ядра `CONFIG_PAGE_OFFSET` не определяется — 48-битная граница фиксирована и в переопределении не нуждается:

[4] В примере показан реальный вывод в системе с процессором ARM, хотя в более ранней литературе можно встретить утверждение, что границей раздела для ARM является `0x80000000`.

```
$ cat /boot/config-`uname -r` | grep CONFIG_PAGE_OFFSET
$
```

Форматы вывода

Форматы, используемые `printk()`, напоминают `printf()`, но существенно отличаются от них по существу. Во-первых, в ядре нет (и не нужно) операций с вещественными значениями, а поэтому нет и форматов `%f`. Все остальные форматы описаны в небольшом документе «How to get printk format specifiers right» [47] от участников команды разработчиков ядра (который настоятельно рекомендую изучить). Среди них есть много интуитивно понятных (по аналогии с `printf()`): `%d`, `%u`, `%x`, ... и соответственно: `%lu`, `%llu` ...

Из других полезных форматов:

- `%pM` — сетевые 6-byte адреса MAC/FDDI: 00:01:02:03:04:05;
- `%pI4` — сетевые IPv4 адреса: 1.2.3.4;
- `%pI6` — сетевые Ipv6 адреса: 0001:0002:0003:0004:0005:0006:0007:0008.

Но самый необходимый нам тип данных в ядре — это адреса, указатели. Которые, кроме того, могут принципиально различаться на 32-битные и 64-битные. Для вывода адресов в ядре предусмотрено до десятка разных форм. Мы будем использовать в примерах «нативную» форму `%px`, о которой авторы документа пишут:

> Для печати указателей, когда вы действительно хотите напечатать адрес. Прежде чем печатать указатели с помощью `%px`, подумайте, не происходит ли утечка конфиденциальной информации о расположении ядра в памяти. `%px` функционально эквивалентен `%lx`. `%px` предпочтительнее `%lx`, потому что он более удобен для `grep`.

Поскольку это нам чрезвычайно понадобится в последующем изложении, проведем некоторые эксперименты (в том числе и относительно соотношения указателей и их целочисленных эквивалентов). Прежде всего, вспомним и посмотрим, чему равны и как отображаются адреса в пространстве пользователя:

intptr.c

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

static uint16_t d1 = 1234;
static uint16_t d2;

int main(void) {
        uint16_t d3 = 1234;
        printf("%20p%20p%20p\n", &d1, &d2, &d3);
        printf("%20lx%20lx%20lx\n",
               (uintptr_t)&d1,
               (uintptr_t)&d2,
               (uintptr_t)&d3);
```

```
        printf("%20llx%20llx%20llx\n",
               (long long unsigned int)&d1,
               (long long unsigned int)&d2,
               (long long unsigned int)&d3);
        return 0;
}
```

Выполнение:

- 32-бит:

```
$ uname -a
Linux orangepione 5.15.25-sunxi #22.02.1 SMP Sun Feb 27 09:23:25 UTC 2022 armv7l GNU/Linux
$ ./intptr
            0x421044            0x421048          0xbe82442e
              421044              421048            be82442e
              421044              421048    ffffffffbe82442e
```

- 64-бит:

```
$ uname -a
Linux R420 5.4.0-117-generic #132-Ubuntu SMP Thu Jun 2 00:39:06 UTC 2022 x86_64 x86_64
x86_64 GNU/Linux
$ ./intptr
      0x560478512010      0x560478512014      0x7fff53096ea6
        560478512010        560478512014        7fff53096ea6
        560478512010        560478512014        7fff53096ea6
```

На что (особенно интересное) обратим особое внимание? На то, как адрес переменной в локальном стеке (d3) отображается «впритык» к области раздела пользователь/ядро (`c0000000` — для 32-бит, `ffff800000000000` — для 64-бит).

Теперь все то же самое, но уже в пространстве ядра (модуль):

intptr.c

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

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");

static int __init my_init(void) {
        uint16_t d = 1234;
        void show(void* p) {
                printk("%20p%20px%20pK%20lx\n",
                       p, p, p, (uintptr_t)p);
        }
        if (IS_ENABLED(CONFIG_64BIT)) {
                printk("64-bit");
        }
```

```
        else {
                printk("32-bit");
        }
        printk("%20s%20s%20s%20s\n", "%p", "%px", "%pK", "int");
        show(&d);
#if defined(CONFIG_64BIT)
        show(printk);
#else
        show(_printk);
#endif
        show((void*)PAGE_OFFSET);
        show((void*)current);
        show((void*)0xffff800000000000);
        return -2;
}

module_init(my_init);
```

Сравниваем выполнение:

- 32-бит (архитектура ARM):

```
$ uname -a
Linux orangepione 5.15.25-sunxi #22.02.1 SMP Sun Feb 27 09:23:25 UTC 2022 armv7l GNU/Linux
$ sudo insmod aslr.ko
insmod: ERROR: could not insert module aslr.ko: Unknown symbol in module
$ dmesg | tail -n8
[52406.337384]                   0                   0                   0
[332941.968199] 32-bit
[332941.968219]                  %p                 %px                 %pK       int
[332941.968232]            a2e0bee6            d617ddc2            a2e0bee6       d617ddc2
[332941.968251]            6de3fc8a            c09c49a1            6de3fc8a       c09c49a1
[332941.968260]            6d66c095            c0000000            6d66c095       c0000000
[332941.968270]            849d5575            c17e95c0            849d5575       c17e95c0
[332941.968279]                   0                   0                   0       0
```

- 64-бит:

```
$ uname -a
Linux R420 5.4.0-117-generic #132-Ubuntu SMP Thu Jun 2 00:39:06 UTC 2022 x86_64 x86_64
x86_64 GNU/Linux
$ make
make -C /lib/modules/5.4.0-117-generic/build
M=/home/olej/2022/own.BOOKs/BHV.kernel/examples.tmp/aslr modules
make[1]: вход в каталог «/usr/src/linux-headers-5.4.0-117-generic»
  CC [M]  /home/olej/2022/own.BOOKs/BHV.kernel/examples.tmp/aslr/aslr.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/olej/2022/own.BOOKs/BHV.kernel/examples.tmp/aslr/aslr.mod.o
  LD [M]  /home/olej/2022/own.BOOKs/BHV.kernel/examples.tmp/aslr/aslr.ko
```

```
make[1]: выход из каталога «/usr/src/linux-headers-5.4.0-117-generic»
$ sudo insmod aslr.ko
insmod: ERROR: could not insert module aslr.ko: Unknown symbol in module
$ dmesg | tail -n8
[ 3295.057036] EXT4-fs (sdc1): mounted filesystem with ordered data mode. Opts: (null)
[18255.111738] 64-bit
[18255.111741]         %p                %px                %pK         int
[18255.111746]        711d7836    ffffc1b8ebea3c56    ffffc1b8ebea3c56    ffffc1b8ebea3c56
[18255.111747]        f2444aa0    ffffffff84c8b926    ffffffff84c8b926    ffffffff84c8b926
[18255.111748]        b585547a    ffff9fa9c0000000    ffff9fa9c0000000    ffff9fa9c0000000
[18255.111749]        7b1f8113    ffff9fc08c079740    ffff9fc08c079740    ffff9fc08c079740
[18255.111749]        67ea386c    ffff800000000000    ffff800000000000    ffff800000000000
```

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 'xxxx': No such file or directory` — неверно указан путь к файлу модуля, возможно, в указание имени файла не включено стандартное расширение имени *файла* модуля (*.ko), но это нужно делать обязательно;
- `insmod: error inserting 'xxxx.ko': -1 Operation not permitted` — наиболее вероятная причина: у вас элементарно нет прав root для выполнения операций установки модулей. Другая причина того же сообщения: функция инициализации модуля возвратила *ненулевое значение*, нередко такое завершение планируется *преднамеренно*, особенно на этапах отладки модуля (и о таком варианте использования мы будем говорить неоднократно);
- `insmod: error inserting 'xxxx.ko': -1 Invalid module format` — модуль скомпилирован для другой версии ядра (например 32-бит вместо 64). Просто перекомпилируйте модуль. Это та ошибка, которая почти наверняка возникнет, когда вы перенесете любой рабочий пример модуля на другой компьютер и попытаетесь там загрузить модуль: получить совпадение сигнатур разных инсталляций до уровня подверсий почти нереально;
- `insmod: error inserting 'xxxx.ko': -1 File exists` — модуль с таким именем уже загружен, попытка загрузить модуль повторно;
- `insmod: error inserting 'xxxx.ko': -1 Invalid parameters` — модуль запускается с указанным параметром, не соответствующим по типу ожидаемому для этого параметра;
- `insmod: ERROR: could not insert module md_32o.ko: Unknown symbol in module` — в коде модуля использовано имя (вызов функции API ядра), которое не объявлено в ядре как *экспортируемое*. Такой модуль может быть собран, но при сборке — будьте внимательны — вы *уже* получите сообщение (WARNING):

```
$ make
make -C /lib/modules/4.9.0-279-antix.1-486-smp/build
M=/home/olej/2022/kernel/sys_call_table/export modules
make[1]: вход в каталог «/usr/src/linux-headers-4.9.0-279-antix.1-486-smp»
  CC [M]  /home/olej/2022/kernel/sys_call_table/export/md_32o.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: "sys_open" [/home/olej/2022/kernel/sys_call_table/export/md_32o.ko] undefined!
  CC      /home/olej/2022/kernel/sys_call_table/export/md_32o.mod.o
  LD [M]  /home/olej/2022/kernel/sys_call_table/export/md_32o.ko
make[1]: выход из каталога «/usr/src/linux-headers-4.9.0-279-antix.1-486-smp»
```

Загрузить такой модуль нельзя, и при загрузке причина будет указана в логе ядра:

```
$ dmesg | tail -n1
[125175.046023] md_32o: Unknown symbol sys_open (err 0)
```

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

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

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

  Здесь счетчик ссылок модуля ненулевой, но нет имени модулей, ссылающихся на него. Что можно предпринять в подобных случаях? Только перезагрузка системы!

- `ERROR: Removing 'xxxx': 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 в принципе невыгружаемый модуль, и поможет только перезагрузка системы):

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

Обсуждение

Здесь и далее некоторые главы будут завершаться короткими разделами под названием «Обсуждение». Там, где они есть (а есть они далеко не во всех главах), это не непосредственные итоги состоявшегося в главе обсуждения ее основного материала, а, скорее, некоторые отступления в сторону, «вынесенные за скобки», — те попутные мысли, которые возникали при обсуждении, но не сформулировались в законченную форму. Иногда это дополнительные рутинные детали, важные на практике, иногда вопросы по обсужденному материалу или дополнения к нему...

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

- программирование модуля ядра ничем принципиально не отличается от программирования в пространстве пользователя. Однако для обеспечения функциональности модуля мы используем *совершенно другой* набор API (`printk()`, например, вместо привычного `printf()`). Такая дуальность вызовов (внешняя подобность) будет наблюдаться практически для всех разделов API (функции работы с текстовыми строками, управление памятью, примитивы синхронизации, ...). Но не стоит и обольщаться — эти имена и форматы вызовов API ядра при внешнем подобии временами могут существенно отличаться от API POSIX. Относительно API пространства пользователя существуют стандарты (POSIX и др.), и они хорошо описаны в литературе. API пространства ядра плохо описаны и могут существенно изменяться от одной версии ядра к другой (главным образом именно из-за отсутствия таких сдерживающих стандартов, как в API пространства пользователя). Поиск адекватных вызовов API для текущей версии ядра и есть одна из существенных трудностей программирования модулей ядра Linux. Мы еще неоднократно будем возвращаться к этим вопросам по ходу дальнейшего изложения;
- в приведенных в главе примерах и далее по всему тексту книги команды загрузки модуля я часто буду записывать вот так:

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

  Это связано с тем, что команда `insmod`, в отличие от подобной по действию команды `modprobe`, на которой мы остановимся позже, не предполагает какого-либо *поиска* указанного ей к загрузке *файла* модуля, как это имеет место при запуске файлов приложений (список `$PATH`): либо файл точно указан со своим полным путевым именем (абсолютным или относительным), и тогда он загружается, либо файл не находится по такому имени файла, и тогда `insmod` аварийно завершается:

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

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

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

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

Примечание

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

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

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

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

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

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

---

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

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

```
# insmod hello_printk.ko
```

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

```
# rmmod hello_printk
```

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

- обратите внимание еще на одну особенность: как бы мы ни переименовывали *имя файла* уже ранее скомпилированного модуля, имя, под которым модуль *будет известен системе*, останется неизменным:

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

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

- остановимся еще на одном коротком замечании, которое будем еще неоднократно подтверждать и уточнять далее.. Модуль ядра, как уже можно предварительно заметить, может быть эквивалентен приложению пользовательского пространства, он может выполнять некоторые действия и не обязательно загружаться после этого резидентно. Первое отличие кода модуля от пользовательского приложения состоит в том, что ему позволено выполнять супервизорные действия: привилегированные команды, операции с аппаратными портами, реакцию на прерывания. Но такие привилегии означают и очень высокую стоимость ошибки в таком коде — он легко может разрушить операционную систему. Вторым важным качественным отличием будет то, что пользовательское приложение может позволить себе определенную небрежность в завершении: не освобождать динамически выделенную память, не закрывать открытые файловые дескрипторы — после завершения приложения такие действия за него «подчистит» ядро операционной системы. Следы же некорректных действий модуля останутся на все время жизни системы (до ее перезагрузки) — выделенная модулю, но не освобожденная им память, например, останется на все это время как область с потерянными путями доступа, и не может быть освобождена.

- Глава 2 -
Архитектура и вокруг...

Эти правила, язык знаков 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»

Чтобы ясно понимать, чем является модуль для ядра, необходимо вернуться к рассмотрению того, как пользовательские процессы взаимодействуют с возможностями ядра, что представляют собой системные вызовы и какие интерфейсы возникают в этой связи от пользователя к ядру или к модулям ядра, предоставляющим услуги ядра пользовательскому процессу.

В этой главе мы временно отвлечемся от ядра и рассмотрим, что происходит именно в пользовательском приложении, и то, что связывает его с ядром. Те, кто спешат перейти непосредственно к написанию модулей ядра, могут безболезненно пропустить эту главу.

Ядро: монолитное и микроядро

...передача сообщений как фундаментальная операция ОС — это просто упражнение в компьютерной мастурбации. Это может показаться приятным, но на самом деле вы ничего не сделаете.

Линус Торвальдс

Исторически все операционные системы, начиная от самых ранних (или 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 еще 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), известные также как клиент-серверные операционные системы или системы с обменом сообщениями — к этому классу относятся, например: QNX, MINIX 3, HURD, семейство ядер L4.

В микроядерной архитектуре все услуги для прикладного приложения система (микроядро) обеспечивает, отсылая сообщения (запросы) соответствующим сервисам (драйверам, серверам, менеджерам ресурсов...), которые, что самое важное, выполняются не в супервизорном пространстве ядра (в одном из низших колец защиты, не обязательно пользовательском). В этом случае не возникает никаких проблем с динамической реконфигурацией системы «на ходу» 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 одним своим существованием оправдывает существование микроядерной архитектуры для сверхнадежных областей применения. Но это уже совсем другая история, а сегодня мы занимаемся исключительно Linux.

В макроядерной архитектуре все услуги для прикладного приложения выполняют отдельные ветки кода внутри ядра (в адресном 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 ее безопасности и устойчивости). Для Linux это технология модулей ядра, которая появляется, начиная с ядра 0.99 (1992 г.), благодаря Питеру Мак-Дональду (Peter MacDonald).

С потребительской точки зрения сама операционная система пользователю... «почти не нужна». Если не считать того, что она является для потребителя фактически библиотекой большого числа (несколько сотен) общеупотребимых функций, предоставляемых *системными вызовами*. В противном случае все такие функции пользователю пришлось бы реализовывать в своем коде.

Кроме этой главной услуги для пользователя, ядро Linux обеспечивает еще ряд задач, но они завуалированы внешне и малозаметны конечному потребителю:

- управление ресурсами: файловыми системами, оперативной памятью, сетевым трафиком...
- поддержка службы времени;
- обслуживание периферии: обработка прерываний, операции ввода/вывода;
- арбитраж процессоров между параллельными потоками в многозадачной операционной системе.

Мы частично коснемся всех этих сервисов ядра, но позже, в следующих разделах.

Траектория системного вызова

Основным предназначением ядра любой операционной системы, вне всякого сомнения, является обслуживание системных вызовов по запросам из выполняющихся в системе процессов (операционная система занимается этим, скажем, 99,9% 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 нормальное последовательное выполнение команд и передающей управление коду режима супервизора (привилегированного режима)[1]. Это обычно некоторая команда программного прерывания — в зависимости от архитектуры процессора исторически это были команды с мнемониками типа: `svc`, `emt`, `trap`, `int` и подобными[2]. Если для конкретики проследить архитектуру Intel *x*86, то это традиционно команда программного прерывания с различным вектором — интересно сравнить, как это делают самые разнородные системы (табл. 2.1).

Таблица 2.1. *Команды программного прерывания различных систем*

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

Я специально добавил в таблицу и 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 отличающаяся 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.

---

[1] Теоретически реализовать системный вызов можно при помощи любого исключения, хоть при помощи деления на 0. Главное — это передача управления ядру.

[2] Ранее, еще на 80386, как утверждается, это делалось просто выполнением любой заведомо неверной инструкции — это был самый быстрый способ сделать системный вызов. Для этого обычно применялась бессмысленная и неверная инструкция `lock nop`, после исполнения которой процессором вызывался обработчик неверной инструкции. Так было более 20 лет назад, и, говорят, этим приемом обрабатывались системные вызовы в корпорации Microsoft.

Примечание

Начиная с некоторого времени (утверждается, что это примерно относится к началу 2008 года, или ко времени версии Windows XP Service Pack 2), многие операционные системы (Windows, Linux) при выполнении системного вызова от использования программного прерывания `int` перешли к реализации системного вызова (и возврата из него) через новые команды процессора `sysenter`/`sysexit`. Это было связано с заметной потерей производительности Pentium IV при классическом способе системного вызова и желанием из коммерческих побуждений эту производительность восстановить любой ценой. Но принципиально нового ничего не произошло: ключевые параметры перехода (CS, SP, IP) теперь загружаются не из памяти, а из специальных внутренних регистров MSR (Model Specific Registers) с предопределенными (0x174, 0x175, 0x176) номерами (из большого их общего числа), куда предварительно эти значения записываются, опять же, специальной новой командой процессора `wmsr`... В деталях это громоздко, реализационно — производительно, а по сути происходит то, о чем где-то сказали: «вектор прерывания теперь забит в железо, и процессор помогает нам быстрее перейти с одного уровня привилегий на другой». Еще позже AMD предложила для своей модели (которую мы сейчас знаем как *x*86_64) аналогичную по смыслу пару команд: `syscall`/`sysret`.

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

- PowerPC обладает специальной процессорной инструкцией `sc` (system call), регистр `r3` загружается номером системного вызова, а параметры загружаются последовательно в регистры от `r4` по `r8`;
- платформа ARM64 также имеет специальную инструкцию `syscall` для осуществления системного вызова, номер системного вызова загружается в `raw` регистр, а параметры — в `rdi`, `rsi`, `rdx`, `r10`, `r8` и `r9`.

Этих примеров достаточно, чтобы показать, что на любой интересующей вас платформе картина сохраняется качественно совершенно одинаковая при полном различии в деталях.

Библиотечный и системный вызов из процесса

Теперь мы готовы перейти к более детальному рассмотрению прохождения системного вызова в Linux. Там этот экскурс нужен, в общем, только для того, чтобы в дальнейшем ориентироваться: *что и где найти*!

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

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

Часть (весьма значительная) вызовов обслуживается непосредственно *внутри* библиотеки, вообще не требуя никакого вмешательства ядра, — пример тому `sprintf()` (или все строковые POSIX-функции вида `str*()`). Другая же 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 четко регламентирует группы вызовов, относя библиотечные API к *секции 3* руководств `man`. Хорошим примером тому может служить целая группа функций для запуска дочернего процесса: `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`:

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

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

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

Описания *системных* вызовов (в отличие от библиотечных) отнесены к *секции 2* руководств `man`. Системные вызовы далее преобразовываются в вызов ядра функцией `syscall()`, первым параметром которого будет номер требуемого системного вызова — например: `NR_execve`. Для конкретности еще один пример — вызов `printf(string)`, где `char *string` будет трансформироваться в `write(1, string, strlen(string))`, который далее — в `sys_call(__NR_write,...)`, и еще далее — в `int 0x80` (полный код такого примера будет показан далее).

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

```
$ man man
...
В таблице ниже показаны номера справочных разделов и описание их содержимого.
       1   Исполняемые программы или команды оболочки (shell)
       2   Системные вызовы (функции, предоставляемые ядром)
       3   Библиотечные вызовы (функции, предоставляемые программными библиотеками)
       4   Специальные файлы (обычно находящиеся в каталоге /dev)
       5   Форматы файлов и соглашения, например о /etc/passwd
       6   Игры
       7   Разное (включает пакеты макросов и соглашения), например man(7), groff(7)
       8   Команды администрирования системы (обычно запускаемые только суперпользователем)
       9   Процедуры ядра [нестандартный раздел]
```

Таким образом, в подтверждение 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 здесь:

```
$ 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 задается значением number
            с заданными аргументами. Символьные константы для системных вызовов можно найти
            в заголовочном файле (sys/syscall.h).
...
```

Вот образцы констант некоторых хорошо известных системных вызовов для архитектуры Intel (начало таблицы в качестве примера):

```
$ head -n10 /usr/include/asm/unistd_32.h
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6

$ head -n10 /usr/include/asm/unistd_64.h
#ifndef _ASM_X86_UNISTD_64_H
#define _ASM_X86_UNISTD_64_H 1

#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
```

И такие же, аналогичные, определения в 32-битной архитектуре ARM (новый интерфейс EABI):

```
$ head -n10 /usr/include/arm-linux-gnueabihf/asm/unistd-eabi.h
#ifndef _ASM_UNISTD_EABI_H
#define _ASM_UNISTD_EABI_H

#define __NR_restart_syscall (__NR_SYSCALL_BASE + 0)
#define __NR_exit (__NR_SYSCALL_BASE + 1)
#define __NR_fork (__NR_SYSCALL_BASE + 2)
#define __NR_read (__NR_SYSCALL_BASE + 3)
#define __NR_write (__NR_SYSCALL_BASE + 4)
#define __NR_open (__NR_SYSCALL_BASE + 5)
#define __NR_close (__NR_SYSCALL_BASE + 6)
```

Кроме `syscall()`, в Linux поддерживается (скорее рудиментно) и другой механизм системного вызова — `lcall7()`, устанавливая шлюз системного вызова так, чтобы поддерживать стандарт iBCS2 (Intel Binary Compatibility Specification), благодаря чему на *x*86 и Linux может выполняться *бинарный код*, подготовленный ранее для операционных систем FreeBSD, Solaris/86, SCO UNIX. Но больше мы этот механизм совместимости упоминать не будем (он не привносит ничего принципиально иного).

Таблицы системных вызовов для разной архитектуры могут *существенно отличаться* даже для близких родственных архитектур — сравните реализацию сетевых (сокетных) вызовов в Intel 32- и 64-бит. В 32-бит все сокетные операции реализуются через один-единственный системный вызов (мультиплексированная, старая реализация):

```
#define __NR_socketcall 102
```

Все сокетные операции здесь передаются как единый системный вызов с кодом 102, который завершается в ядре вызовом, где первым параметром передается код сокетной операции, а вторым — указатель на массив параметров вызова[3]:

```
/* obsolete: net/socket.c */
asmlinkage long sys_socketcall(int call, unsigned long __user *args);
```

А для 64-битной архитектуры все сокетные операции реализуются уже как раздельные системные вызовы:

```
#define __NR_socket 41
#define __NR_connect 42
#define __NR_accept 43
#define __NR_sendto 44
#define __NR_recvfrom 45
#define __NR_sendmsg 46
```

[3] Сам автор этой схемы пишет: «Данная методика кажется довольно уродливой (rather ugly) на первый взгляд, при сравнении с современными методами объектно-ориентированного программирования, но есть и определенная простота в нем. Она также хранит данные компактно, что улучшает попадание в кеши. Единственная проблема заключается в том, что выборка должна быть выполнена вручную, а это означает, что здесь легко выстрелить себе в ногу».

```
#define __NR_recvmsg 47
#define __NR_shutdown 48
#define __NR_bind 49
#define __NR_listen 50
#define __NR_getsockname 51
#define __NR_getpeername 52
#define __NR_socketpair 53
#define __NR_setsockopt 54
#define __NR_getsockopt 55
```

Показательно различие как в технике вызова, так и в числе задействованных в сетевых операциях системных вызовов.

Системные вызовы `syscall()` в Linux *на процессоре x86* (32-бит) изначально выполнялись 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, а входные аргументы размещаются в других регистрах процессора (таким образом, системному вызову может быть передано до 6 аргументов последовательно через регистры `ebx`, `ecx`, `edx`, `esi`, `edi` и `ebp`), после чего вызывается инструкция `int 0x80`. В тех относительно редких случаях, когда системному вызову необходимо передать *большее количество* аргументов (например, `mmap`), то они размещаются в структуре, адрес на которую передается в качестве первого аргумента (`ebx`). Результат возвращается в регистре `eax`, а стек вообще не используется. Системный вызов `syscall()`, попав в ядро, всегда попадает в таблицу `sys_call_table` и далее переадресовывается по индексу (смещению) в этой таблице на величину первого параметра вызова `syscall()` — номера требуемого системного вызова. Уже в архитектуре *x*86_64 (дальнейшем развитии *x*86) использование возможности `int 0x80` остается только для очень ограниченных случаев (статическая линковка и другие ограничения), но независимо от того, каким механизмом передается управление (парой команд `sysenter`/`sysexit`, или парой `syscall`/`sysret`, или другим из ряда используемых) оно *всегда* попадает в таблицу ядра `sys_call_table`!

В любой другой поддерживаемой Linux/GCC аппаратной платформе (из многих) результат будет аналогичный: системный вызов `syscall()` будет «доведен» до команды программного прерывания (вызова ядра), применяемой на этой платформе, — команды `EMT`, `TRAP` или подобной.

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

- архитектура 32-бит Intel *x*86 (**http://lxr.free-electrons.com/source/arch/?v=3.0**) — самое окончание таблицы:

```
...
.long sys_prlimit64             /* 340 */
.long sys_name_to_handle_at
.long sys_open_by_handle_at
.long sys_clock_adjtime
.long sys_syncfs
.long sys_sendmmsg              /* 345 */
.long sys_setns
```

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

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

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

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

Последний (по номеру) системный вызов *для этой версии ядра* (3.0) с именем `sys_setns` имеет номер (а это и полный размер таблицы, и *число* системных вызовов в архитектуре) соответственно: 346 — для 32-битной архитектуры Intel *x*86, 308 — для 64-битной архитектуры AMD/Intel и 375 — для архитектуры ARM. И так мы можем проследить размер и состав таблицы системных вызовов (селектора) для всех архитектур, поддерживаемых Linux. Кстати, здесь же мы можем посмотреть, какие и сколько архитектур поддерживает Linux:

```
$ ls -w80 /usr/src/linux-headers-`uname -r`/arch
alpha  arm64  h8300     Kconfig     mips     openrisc  riscv  sparc       x86
arc    c6x    hexagon   m68k        nds32    parisc    s390   um          xtensa
arm    csky   ia64      microblaze  nios2    powerpc   sh     unicore32
$ ls /usr/src/linux-headers-`uname -r`/arch | wc -w
27
```

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

Возьмите на заметку еще и то, что для некоторых архитектур в этом списке имя — это только родовое название, которое объединяет много частных технических реа-

лизаций, часто несовместимых друг с другом. Вот, например, сколько для общей архитектуры ARM существует частных архитектур производителя:

```
$ ls /usr/src/linux-headers-`uname -r`/arch/arm
boot               mach-dove          mach-nomadik   mach-tegra
common             mach-ep93xx        mach-npcm      mach-uniphier
crypto             mach-exynos        mach-nspire    mach-ux500
include            mach-footbridge    mach-omap1     mach-versatile
Kconfig            mach-gemini        mach-omap2     mach-vexpress
Kconfig.assembler  mach-highbank      mach-orion5x   mach-vt8500
Kconfig.debug      mach-hisi          mach-oxnas     mach-zynq
Kconfig-nommu      mach-imx           mach-pxa       Makefile
kernel             mach-integrator    mach-qcom      mm
lib                mach-iop32x        mach-rda       net
mach-actions       mach-ixp4xx        mach-realtek   nwfpe
mach-alpine        mach-keystone      mach-realview  plat-omap
mach-artpec        mach-lpc18xx       mach-rockchip  plat-orion
mach-asm9260       mach-lpc32xx       mach-rpc       plat-pxa
mach-aspeed        mach-mediatek      mach-s3c       plat-versatile
mach-at91          mach-meson         mach-s5pv210   probes
mach-axxia         mach-milbeaut      mach-sa1100    tools
mach-bcm           mach-mmp           mach-shmobile  vdso
mach-berlin        mach-moxart        mach-socfpga   vfp
mach-clps711x      mach-mstar         mach-spear     xen
mach-cns3xxx       mach-mv78xx0       mach-sti
mach-davinci       mach-mvebu         mach-stm32
mach-digicolor     mach-mxs           mach-sunxi
```

Ну и в завершение рассмотрим код приложения, который выполняет три системных вызова через *непрямой системный вызов* — `syscall()` (вызывающая часть `main` элементарна и не показана). Мы специально используем вызовы с различающимися параметрами, как по числу, так и по типу:

mpsys.c

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

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

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

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

Запись, использующая `syscall()`, будет корректно компилироваться в любой из поддерживаемых целевых платформ. В чем недостаток такой записи и почему так не следует писать? А дело в том, что в ней уже полностью исключен контроль параметров вызова `syscall()` как по их числу, так и по типам — прототип `syscall()` описан как функция с переменным числом параметров. И именно стандартная библиотека GCC в своей реализационной части вводит взаимно однозначное соответствие каждого `syscall()` определенному библиотечному системному вызову со строго оговоренным прототипом. Библиотечная эквивалентная реализация будет выглядеть так:

mplib.c

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

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

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

Под капотом системного вызова

Все, что еще ниже системного вызова `syscall()`, независимо от того, каким механизмом этот системный вызов доводит управление до передачи ядру (`int 0x80`, или

пара команд sysenter/sysexit, или пара syscall/sysret), — это уже внутренняя «кухня», то, что под капотом, и то, куда разработчику, даже модулей к ядру, обращаться не следует[4], да и нужды на то нет. Прослеживание этого пути имеет смысл в том, чтобы полностью представлять сквозной путь функционального вызова библиотеки — через системный вызов[5] и до таблицы ядра sys_call_table.

Вот ассемблерный код, использующий int 0x80:

int80.s

```
section     .text
global      _start

_start:

    mov     edx,len
    mov     ecx,msg
    mov     ebx,1               ; file descriptor (stdout)
    mov     eax,4               ; system call number (sys_write)
    int     0x80                ; call kernel

    mov     eax,1               ; system call number (sys_exit)
    int     0x80                ; call kernel

section     .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

Сборка[6] в режиме эмуляции режима *x*32 (для обратной совместимости):

```
$ nasm -f elf int80.s -o int80.32.o
$ ld -m elf_i386 int80.32.o -o int80.32
$ file int80.32
int80.32: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked,
not stripped
```

И сборка в расширенном режиме, в рамках ограничений использования в 64-битном режиме:

---

[4] Разработчики ядра убедительно просят не использовать непосредственно эти механизмы, а при необходимости определять новый syscall(). Они мотивируют это тем, что на уровне ниже syscall() не гарантируется сохранение и совместимость существующих механизмов в последующих версиях.

[5] Есть очень хорошее описание этого на качественном уровне: «Эволюция системных вызовов архитектуры x86» [49]. Иллюстрирующие текст ассемблерные коды заимствованы оттуда.

[6] Программу ассемблера nasm вам, скорее всего, придется дополнительно установить стандартным образом в своем дистрибутиве — в составе одноименного пакета в стандартном репозитории.

```
$ nasm -f elf64 int80.s -o int80.64.o
$ ld int80.64.o -o int80.64
$ file int80.64
int80.64: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
$ ls -l int80*
...
-rwxrwxr-x 1 olej olej  8688 июн 12 22:00 int80.32
-rw-rw-r-- 1 olej olej   624 июн 12 21:59 int80.32.o
-rwxrwxr-x 1 olej olej  8912 июн 12 22:02 int80.64
-rw-rw-r-- 1 olej olej   864 июн 12 22:01 int80.64.o
-rw-rw-r-- 1 olej olej   444 июн 11 18:54 int80.s
```

Проверка выполнением:

```
$ ./int80.32
Hello, world!
$ ./int80.64
Hello, world!
```

И один из основных механизмов (`syscall/sysret`), используемых в режиме 64-бит, как он выглядит:

syscall.s

```
section     .text
global      _start

_start:

    mov     rdx,len                         ;message length
    mov     rsi,msg                         ;message to write
    mov     rdi,1                           ;file descriptor (stdout)
    mov     rax,1                           ;system call number (sys_write)
    syscall

    mov     rax,60                          ;system call number (sys_exit)
    syscall

section     .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

Сборка:

```
$ nasm -f elf64 syscall.s -o syscall.o
$ ld syscall.o -o syscall.64
$ file syscall.64
syscall.64: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
```

```
$ ./syscall.64
Hello, world!
```

Отслеживание системного вызова в процессе

Где размещен код, являющийся интерфейсом к системному вызову Linux? Особенно, если выполняемый процесс скомпилирован из языка, отличного от C... Подобные особенности легко отследить, если собрать два сравнительных приложения (на C и C++) по принципу «проще не бывает» (см. папку tools/prog_sys_call в сопровождающем книгу файловом архиве):

prog_c.c

```
#include <stdio.h>

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

prog_cc.cc

```
#include <iostream>

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

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

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

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

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

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

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

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

Различия программ пространств ядра и пользователя

Того, что мы достигли в обсуждении, уже достаточно, чтобы подвести некоторые итоги и систематизировать основные отличия от пространства пользовательских процессов, которые нас будут подстерегать в пространстве ядра [25].

- Это самое главное различие для программиста, пишущего код: ядро *не имеет* **доступа** ни к каким библиотекам языка C (как, собственно, и к любым другим библиотекам любого другого языка). Причин на то много, и их обсуждение не входит в наши планы. А как следствие: ядро оперирует со *своим собственным* набором API, отличающимся от POSIX API (набором функций, их наименованиями, прототипами, параметрами...). Мы это видели на примере идентичных по смыслу, но различающихся вызовов `printk()` и `printf()`. И хотя мы будем встречаться довольно часто с *якобы идентичными* функциями (`sprint()`, `strlen()`, `strcat()` и многими другими), то это только внешняя *видимость совпадения*. Эти функции реализуют ту же функциональность, но это *дубликатная* реализация — подобный код реализуется и размещается в разных местах: для POSIX API — в составе библиотек, а для модулей — непосредственно в составе ядра.

  Возьмем на заметку, что у этих двух эквивалентных реализаций будут и различная авторская (если можно так сказать) принадлежность, и время обновления. Реализация в составе библиотеки `libc.so` организуется сообществом

GNU/FSF в комплексе проекта компилятора GCC, и новая версия библиотеки (и ее заголовочные файлы в /usr/include) устанавливается, когда вы обновляете версию *компилятора* (пакета gcc). А реализация версии такой же функции, но в составе ядра, по авторству принадлежит разработчикам ядра Linux, и будет обновляться, когда вы обновляете *ядро*, — будь то из репозитория используемого вами дистрибутива, или собирая его самостоятельно из исходных кодов. А эти обновления (*компилятора и ядра*), как понятно, являются не всегда коррелированными и синхронизированными. Это неочевидная и часто опускаемая из виду особенность!

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

Следствием этой двойственности является и то, что одна из основных трудностей программирования модулей как раз и есть *нахождение и выбор* адекватных средств API из набора плохо документированных и достаточно часто изменяющихся API ядра. Если по POSIX API существуют многочисленные обстоятельные справочники, то по именам ядра (вызовам и структурам данных) таких руководств нет. А общая размерность имен ядра (в /proc/kallsyms) превышает 100 000:

```
$ uname -r
5.4.0-117-generic
$ cat /proc/kallsyms | wc -l
144251
```

Из которых несколько десятков тысяч имен — это *экспортируемые* имена ядра:

```
$ cat /proc/kallsyms | grep " T " | wc -l
24465
$ cat /proc/kallsyms | grep " T " | head -n 10
ffffffff8ae00000 T startup_64
ffffffff8ae00000 T _stext
ffffffff8ae00000 T _text
ffffffff8ae00030 T secondary_startup_64
ffffffff8ae000e0 T verify_cpu
ffffffff8ae001e0 T start_cpu0
ffffffff8ae001f0 T __startup_64
ffffffff8ae00470 T pvh_start_xen
ffffffff8ae01000 T hypercall_page
ffffffff8ae02000 T __startup_secondary_64
```

Большинство механизмов ядра по своей функциональности сильно напоминают дуальные им, но гораздо лучше известные механизмы POSIX, однако специфика исполнения их в ядре (и еще историческая преемственность) накладывает отпечаток на форму API, делая вызовы различающимися как по наименованию, так и

по формату. Иногда очень помогает отслеживание аналогичных вызовов пространств пользователя и ядра — примеры только некоторых из них собраны в табл. 2.2 (см. *разд. «Обсуждение»* в конце этой главы), и они красноречиво говорят сами за себя.

- Отсутствие защиты памяти. Если обычная программа предпринимает попытку некорректного обращения к памяти, ядро аварийно завершит процесс, послав ему сигнал `SIGSEGV`. И на этом все заканчивается — система аварийно завершит процесс. Но если ядро предпримет попытку некорректного обращения к памяти, результаты могут быть куда менее контролируемыми, и с высокой вероятностью это вызовет тут же крах всей системы (часто даже не успеющей создать посмертную диагностику). К тому же ядро не использует замещение страниц: каждый байт, используемый в ядре, — это один байт физической памяти.
- В ядре нельзя использовать вычисления с плавающей точкой. Активизация режима вычислений с плавающей точкой требует (при переключении контекста) сохранения и восстановления регистров устройства поддержки вычислений с плавающей точкой (FPU) — помимо других рутинных операций. Вряд ли в модуле ядра могут понадобится вещественные вычисления, но если такое и случится, то их нужно эмулировать через целочисленные вычисления (для таких целей существует множество библиотек, откуда вы можете *заимствовать* код).
- Фиксированный стек — область адресного пространства, в которой выделяются локальные переменные. Локальные переменные — это все переменные, объявленные внутри блока, начинающегося левой открывающей фигурной скобкой, и не имеющие ключевого слова `static`. Стек в пользовательском режиме настраиваемый и большой. Стек в режиме ядра ограничен по размеру (во время компиляции ядра) и не может быть изменен. Исторически стек ядра состоит из двух страниц, что обычно подразумевает, что это 8 Кбайт на 32-битных архитектурах и 16 Кбайт на 64-битных. Размер этот фиксированный и абсолютный. Свой отдельный стек имеет каждый поток выполнения или поток ядра, кроме того, свой отдельный стек в ядре имеют процедуры обработки прерываний ISR.

  Исходя из всего этого, в коде ядра нужно крайне осмотрительно использовать (или не использовать) конструкции, расточающие пространство стека: рекурсию, передачу параметром структуры или возвращаемого значения из функции как структуры, объявление крупных локальных структур внутри функций и подобных им.
- Последнее, пожалуй, немаловажное различие состоит в том, что в прикладном программировании вы вряд ли станете использовать инструментарий устаревших версий, — например, компилятор GCC версии 4 или 6... при наличии под рукой:

```
$ gcc --version
gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- Это же относится к любому инструментарию, применяемому в прикладном программировании, — происходит активное обновление версий. Напротив, что касается ядра, то в обиходе (в разных областях применения) реально находятся ядра, начиная от 2.6.32–2.6.38 и до самых последних на сегодня 5.4–5.19 — их различие по времени ввода в эксплуатацию составляет 10–13 лет! И разработчики (модулей и приложений) часто вынуждены адаптировать свой код под механизмы API, различающиеся на десятилетие, в зависимости от техусловий проекта, в котором они работают. А поэтому нужно понимать и применять эти механизмы во всем спектре вариабельности их от версии к версии. Мы не можем просто взять и забыть «...как это делалось в версии 3.14»!

Интерфейсы модуля

Модуль ядра является некоторым согласующим элементом потребностей в пространстве пользователя с возможностями, обеспечивающими эти потребности в пространстве ядра. Модуль (код модуля) может иметь (и пользоваться ими) набор предоставляемых интерфейсов как в сторону взаимодействия с монолитным ядром Linux (с кодом ядра, с API ядра, структурами данных, ...), так и в сторону взаимодействия с пользовательским пространством (пользовательскими приложениями, пространством файловых имен, реальным оборудованием, каналами обмена данными, ...). Поэтому удобно раздельно рассматривать механизмы взаимодействия модуля в направлении пользователя (наружу) и в направлении механизмов ядра (внутрь).

Взаимодействие модуля с уровнем пользователя

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

- **Диагностика из модуля** (в системный журнал) `printk()`:
  - осуществляет вывод в *текстовую консоль* (или даже в графический терминал при определенных настройках!);
  - осуществляет вывод в файл журнала /var/log/messages, /var/log/kern.log, /var/log/dmesg;
  - содержимое файла журнала можно дополнительно посмотреть командой `dmesg`;
  - это основное средство диагностики, а часто и средство отладки.

  Конечно, сам вызов `printk()` — это такой же вызов API ядра (интерфейс вовнутрь), как, к примеру, скажем, `strlen()`. Но эффект, результат, производимый таким вызовом, *наблюдается в пространстве пользователя* (вовне). Кроме того, это настолько значимый интерфейс к модулю, что мы должны его здесь выде-

лить из числа прочих вызовов API ядра и отметить как отдельный специфический интерфейс.

- **Копирование данных** в программном коде между пользовательским адресным пространством и пространством ядра (выполняется *только по инициативе модуля*). Это самая важная часть взаимодействия модуля с пользователем. Конечно, опять же, это все те же вызовы из числа API ядра, но эти несколько вызовов предназначены для узко утилитарных целей: обмена данными между адресным пространством ядра и пространством пользователя. Логика *любого* системного вызова, а все взаимодействия с ядром сводятся к системным вызовам, состоит в том, что: а) при поступлении запроса в ядро сразу выполняется `copy_from_user()` — копирование данных из пользовательского пространства, и б) после выполнения запроса выполняется (при необходимости) `copy_to_user` — копирование результатов в пользовательское пространство.

  Вот эти вызовы (мы их получим динамически из таблицы имен ядра, и только после этого посмотрим определения в заголовочных файлах):

```
$ sudo cat /proc/kallsyms | grep T | grep copy_to_user
...
ffffffff96316350 T _copy_to_user
...
$ sudo cat /proc/kallsyms | grep T | grep copy_from_user
...
ffffffff963162f0 T _copy_from_user
...
```

  Реализованные `inline` вызовы `copy_to_user()` и `copy_from_user()` являются вызовами API ядра для данных *произвольного размера*. Для скалярных данных (фиксированного размера: `u8–u64`) они просто копируют требуемый элемент данных (1–8 байт) между пространствами пользователя и ядра (в том или другом направлении). Для данных отличающегося размера (произвольных данных) эти вызовы, после требуемой проверки *доступности страницы данных*, вызывают обычный `memcpy()` между областями[7].

```
$ sudo cat /proc/kallsyms | grep T | grep put_user
...
c05a7264 T __put_user_1
c05a7280 T __put_user_2
c05a729c T __put_user_4
c05a72b8 T __put_user_8
...
$ sudo cat /proc/kallsyms | grep T | grep get_user
...
```

[7] Сказанное относится к процессорной архитектуре *x*86. В другой процессорной архитектуре конкретная реализация `copy_*_user()` может быть совершенно другой, на что и указывает каталог размещения этих определений: `asm/uaccess.h`.

```
c05a62e0 T __get_user_1
c05a62fc T __get_user_2
c05a6318 T __get_user_4
c05a6334 T __get_user_8
...
```

> ***Пояснение***
>
> Я сознательно здесь и далее показываю вывод команд в некоторых случаях в 64-битной системе *x*86, а в других случаях в 32-битной системе ARM или *x*86 для того, чтобы мы привыкали к этому и различали, как адреса ядра выглядят в системах разной разрядности, и к какой системе относится диагностика.

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

Определения всех API этой группы можно найти в <asm/uaccess.h> (**https://elixir.bootlin.com/linux/v5.4/source/include/asm-generic/uaccess.h**) — все они реализованы `inline` и доступны для рассмотрения. Прототипы имеют вид (восстановленный в вид, какой они имели бы для функциональных вызовов — с типизированными параметрами):

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

Каждый вызов этой группы API возвращает нулевое значение в качестве признака успешной операции либо исходно переданное значение параметра `n` (положительное целое) — как признак недоступности области для копирования. В комментариях утверждается, что передача коротких порций данных через `put_user()` и `get_user()` будет осуществляться быстрее.

Ранее обсуждались наиболее часто упоминаемые вызовы обмена пространств ядра и пользователя. Там же определялся еще ряд менее употребляемых вызовов (их конкретный перечень может меняться от версии ядра). Они предназначены для работы с нуль-терминальными символьными строками, учитывая, что в значительной части случаев обмениваемые данные — это именно символьные строки:

---

[8] Почему такая асимметрия, я сказать не готов…

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

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

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

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

```
int fd = open(/dev/XXX, O_RDWR);
int n = read(fd, ...);
n = write(fd, ...);
close(fd);
```

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

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

Информация в /proc (и в /sys, о которой сказано далее) представляется только в *символьном* отображении, даже если это отображение единичного числового значения.

- **Взаимодействие через файлы** (имена) системы /sys (файловая система sysfs). Эта файловая система подобна (по назначению) /proc, но возникла заметно *позже*[9]. Первоначально система /sys была создана только для конфигурирования и управления периферийными устройствами (совместно с дуальной к ней подсистемой пространства пользователя udev), в том числе и устройствами горячего подключения. Но к настоящему времени ее возможности стали намного шире. Считается, что ее функциональность выше /proc, и она во многих качествах будет со временем заменять /proc. Некоторой отличительной особенностью информации в /proc и в /sys на сегодня является то (но это больше традиция и привычки, ничего более), что в /proc текстовая информация чаще представлена многострочным содержимым, таблицами, а в /sys — это чаще символьное представление одиночных значений:

```
$ cat /proc/partitions
major minor  #blocks  name

   1        0       4096 ram0
   1        1       4096 ram1
   1        2       4096 ram2
   1        3       4096 ram3
 179        0   15558144 mmcblk0
 179        1   15396864 mmcblk0p1
 254        0     251768 zram0
 254        1      51200 zram1
$ cat /sys/module/battery/parameters/cache_time
1000
```

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

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

[9] Подсистема /proc — это общая идеология систем UNIX, постулированная на заре UNIX (ранние 70-е). Подсистема /sys — это специфика Linux, родившаяся в процессе развития Linux.

```
3: wlo1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DORMANT
                                                              group default qlen 1000
    link/ether 34:23:87:d6:85:0d brd ff:ff:ff:ff:ff:ff
```

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

Взаимодействие модуля с ядром

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

```
# 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[10]. С каждым экспортированным ядром или любым модулем именем соотносится адрес, он и служит для связывания при загрузке модуля, использующего такое имя. Это основной механизм взаимодействия модуля с ядром.

Динамически формируемый (после загрузки) список имен ядра находится в файле /proc/kallsyms. Но в этом файле: а) более 100К строк и б) далеко не все содержащиеся там имена доступны модулю для связывания:

```
$ cat /proc/kallsyms | wc -l
151165
```

Для того чтобы разрешить первую проблему, нам необходимо бегло пользоваться (для фильтрации по самым замысловатым критериям) такими инструментами ана-

[10] Именно поэтому при изменениях в ядре (например, в сборке с другими конфигурационными параметрами) адреса имен в `/proc/kallsyms` «поплывут», и вызов из модуля, не соответствующего ядру сигнатурой (собранный вне этого ядра), произведет вызов «в молоко» и приведет к краху системы.

лиза регулярных выражений, как grep, awk (gawk), sed, perl или им подобными. Ключ ко второй нам дает 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
```

Здесь важнейшее имя ядра sys_call_table (таблица системных вызовов) известно в таблице имен, но не экспортируется ядром и недоступно для связывания коду модулей (мы еще детально вернемся к этому вопросу).

> ***Примечание***
>
> Имя sys_call_table может присутствовать в /proc/kallsyms в одних дистрибутивах и отсутствовать в других (я наблюдал первое в Fedora и множестве других и второе — в CentOS). Это имя вообще экспортировалось ядром до версий ядра 2.5 и могло напрямую быть использовано в коде, но такое состояние дел было признано небезопасным к ядру 2.6, и после этого оно не экспортируется.

В первую очередь разработчика модулей интересуют *экспортируемые* имена ядра, которых обычно на порядок меньше их общего числа:

```
$ cat /proc/kallsyms | grep " T " | wc -l
24465
```

Относительно API ядра нужно учитывать следующее: эти функции реализованы в ядре и при совпадении многих из них по форме с вызовами стандартной библиотеки C или системными вызовами (например, всё тем 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.

Разработчики ядра не связывают себя особенно требованиями совместимости снизу вверх, в отличие от очень жестких ограничений на пользовательские API, налагаемые стандартом POSIX. Поэтому API ядра достаточно произвольно меняются даже

от одной *подверсии* ядра к другой. Они плохо документированы (по крайней мере, в сравнении с документацией POSIX-вызовов пользовательского пространства). Детально изучать их приходится только по исходным кодам Linux, комментариям в коде и по кратким текстовым файлам заметок в дереве исходных кодов ядра.

Коды ошибок

Коды ошибок API ядра в основной массе — это те же коды ошибок, прекрасно известные по пространству пользователя. Определены они в source/include/uapi/asm-generic (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»*, т. к. 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.

Загрузка модулей

Утилита `insmod` получает *имя файла модуля* и пытается загрузить его без проверок взаимосвязей, как это показано далее. Утилита `modprobe` сложнее — ей 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.d(5)`).

Получив имя модуля, утилита `modprobe` по содержимому файла :

```
$ ls -l /lib/modules/`uname -r`/*.dep
-rw-r--r-- 1 root root 611114 июн  8 02:28 /lib/modules/5.4.0-117-generic/modules.dep
```

пытается установить зависимости запрошенного модуля — модули, от которых зависит запрошенный, будут загружаться утилитой прежде него. Файл зависимостей modules.dep формируется (обновляется) 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 включает 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* рассматриваемый (например, модуль `ubifs` зависим от двух модулей: `ubi` и `mtd`) и б) полные пути к файлам всех модулей. После этого загрузить модули не составляет труда, и непосредственно для этой работы включается (по каждому модулю последовательно) утилита `insmod`.

> ***Примечание***
>
> Если загрузка модуля производится непосредственно утилитой `insmod` с указанием ей имени файла модуля, то утилита никакие зависимости не проверяет, а если обнаруживает неразрешенное имя — просто 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 (счетчик ссылок использования модуля больше нуля), то выгрузка модуля не произойдет, 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.

Автоматическая загрузка модулей

На сегодня загрузка подавляющего большинства необходимых на этапе эксплуатации модулей выполняется udev автоматически, поэтому нет необходимости специально помещать имена модулей для загрузки в какой-либо файл конфигурации. Это же относится и к разработанным вами новым модулям. Выполняется все это на основе анализа идентификаторов PCI, USB-идентификаторов, идентификаторов

DMI или аналогичных триггеров, закодированных в самих модулях ядра. Фактически большинство современных модулей ядра уже подготовлены для автоматической загрузки.

Однако в некоторых случаях может потребоваться статически загрузить дополнительный модуль во время процесса загрузки компьютера. Чаще всего это характерно для модулей, не являющихся драйверами реальных или псевдоустройств. Для этого используются статические конфигурации загружаемых модулей.

Модули ядра могут быть явно указаны в файлах в каталоге /etc/modules-load.d/ для загрузки их средствами systemd во время включения компьютера. Каждый файл конфигурации имеет имя вида /etc/modules-load.d/*.conf и просто содержит список имен модулей ядра для загрузки, разделенных символами новой строки (пустые строки и строки, чей первый непробельный символ # или ; игнорируется как комментарий):

```
$ ls /etc/modules-load.d/
cups-filters.conf  modules.conf
$ cat /etc/modules-load.d/cups-filters.conf
# Parallel printer driver modules loading for cups
# LOAD_LP_MODULE was 'yes' in /etc/default/cups
lp
ppdev
parport_pc
```

Кроме этого каталога, статические конфигурационные файлы модулей (в разных дистрибутивах) могут помещаться в каталоги /run/modules-load.d и /usr/lib/modules-load.d — проверьте их на своем компьютере:

```
$ ls /run/modules-load.d/
ls: невозможно получить доступ к '/run/modules-load.d/': Нет такого файла или каталога
$ ls /usr/lib/modules-load.d
fwupd-msr.conf  fwupd-redfish.conf
$ cat /usr/lib/modules-load.d/fwupd-msr.conf
msr
$ lsmod | grep msr
intel_rapl_msr         20480  0
intel_rapl_common      24576  1 intel_rapl_msr
msr                    16384  0
```

Такая техника статической конфигурации загружаемых модулей относится, естественно, не к разработке модулей (чему посвящена эта книга), а к последующему этапу эксплуатации после завершения такой разработки.

Запрет загрузки (черный список модулей)

Иногда нужно явно запретить загрузку модуля из числа выбираемых средствами udev для загрузки. Чаще всего это возникает при необходимости загрузки новой версии модуля или новой модификации поддерживаемого модулем чипа, при этом загрузку оригинального модуля (часто с тем же именем) нужно запретить.

Достигается это включением *имени* запрещаемого модуля в новый или существующий файл *.conf в каталог /etc/modprobe.d с предшествующим словом-определением blacklist:

```
$ ls /etc/modprobe.d -w100
alsa-base.conf                  blacklist-firewire.conf      blacklist-rare-network.conf
amd64-microcode-blacklist.conf  blacklist-framebuffer.conf   dkms.conf
blacklist-ath_pci.conf          blacklist-modem.conf         intel-microcode-blacklist.conf
blacklist.conf                  blacklist-oss.conf           iwlwifi.conf
```

> ***Пояснение***
>
> Префикс blacklist в *имени* конфигурационного файла *не является* определяющим — это только напоминание назначения файла. Определяющим является *строка* такого файла, начинающаяся словом blacklist.

В любом таком файле может определяться как один запрещаемый модуль, так и большее их число (каждый модуль отдельной строкой).

```
$ cat /etc/modprobe.d/amd64-microcode-blacklist.conf
# The microcode module attempts to apply a microcode update when
# it autoloads.  This is not always safe, so we block it by default.
blacklist microcode
```

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 (см. папку chdev/cdev в сопровождающем книгу файловом архиве). Более того, в этом модуле, если не указано явно значение параметра, для него устанавливается его умалчиваемое значение (динамически определяемый системой старший номер устройства), а если параметр указан — то принудительно устанавливается заданное значение, даже если оно и недопустимо с точки зрения системы. Этот фрагмент выглядит так:

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

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

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

Обработка входных параметров модуля обеспечивается макросами (описаны в <linux/moduleparam.h>) — вот основные (там же есть еще ряд мало употребляемых),

причем два из них приводятся с полным определением через другие макросы (что добавляет понимания):

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

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

> ***Примечание***
>
> Последним параметром `perm` указаны права доступа (например, `S_IRUGO | S_IWUSR`), относящиеся к имени параметра, отображаемому в подсистеме /sys, и 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 определен и массив ранее отмеченных типов (макрос `module_param_array()`).

Вот пример, показывающий большинство приемов использования параметров загрузки модуля (см. папку tools/parms в сопровождающем книгу файловом архиве):

mod_params.c

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

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

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

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

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

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

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

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

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

module_init(mod_init);
```

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

- отсутствие резервирования памяти для символьного параметра `sparam`[11];
- и динамический размер параметра-массива `aparam` (динамически изменяющийся после загрузки модуля).

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

Притом что этот динамический размер *не может* превысить статически зарезервированную максимальную размерность массива (такая попытка вызывает ошибку).

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

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

```
# sudo insmod mod_params.ko
insmod: ERROR: could not insert module ./mod_params.ko: Operation not permitted
$ dmesg | tail -n8
[14562.245812] ======================================
[14562.245816] iparam = 0
[14562.245818] nparam = 0
[14562.245820] bparam = 1
[14562.245822] sparam = (null)
[14562.245824] cparam =  {0}
[14562.245828] aparam [ 5 ] =  0  0  0  0  0
[14562.245830] ======================================
# insmod mod_params.ko iparam=3 nparam=4 bparam=1 sparam=str1 cparam=str2 aparam=5,4,3
insmod: ERROR: could not insert module mod_params.ko: Operation not permitted
$ dmesg | tail -n8
[15049.389328] ======================================
[15049.389336] iparam = 3
[15049.389338] nparam = 4
[15049.389340] bparam = 0
[15049.389342] sparam = str1
[15049.389345] cparam = str2 {4}
[15049.389348] aparam [ 3 ] =  5  4  3
[15049.389350] ======================================
```

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

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

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

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

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

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

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

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

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

Так выглядит попытка присвоения нечислового значения числовому типу:

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

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

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

Конфигурационные параметры ядра[12]

Все, кто собирал ядро, знают, что прежде компиляции ядра мы конкретизируем великое множество его параметров. Эти параметры определяют состав и поведение ядра и не могут быть позже изменены. Собираемому коду модуля часто бывает необходимо определиться в том, с какими параметрами собиралось ядро, и получить доступ к этим параметрам.

Параметры в ядре

При сборке ядра каждый из параметров (все они имеют символьный вид констант с именами вида `CONFIG_*`) может быть конфигурирован так:

- `y` — собирать такую возможность непосредственно в монолит ядра;
- `m` — собирать такую возможность как подгружаемый модуль;
- `n` — не включать такую возможность в сборку;
- `<строка>` — символьное значение, присваиваемое параметру.

Параметры, с которыми было собрано ядро, находятся *обычно* (если сборщик не запретил такую возможность) в каталоге `/boot` в текстовом файле с именем /boot/config-`uname -a`:

```
$ ls /boot/config-*
/boot/config-5.4.0-113-generic  /boot/config-5.4.0-117-generic
```

Например (некоторые реально очень важные определения):

```
$ grep CONFIG_X86_64 /boot/config-`uname -r`
CONFIG_X86_64=y
CONFIG_X86_64_SMP=y
CONFIG_X86_64_ACPI_NUMA=y
$ grep CONFIG_64BIT /boot/config-`uname -r`
CONFIG_64BIT=y
CONFIG_64BIT_TIME=y
$ grep CONFIG_COMPAT /boot/config-`uname -r` | grep -v ^#
CONFIG_COMPAT_32=y
CONFIG_COMPAT=y
CONFIG_COMPAT_FOR_U64_ALIGNMENT=y
CONFIG_COMPAT_OLD_SIGACTION=y
CONFIG_COMPAT_32BIT_TIME=y
CONFIG_COMPAT_BINFMT_ELF=y
CONFIG_COMPAT_NETLINK_MESSAGES=y
```

Но в некоторых дистрибутивах вы с удивлением обнаружите, что в каталоге /boot нет конфигурационных файлов (как и других, которые вы ожидаете). Вот, например, одноплатный Raspberry Pi с операционной системой (сборкой) Raspberry Pi OS:

[12] Тема подсказана одним из читателей рукописи.

```
$ inxi -Sxxx
System:
  Host: raspberrypi Kernel: 5.15.32-v7+ armv7l bits: 32 compiler: gcc v: 10.2.1
  Console: tty 2 DM: LightDM 1.26.0 Distro: Raspbian GNU/Linux 11 (bullseye)
$ grep CONFIG_PAGE_OFFSET /boot/config-`uname -r`
grep: /boot/config-5.15.32-v7+: Нет такого файла или каталога
$ ls /boot
bcm2708-rpi-b.dtb         bcm2710-rpi-zero-2.dtb    fixup4.dat    kernel.img
bcm2708-rpi-b-plus.dtb    bcm2710-rpi-zero-2-w.dtb  fixup4db.dat  LICENCE.broadcom
bcm2708-rpi-b-rev1.dtb    bcm2711-rpi-400.dtb       fixup4x.dat   overlays
bcm2708-rpi-cm.dtb        bcm2711-rpi-4-b.dtb       fixup_cd.dat  start4cd.elf
bcm2708-rpi-zero.dtb      bcm2711-rpi-cm4.dtb       fixup.dat     start4db.elf
bcm2708-rpi-zero-w.dtb    bcm2711-rpi-cm4s.dtb      fixup_db.dat  start4.elf
bcm2709-rpi-2-b.dtb       bootcode.bin              fixup_x.dat   start4x.elf
bcm2710-rpi-2-b.dtb       cmdline.txt               issue.txt     start_cd.elf
bcm2710-rpi-3-b.dtb       config.txt                kernel7.img   start_db.elf
bcm2710-rpi-3-b-plus.dtb  COPYING.linux             kernel7l.img  start.elf
bcm2710-rpi-cm3.dtb       fixup4cd.dat              kernel8.img   start_x.elf
```

Это просто означает, что авторы дистрибутива (сборки) не сочли нужным помещать файл конфигурации в `/boot/*` — это их право. Но не отчаивайтесь! Во *всех* дистрибутивах (современных, по крайней мере) — там, где есть это в /boot/*, и там, где нет, *дубликат* конфигураций находится в каталоге текущего ядра /lib/modules/`uname -r`/build/.config:

```
$ grep CONFIG_CPU_32 /lib/modules/`uname -r`/build/.config
CONFIG_CPU_32v6K=y
CONFIG_CPU_32v7=y
$ grep CONFIG_ARM= /lib/modules/`uname -r`/build/.config
CONFIG_ARM=y
```

Параметры которые вы конфигурировали как `n`, в конфигурационном файле будут закомментированы (`#` в первой позиции строки, и это позволяет с помощью `grep` отфильтровать только действующие параметры). Можем посмотреть, насколько велико число конфигурационных параметров:

```
$ cat /boot/config-`uname -r` | grep '=y' | wc -l
2562
$ cat /boot/config-`uname -r` | grep '=m' | wc -l
5304
$ cat /boot/config-`uname -r` | grep ^# | wc -l
1985
```

В заголовочных (хедер) файлах ядра, использующихся при сборке модулей, эти ядерные параметры ретранслируются в файл:

```
$ ls -l /lib/modules/`uname -r`/build/include/generated/autoconf.h
-rw-r--r-- 1 root root 294109 июн  2 02:07 /lib/modules/5.4.0-117-generic/build/include/
generated/autoconf.h
```

Он же[13]:

```
$ ls -l /usr/src/linux-headers-`uname -r`/include/generated/autoconf.h
-rw-r--r-- 1 root root 294109 июн  2 02:07 /usr/src/linux-headers-5.4.0-117-generic/include/
generated/autoconf.h
```

Этот же файл параметров (/lib/modules/`uname -r`/build/include) с точки зрения *умалчиваемого* каталога хедеров ядра: <generated/autoconf.h>[14] — доступен вместе с параметрами в коде модуля. Символические имена параметров в этом файле определяются как препроцессорные константы, имеющие вид: `CONFIG_*`. Все конфигурационные параметры ядра, определяемые в диалоге сборки, ретранслируются в этот файл следующим образом:

- те параметры ядра, которые не были выбраны в диалоге сборки (с ответом `n`), — просто *не попадают* в хедер-файл, естественно:

```
$ cat /boot/config-`uname -r` | grep CONFIG_KERNEL_GZIP
# CONFIG_KERNEL_GZIP is not set
$ cat /lib/modules/`uname -r`/build/include/generated/autoconf.h | grep CONFIG_KERNEL_GZIP
$
```

- для тех параметров ядра, которые были выбраны в диалоге сборки с ответом `y`, — соответствующие им конфигурационные параметры определены в файле <generated/autoconf.h> со значением 1:

```
$ cat /boot/config-`uname -r` | grep CONFIG_SWAP
CONFIG_SWAP=y
$ cat /lib/modules/`uname -r`/build/include/generated/autoconf.h | grep CONFIG_SWAP
#define CONFIG_SWAP 1
```

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

```
$ cat /lib/modules/`uname -r`/build/include/generated/autoconf.h | grep
CONFIG_INIT_ENV_ARG_LIMIT
#define CONFIG_INIT_ENV_ARG_LIMIT 32
$ cat /lib/modules/`uname -r`/build/include/generated/autoconf.h | grep
CONFIG_SECURITY_TOMOYO_POLICY_LOADER
#define CONFIG_SECURITY_TOMOYO_POLICY_LOADER "/sbin/tomoyo-init"
```

- сложнее с теми параметрами ядра, которые были выбраны в диалоге сборки с ответом `m` (собирать такую возможность как подгружаемый модуль). При этом символическая константа с именем `CONFIG_XXX`, соответствующим конфигураци-

---

[13] В некоторых дистрибутивах (Fedora 35, например) этот путь `/usr/src/linux-headers-*` может отличаться и, например, выглядеть так: `/usr/src/kernels/linux-headers-*`, поэтому правильный путь — это определять заголовочные файлы именно через `/lib/modules`.

[14] В некоторых дистрибутивах и сборках (Fedora 35, например) схема помещения кофигурационных параметров в заголовочные файлы может существенно различаться.

онному параметру, не будет определена в файле <generated/autoconf.h>, но будет определена *другая* символическая константа — с суффиксом _MODULE: CONFIG_XXX_MODULE, и значение ее будет, естественно, 1:

```
$ cat /boot/config-`uname -r` | grep =m | grep CONFIG_NET_ACT_BPF
CONFIG_NET_ACT_BPF=m
$ cat /lib/modules/`uname -r`/build/include/generated/autoconf.h | grep CONFIG_NET_ACT_BPF
#define CONFIG_NET_ACT_BPF_MODULE 1
```

Несколько примеров:

```
$ cat /boot/config-`uname -r` | grep CONFIG_KERNEL_GZIP
# CONFIG_KERNEL_GZIP is not set

$ cat /boot/config-`uname -r` | grep CONFIG_SWAP
CONFIG_SWAP=y
$ cat /lib/modules/`uname -r`/build/include/generated/autoconf.h | grep CONFIG_SWAP
#define CONFIG_SWAP 1

$ cat /boot/config-`uname -r` | grep CONFIG_NET_ACT_BPF
CONFIG_NET_ACT_BPF=m
$ cat /lib/modules/`uname -r`/build/include/generated/autoconf.h | grep CONFIG_NET_ACT_BPF
#define CONFIG_NET_ACT_BPF_MODULE 1

$ grep CONFIG_64BIT /usr/src/linux-headers-`uname -r`/include/generated/autoconf.h
#define CONFIG_64BIT_TIME 1
#define CONFIG_64BIT 1
$ grep CONFIG_OUTPUT_FORMAT /usr/src/linux-headers-`uname -r`/include/generated/autoconf.h
#define CONFIG_OUTPUT_FORMAT "elf64-x86-64"
```

> ***Пояснение***
>
> Не ищите какого-то смысла в том, почему из многих тысяч параметров выбраны именно эти и что они означают, — показанные здесь параметры выбраны почти наугад из числа тех, которые позволят нам наглядно продемонстрировать в примерах, как это выглядит в разных архитектурах.

Параметры в модуле

Мы рассмотрели, как конфигурационные параметры выглядят в ядре. Теперь мы можем сосредоточиться на том, как они выглядят со стороны подгружаемых модулей.

Разрабатываемый вами модуль может динамически считывать параметры ядра, к которому он подгружается, и реконфигурироваться в зависимости от набора обнаруженных параметров. Естественно, что параметры конфигурации ядра являются для модуля «только для чтения» — модуль не может на них никак влиять. Но модуль может анализировать такие параметры как *окружение*, в котором ему предстоит работать, и в зависимости от этого устанавливаться в системе тем или иным

образом (или вообще не устанавливаться при отсутствии каких-то возможностей, критических с позиции разработчика модуля).

Вот пример модуля, демонстрирующего доступность конфигурационных параметров ядра в коде модуля (см. папку tools/config в сопровождающем книгу файловом архиве):

config.c

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

static int __init config_init(void) {
#if defined(CONFIG_SMP)
        printk("CONFIG_SMP = %d\n", CONFIG_SMP);
#else
        printk("CONFIG_SMP не определено\n");
#endif

#if defined(CONFIG_64BIT)
        printk("CONFIG_64BIT = %d\n", CONFIG_64BIT);
#else
        printk("CONFIG_64BIT не определено\n");
#endif

#if defined(CONFIG_OUTPUT_FORMAT)
        printk("CONFIG_OUTPUT_FORMAT = %s\n", CONFIG_OUTPUT_FORMAT);
#else
        printk("CONFIG_OUTPUT_FORMAT не определено\n");
#endif

#if defined(CONFIG_NET_ACT_BPF)
        printk("CONFIG_NET_ACT_BPF = %d\n", CONFIG_NET_ACT_BPF);
#elif defined(CONFIG_NET_ACT_BPF_MODULE)
        printk("CONFIG_NET_ACT_BPF_MODULE = %d\n", CONFIG_NET_ACT_BPF_MODULE);
#else
        printk("CONFIG_NET_ACT_BPF* не определено\n");
#endif
        return -1;
}

module_init(config_init);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");
```

Здесь модуль *статически* скомпилируется в зависимости от *препроцессорных* констант параметров в той системе, где он компилируется (или не скомпилируется вовсе из-за отсутствия некоторых зависимостей).

Но модуль может и *динамически* исследовать значения параметров конфигурации ядра и адаптировать свое поведение в зависимости от них (см. пример в той же папке):

configr.c

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

static int __init config_runtime_init(void) {
        printk("CONFIG_SMP %sопределено\n",
               IS_ENABLED(CONFIG_SMP) ? "" : "не ");

        printk("CONFIG_64BIT %sопределено\n",
               IS_ENABLED(CONFIG_64BIT) ? "" : "не ");

        if (IS_ENABLED(CONFIG_NET_ACT_BPF))
                printk("CONFIG_NET_ACT_BPF определено\n");
        if (IS_ENABLED(CONFIG_NET_ACT_BPF_MODULE))
                printk("CONFIG_NET_ACT_BPF_MODULE определено\n");
        if (!IS_ENABLED(CONFIG_NET_ACT_BPF))
                printk("CONFIG_NET_ACT_BPF* не определено\n");

        return -1;
}

module_init(config_runtime_init);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");
```

Сборка модулей элементарна, и это не интересно (Makefile приведен в папке примеров). А вот результаты интересно рассмотреть при компиляции одного и того же кода модулей в разных системах (архитектура процессора и разрядность):

- Intel 64-бит:

```
$ uname -a
Linux R420 5.4.0-117-generic #132-Ubuntu SMP Thu Jun 2 00:39:06 UTC 2022 x86_64 x86_64
x86_64 GNU/Linux
$ sudo insmod config.ko
insmod: ERROR: could not insert module config.ko: Operation not permitted
$ dmesg | tail -n4
[10169.865430] CONFIG_SMP = 1
[10169.865432] CONFIG_64BIT = 1
[10169.865434] CONFIG_OUTPUT_FORMAT = elf64-x86-64
[10169.865434] CONFIG_NET_ACT_BPF_MODULE = 1
$ sudo insmod configr.ko
insmod: ERROR: could not insert module configr.ko: Operation not permitted
```

```
$ dmesg | tail -n4
[10204.438786] CONFIG_SMP определено
[10204.438788] CONFIG_64BIT определено
[10204.438789] CONFIG_NET_ACT_BPF определено
[10204.438789] CONFIG_NET_ACT_BPF_MODULE определено
```

- AMD 32-бит:

```
$ uname -a
Linux orangepione 5.15.25-sunxi #22.02.1 SMP Sun Feb 27 09:23:25 UTC 2022 armv7l GNU/Linux
$ sudo insmod config.ko
[sudo] пароль для olej:
insmod: ERROR: could not insert module config.ko: Operation not permitted
$ dmesg | tail -n4
[488594.849785] CONFIG_SMP = 1
[488594.849811] CONFIG_64BIT не определено
[488594.849816] CONFIG_OUTPUT_FORMAT не определено
[488594.849820] CONFIG_NET_ACT_BPF_MODULE = 1
$ sudo insmod configr.ko
insmod: ERROR: could not insert module configr.ko: Operation not permitted
$ dmesg | tail -n4
[488627.271958] CONFIG_SMP определено
[488627.271984] CONFIG_64BIT не определено
[488627.271990] CONFIG_NET_ACT_BPF определено
[488627.271995] CONFIG_NET_ACT_BPF_MODULE определено
```

Возможность анализа значений параметров конфигурации ядра важна не только для модификации поведения кода модуля, но и, что еще важнее, для предотвращения компиляции модуля в среде неподобающей ему конфигурации ядра (target platform not supported)!

Подсчет ссылок использования

Одно из самых важных (и очень запутанных по описаниям) понятий из сферы модулей есть подсчет ссылок использования модуля. Счетчик ссылок является *внутренним полем* структуры описания модуля и, вообще-то говоря, слабо доступен пользователю непосредственно (что, может, и хорошо). При загрузке модуля начальное значение счетчика ссылок нулевое. При загрузке любого следующего модуля, который использует имена (импортирует), экспортируемые этим модулем, счетчик ссылок этого модуля инкрементируется. Модуль, счетчик 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 уже *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 счетчика ссылок `5`, и дальше в выводе следует указание имен пяти модулей, на него ссылающихся. До тех пор, пока эти пять модулей не будут удалены из системы, удалить модуль `i2c_core` будет невозможно.

В чем состоит отмеченная ранее 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 счетчик ссылок. Но эти макросы остались где-то в недрах ядер 2.4. В ядре 2.6 их место заняли *функциональные* вызовы (определенные в <linux/module.h>):

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

В качестве параметра всех этих вызовов, как правило, передается 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 счетчика ссылок из собственного модуля. Делать это нужно крайне обдуманно, поскольку если мы увеличим счетчик и симметрично его позже не уменьшим, то мы вообще никогда не сможем выгрузить модуль (до перезагрузки системы), — это один из путей возникновения в системе «перманентных» модулей. Другая возможность их возникновения — модуль, успешно инсталлирующийся, но не имеющий в коде функции завершения. В некоторых случаях может оказаться нужным динамически изменять счетчик ссылок, препятствуя *на время* возможности выгрузки модуля. Это актуально, например, в функциях, реализующих операции `open()` (увеличиваем счетчик обращений) и `close()` (уменьшаем, восстанавливаем счетчик обращений) для драйверов устройств, — иначе станет возможна выгрузка модуля, обслуживающего еще открытое устройство, а следующие обращения (из процесса пользовательского пространства) к открытому дескриптору устройства будут направлены в неинициализированную память!

И здесь возникает очередная путаница (которую можно наблюдать и по коду некоторых модулей): во многих источниках рекомендуется инкрементировать из собственного кода модуля счетчик использований при открытии устройства и декрементировать счетчик при его закрытии. Это было актуально, но с некоторой версии ядра (я не смог установить, с какой) это отслеживание делается *автоматически* при выполнении открытия/закрытия устройства (а ваши манипуляции создадут

опасные избыточные значения счетчика ссылок). Примеры этого, поскольку мы пока не готовы к усвоению многих деталей такого кода, будут детально приведены позже — при рассмотрении множественного открытия для устройств (каталог dev/mopen).

Обсуждение

Из этой части рассмотрения мы можем уже вынести следующие заключения:

- программирование модулей ядра Linux — это не только создание драйверов специфических устройств, но и вообще гораздо более широкая область: динамическое расширение функциональности ядра, добавление возможностей, которыми ранее ядро не обладало;
- программирование модулей ядра Linux принципиально не отличается во многом от программирования в пространстве пользовательских процессов. Однако для его осуществления невозможно привлечь существующие в пространстве пользователя POSIX API и использовать библиотеки, поэтому в пространстве ядра предлагаются дублирующие («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 (табл. 2.2).

Таблица 2.2. *Примеры вызовов API POSIX и ядра, несущих эквивалентную функциональность*

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

- одна из основных трудностей программирования модулей состоит в *нахождении* и выборе слабо документированных и изменяющихся API ядра. В этом нам значительную помощь оказывают динамические и статические таблицы разрешения имен ядра и заголовочные файлы исходных кодов ядра, с которымм мы должны постоянно сверяться на предмет актуальности ядерных API текущей версии используемого нами ядра. Если по POSIX API существуют многочисленные обстоятельные справочники, то по именам ядра (вызовам и структурам данных) таких руководств нет. А общая размерность *имен ядра* (/proc/kallsyms) уже превышает 100 000, из которых свыше 20 000 — это *экспортируемые* имена ядра (как будет подробно показано далее). Рассмотреть описательно такой объем не представляется возможным, а единственный путь, который обещает успех в освоении техники модулей ядра, видится в наличии максимального объема законченных примеров работающих модулей «на все случаи жизни». Таким построением изложения мы и воспользуемся.

- Глава 3 -
Инструментальное окружение

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. Те, кто уверенно себя чувствует в консольных командах Linux, особенно относящихся к действиям относительно ядра, могут сразу пропустить эту главу.

Основные команды

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

- загрузка модуля в систему:

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

- удаление модуля из системы:

```
# rmmod hello_printk
```

- загрузка модуля, ранее *инсталлированного* в систему, и всех модулей, требуемых его зависимостями (как может делаться постоянная инсталляция модуля в систему будет показано далее):

```
# modprobe hello_printk
```

- вывод информации о *файле* модуля:

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

> ***Примечание***
>
> Команды `rmmod` и `modprobe` требуют указания *имени модуля*, а команды `insmod` и `modinfo` — указания *имени файла* модуля.

- список установленных модулей и их зависимости:

```
$ lsmod
```

- обновление в системе зависимостей модулей, необходимых для корректной работы команды `modprobe`:

```
# depmod
```

- вывод системного журнала (в том числе и сообщений модулей):

```
$ dmesg
```

- вывод системного журнала (в том числе и сообщений модулей). Формат отличается от `dmesg` — это две альтернативные формы в разных дистрибутивах, чтение требует прав root:

```
# cat /var/log/messages
# cat /var/log/kern.log
```

- команда, дающая нам список имен объектного файла, — в частности, имен модуля с глобальной сферой видимости (тип `T`) и имен, импортируемых модулем для связывания (тип `U`):

```
$ nm mobj.ko
```

- детальный анализ объектной структуры модуля:

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

- еще один инструмент анализа объектной структуры модуля:

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

Системные файлы

Здесь приводится краткий перечень системных файлов (псевдофайлов и путевых имен), на которые следует обратить внимание и с которыми нам предстоит много работать при отработке модулей:

- /proc/modules — динамически создаваемый (обновляемый) список модулей в системе. Именно отсюда черпает информацию команда `lsmod`. Считывать этот файл следует с правами root, иначе вместо адресных значений вы получите нулевые значения, — это относится и к большинству файлов, упоминаемых далее. Причина такого поведения связана с безопасностью ядра (доступа по адресам имен). Вот как это выглядит в 32-битной ARM системе:

```
$ wc -l /proc/modules
45 /proc/modules
$ tail /proc/modules
uio_pdrv_genirq 20480 0 - Live 0x00000000
display_connector 20480 0 - Live 0x00000000
uio 16384 1 uio_pdrv_genirq, Live 0x00000000
cpufreq_dt 20480 0 - Live 0x00000000
ip_tables 24576 0 - Live 0x00000000
x_tables 28672 1 ip_tables, Live 0x00000000
autofs4 36864 2 - Live 0x00000000
```

```
sunxi 16384 0 - Live 0x00000000
phy_generic 20480 2 sunxi, Live 0x00000000
gpio_keys 20480 0 - Live 0x00000000
$ sudo tail /proc/modules
uio_pdrv_genirq 20480 0 - Live 0xbf850000
display_connector 20480 0 - Live 0xbf845000
uio 16384 1 uio_pdrv_genirq, Live 0xbf836000
cpufreq_dt 20480 0 - Live 0xbf828000
ip_tables 24576 0 - Live 0xbf82f000
x_tables 28672 1 ip_tables, Live 0xbf820000
autofs4 36864 2 - Live 0xbf816000
sunxi 16384 0 - Live 0xbf811000
phy_generic 20480 2 sunxi, Live 0xbf806000
gpio_keys 20480 0 - Live 0xbf800000
```

- /boot/System.map-`uname -r` — статически созданный (при сборке ядра) файл содержит *статическую* таблицу всех имен ядра (не во всех сборках Linux присутствует такой файл — это определяется решением сборщика). Особый интерес представляют имена с обозначенным типом `T` (но не только они) — это экспортируемые имена функциональных вызовов (API ядра):

```
$ sudo grep ' T ' /boot/System.map-`uname -r` | tail
ffffffff82f1f17e T libata_transport_exit
ffffffff82f1f404 T ledtrig_usb_exit
ffffffff82f1f571 T xhci_debugfs_remove_root
ffffffff82f1f6b6 T rtc_dev_exit
ffffffff82f1f896 T watchdog_dev_exit
ffffffff82f1fca1 T rproc_exit_debugfs
ffffffff82f1fcb3 T rproc_exit_sysfs
ffffffff82f1fe56 T rfkill_handler_exit
ffffffff83600000 T __init_scratch_begin
ffffffff83a00000 T __init_scratch_end
```

- /proc/kallsyms — *динамически* создаваемый список имен ядра. Эта таблица содержит все имена из /boot/System.map-`uname -r`, но дополненные именами из загруженных модулей ядра (отмечены как `[...]`):

```
$ sudo grep ' T ' /proc/kallsyms | tail
ffffffffc035e610 T drm_release  [drm]
ffffffffc0368870 T drm_edid_get_monitor_name    [drm]
ffffffffc03890f0 T drm_agp_acquire      [drm]
ffffffffc036e6e0 T drm_gem_prime_export [drm]
ffffffffc03608a0 T drm_ioctl_kernel     [drm]
ffffffffc0389160 T drm_agp_release      [drm]
ffffffffc0366030 T drm_mode_get_hv_timing       [drm]
ffffffffc037cef0 T drm_vblank_restore   [drm]
ffffffffc01f6e40 T wmi_driver_unregister        [wmi]
ffffffffc01f6a80 T __wmi_driver_register        [wmi]
```

До некоторого времени адреса, объявляемые в /proc/kallsyms (динамические) и в /boot/System.map-`uname -r` (статические), совпадали по значениям. Но с 2005 года (с версии 2.6.12) в ядре Linux начал использоваться сначала простой, а затем и более сложный вариант ASLR (Address Space Layout Randomization, рандомизация размещения адресного пространства). Работа этого механизма заключается в том, что адрес загрузки и все имена ядра смещаются при загрузке относительно их значений (адреса) в /boot/System.map-`uname -r` на: а) фиксированную величину; б) которая случайным образом выбирается при загрузке и в) которая меняется от одной загрузки к следующей. Посмотрим сообщения уже рассматривавшегося модуля aslr.ko относительно адреса самого вызова `printk()`:

```
$ dmesg | tail -n6
[12173.441037]         %p             %px                %pK                int
...
[12173.441041]       d1756e2b    ffffffff8d88b926    ffffffff8d88b926      ffffffff8d88b926
...
```

И сравним:

```
$ sudo grep ' printk'$ /boot/System.map-`uname -r` | grep ' T '
ffffffff81a8b926 T printk
$ sudo grep ' printk'$ /proc/kallsyms | grep ' T '
ffffffff8d88b926 T printk
```

Значение в динамической таблице (/proc/kallsyms) совпадает с экспериментально считанным адресом размещения `printk()` и отличается от размещения этого имени в статической таблице на величину `BE00000`, и *на эту* величину будут смещены *все* адреса размещения при *этой загрузке* системы. Технология ASLR создана для усложнения эксплуатации нескольких типов уязвимостей.

Но нужно проявлять осторожность при толковании адресных результатов — не во всех сборках системы! Вот как это выглядит, например, в 32-битной ARM системе:

```
$ sudo grep ' _printk'$ /boot/System.map-`uname -r` | grep ' T '
c09c49a0 T _printk
$ sudo grep ' _printk'$ /proc/kallsyms | grep ' T '
c09c49a0 T _printk
```

Следующие интересующие нас псевдофайлы (из них мы получаем справочную информацию для проектирования модулей ядра и анализируем результаты загрузки своих модулей):

- /proc/slabinfo — динамическая детальная информация сляб-аллокатора (slab allocator) памяти (о нем мы будем говорить отдельно и подробно)[1];

[1] В литературе и электронных публикациях мне встречались самые разнообразные русскоязычные наименования для английского термина «slab», а именно: «слаб», «сляб», «слэб»... Но термин нужно в тексте как-то именовать, и ни одна из транскрипций не лучше других, но... Более устоявшимся, а кроме того, использующимся, помимо IT, в совершенно иной области — металлургии, является произношение «сляб», поэтому давайте использовать его. Здесь и далее.

- /proc/meminfo — сводная информация о использовании памяти в системе;
- /proc/devices — список драйверов устройств, встроенных в действующее ядро;
- /proc/dma — задействованные в текущий момент каналы DMA;
- /proc/filesystems — файловые системы, встроенные в ядро;
- /proc/interrupts — список задействованных в текущий момент прерываний;
- /proc/ioports — список задействованных в текущий момент портов ввода/вывода;
- /proc/version — версия ядра (из этого источника черпает информацию команда `uname`):

```
$ cat /proc/version
Linux version 5.15.32-v7+ (dom@buildbot) (arm-linux-gnueabihf-gcc-8 (Ubuntu/Linaro
8.4.0-3ubuntu1) 8.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #1538 SMP Thu Mar 31 19:38:48
BST 2022
$ uname -a
Linux raspberrypi 5.15.32-v7+ #1538 SMP Thu Mar 31 19:38:48 BST 2022 armv7l GNU/Linux
```

- /lib/modules/`uname -r`/build/include — важнейший для нас каталог, содержащий все хедер-файлы ядра (каталог по умолчанию) для включения определений в код модуля. Это эквивалент того, что мы имеем в пользовательском пространстве в каталоге /usr/include, без которого не обходится ни одна программа на C:

```
$ ls -w80 /lib/modules/`uname -r`/build/include
acpi         config  dt-bindings  kvm       media  pcmcia  scsi   target  vdso
asm-generic  crypto  generated    linux     misc   ras     soc    trace   video
clocksource  drm     keys         math-emu  net    rdma    sound  uapi    xen
```

Графика, терминал и текстовая консоль

Ряд авторов утверждают, что графическая подсистема (X11 или Wayland) не подходит как среда для разработки и отладки модулей ядра — для этого годится только текстовая *консоль* — хотя далее они тут же сами забывают эти свои нравоучения и демонстрируют примеры, явно выполняемые в *графическом терминале* графической подсистемы. То есть все подобные утверждения относятся, скорее, к области красивых народных легенд. Тем не менее нужно отчетливо представлять соотношение текстовых и графических режимов (интерфейсов пользователя) в Linux, их особенности и ограничения. Текстовая *консоль* и текстовый графический *терминал* (эмулятор терминала) в Linux — это не одно и то же, нужно понимать их различия, и знать, когда вам годится одно, а когда — другое.

Вывод (и ввод) на *терминал* (работающий в графической системе) проходит через множество промежуточных слоев, в отличие от *текстовой консоли*, и может значительно отличаться по поведению при работе с программами ядра[2]. По этой причине

[2] Напротив, в прикладных программах пространства пользователя текстовая консоль и графический терминал практически эквиваленты в своем поведении, но могут различаться в отношении языковой локализации.

отладка модулей в текстовой консоли все еще остается актуальной... Так что, в силу ее значимости для отработки программ ядра, есть смысл коротко остановиться на работе с текстовыми консолями.

Управление текстовыми консолями

Текстовая консоль — это одна из самых старых подсистем UNIX и Linux, поэтому за свое существование она сменила *много* разных систем инициализации и управления. Но сегодня практически во всех модификациях Linux для управления сервисами используется *подсистема systemd* [34] — вот ее сервис инициализации консоли:

```
$ ls -l /usr/lib/systemd/system/getty@.service
-rw-r--r-- 1 root root 1975 апр 21 15:54 /usr/lib/systemd/system/getty@.service
```

Вообще-то в Linux, так исторически повелось еще со времен ранних UNIX, может быть *до 64* текстовых консолей (`tty0 ... tty63`):

```
$ ls -w100 /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/tty1   /dev/tty18  /dev/tty26  /dev/tty34  /dev/tty42  /dev/tty50  /dev/tty59
/dev/ttyAMA0
/dev/tty10  /dev/tty19  /dev/tty27  /dev/tty35  /dev/tty43  /dev/tty51  /dev/tty6
/dev/ttyprintk
/dev/tty11  /dev/tty2   /dev/tty28  /dev/tty36  /dev/tty44  /dev/tty52  /dev/tty60
/dev/tty12  /dev/tty20  /dev/tty29  /dev/tty37  /dev/tty45  /dev/tty53  /dev/tty61
/dev/tty13  /dev/tty21  /dev/tty3   /dev/tty38  /dev/tty46  /dev/tty54  /dev/tty62
/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
```

Но и из этого числа *инициализированных* текстовых консолей в Linux обычно по умолчанию 7 — это традиция Linux, идущая еще от нескольких старых систем инициализации консолей:

```
$ sudo fgconsole
7
```

И более того, именно systemd (в отличие от старых систем инициализации) не инициализирует все семь консолей — как легко видеть, инициализированы только `tty1` и `tty7` (с графической сессией `Xorg`):

```
$ ps -Af | grep tty
root       483   465  0 мая30 tty7  00:25:37 /usr/lib/xorg/Xorg :0 -seat seat0 -auth
/var/run/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
root       485     1  0 мая30 tty1  00:00:00 /bin/login -f
root       486     1  0 мая30 ?     00:00:00 /sbin/agetty -o -p -- \u --keep-
baud 115200,57600,38400,9600 ttyAMA0 vt220
olej       529   485  0 мая30 tty1  00:00:00 -bash
olej     15468  2368  0 19:10 pts/4    00:00:00 grep --color=auto tty
```

Все остальные из семи умалчиваемых консолей инициализируются только непосредственно в момент первого переключения на них.

Часто случается, что при отработке в текстовой консоли модулей ядра, требующих такой отладки, этого числа консолей может оказаться недостаточно. Как увеличить число консолей? Для этого делаем инициализацию:

```
$ sudo systemctl start getty@tty9.service
$ sudo systemctl status getty@tty9
● getty@tty9.service - Getty on tty9
     Loaded: loaded (/lib/systemd/system/getty@.service; disabled; vendor preset: enabled)
     Active: active (running) since Sun 2022-06-12 18:30:00 EEST; 16min ago
       Docs: man:agetty(8)
             man:systemd-getty-generator(8)
             http://0pointer.de/blog/projects/serial-console.html
   Main PID: 70161 (agetty)
      Tasks: 1 (limit: 115828)
     Memory: 272.0K
     CGroup: /system.slice/system-getty.slice/getty@tty9.service
             └─70161 /sbin/agetty -o -p -- \u --noclear tty9 linux

июн 12 18:30:00 R420 systemd[1]: Started Getty on tty9.
```

В противовес не инициадизированным еще консолям:

```
$ sudo systemctl status getty@tty10
● getty@tty10.service - Getty on tty10
     Loaded: loaded (/lib/systemd/system/getty@.service; disabled; vendor preset: enabled)
     Active: inactive (dead)
       Docs: man:agetty(8)
             man:systemd-getty-generator(8)
             http://0pointer.de/blog/projects/serial-console.html
```

И убеждаемся в инициализации консоли сверх семи:

```
$ ps -Af | grep tty | grep tty9
root     70161       1  0 18:30 tty9    00:00:00 /sbin/agetty -o -p -- \u --noclear tty9 linux
olej     78056   24154  0 19:21 pts/5   00:00:00 grep --color=auto tty9
```

Общеизвестно, что между текстовыми консолями можно переключаться *одновременным* нажатием комбинаций клавиш <Ctrl>+<Alt>+<F*i*>, где *i* — номер функциональной клавиши: 1–12. Но как переключаться между консолями, когда их больше 12? Самый универсальный способ — команда смены виртуального терминала:

```
# chvt 9
```

Вот 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:

- строки 1 и 2 — это регистрации в текстовых *консолях* (`#2` и `#3` соответственно) под разными именами (`root` и `olej`);
- строка 3 — регистрация в графической консоли X11 (консоль `#7`, графический дисплей `0:0`);
- следующие 7 строк — 7 открытых графических *терминалов* внутри X11 (дисплей `:0`);
- последняя строка — одна удаленная *сессия* с регистрацией по SSH с сетевого хоста с именем `notebook`.

Коротко о компиляторе GCC

Исходный код модулей ядра в Linux, при всем богатстве языковых средств разработки в Linux для *прикладного* программирования, создается *исключительно* на языке C. Но, написав программный код модуля, его еще нужно и скомпилировать. Даже для одного только языка C в Linux могут использоваться, присутствуют и реально используются несколько различных компиляторов:

- GCC — основной компилятор Linux (из GNU-проекта GCC);
- компилятор `cc` из состава интегрированной среды разработки IDE Solaris Studio операционной системы Open Solaris (еще недавно фирмы Sun Microsystems, а на сегодня фирмы Oracle);
- активно развивающийся в рамках проекта LLVM компилятор Clang (кандидат для замены `gcc` в операционной системе FreeBSD, причина чему — лицензия);
- PCC (Portable C Compiler) — современная реализация компилятора, развиваемого еще с 70-х годов прошлого века, получившая новую жизнь в операционных системах NetBSD и OpenBSD.

Все указанные и еще некоторые из свободно развиваемых проектов могут использоваться в Linux с успехом (утверждается, что код, компилированный IDE Solaris Studio, в ряде случаев заметно производительнее компилированного GCC). Тем не менее вся эта альтернативность возможна только **в** *проектах пользовательского адресного пространства* — в программировании ядра и соответственно модулей

ядра сегодня применяется исключительно компилятор GCC. Причина этому — значительные синтаксические расширения GCC относительно стандартов языка C.

> ***Примечание***
>
> Существуют экспериментальные проекты по сборке ядра Linux компилятором, отличным от GCC. Встречаются сообщения о том, что компилятор Intel C имеет достаточную поддержку расширений GCC, чтобы компилировать ядро Linux. В 2013 году сообщалось, что Clang из проекта LLVM достиг такой степени совместимости, что им полностью были собраны ядро и все окружение Linux. Но при всех таких попытках пересборка может быть произведена лишь полностью, «с нуля» — начиная со сборки ядра и только потом переходя к сборке модулей. В любом случае ядро и модули должны собираться одним компилятором.

Начало GCC было положено Ричардом Столлманом, который реализовал первый вариант GCC в 1985 году на нестандартном и непереносимом диалекте языка Pascal. Позднее компилятор был переписан на языке Си Леонардом Тауэром и Ричардом Столлманом и выпущен в 1987 году как компилятор для проекта GNU (см. **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 системе команд множества (нескольких десятков) процессорных архитектур.

Достигается это его двухуровневый структурой. Верхний уровень выполняет синтаксический разбор исходного языка и генерацию *промежуточного кода RTL* (Register Transfer Language, язык регистрового переноса). Нижний уровень генерирует машинный код, который должен выполняться на указанной целевой машине. Этот процесс поддерживается со стороны утилит GNU:

- программой лексического анализатора `flex` (свободный аналог UNIX-утилиты `lex`);
- программой синтаксического парсера `bison` (свободный аналог UNIX-утилиты `yacc`).

Нижний уровень ассемблирования из RTL-кода может быть подключен к любому новому языку программирования.

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

```
$ gcc --version
gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

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
The following options control optimizations:
  -O<number>
  -Ofast
  -Og
  -Os
  -faggressive-loop-optimizations [enabled]
  -falign-functions                        [enabled]
  -falign-functions=                       16
  -falign-jumps                            [enabled]
  -falign-jumps=                           16:11:8
  -falign-labels                           [enabled]
  -falign-labels=                          0:0:8
  -falign-loops                            [enabled]
  -falign-loops=                           16:11:8
  -fassociative-math                       [disabled]
  -fassume-phsa                            [enabled]
  -fasynchronous-unwind-tables             [enabled]
  -fauto-inc-dec                           [enabled]
...
```

Для подтверждения того, что установки опций для разных уровней оптимизации (`-O0`, `-O2`, `-O3`, ...) различаются, и уточнения, в чем состоят эти различия, проделаем следующий эксперимент:

```
$ gcc -Q -O2 --help=optimizer > O2
$ gcc -Q -O3 --help=optimizer > O3
$ diff O2 O3
53c53
<   -fgcse-after-reload                    [disabled]
---
>   -fgcse-after-reload                    [enabled]
67c67
<   -finline-functions                     [disabled]
---
>   -finline-functions                     [enabled]
72c72
<   -fipa-cp-clone                         [disabled]
---
>   -fipa-cp-clone                         [enabled]
...
$ diff O2 O3 | grep "\-\-\-" | wc -l
14
```

Для этих опций — различие в 14 значениях!

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

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

GCC имеет значительные синтаксические расширения (такие, например, как инлайновые ассемблерные вставки или использование вложенных функций), не распознаваемые другими компиляторами языка C — именно поэтому альтернативные компиляторы вполне пригодны для сборки приложений, но малопригодны для пересборки ядра Linux и сборки модулей ядра.

Невозможно в паре 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 страницах, и оно издано в русском переводе [8], — его просто рекомендуется держать под рукой на рабочем столе в качестве справочника.

Ассемблер в Linux

Необходимо несколько слов сказать и об ассемблере. Язык ассемблера практически никогда не бывает нужен *прикладному* программисту (по крайней мере, программисту в Linux). Но разработчику модулей в отдельных случаях он может помочь, часто даже не столько для написания конечного кода, сколько для понимания того, с чем приходится иметь дело (в коде ядра Linux достаточно много ассемблерных инлайновых вставок), для экспериментов, для отладки и поиска тонких ошибок и неисправностей. В сложных случаях иногда бывает полезным изучить ассемблерный код, генерируемый компилятором GCC на промежуточном этапе компиляции, и убедиться, что... компилятор вас неправильно понял. В конечном счете вопрос о том, нужно ли разработчику знание ассемблера, должен разрешаться так: разработчик модулей не обязан писать на языке ассемблера, но ему желательно понимать, написанное на ассемблере... а уж тем более этот код различать.

В предыдущих редакциях этого текста здесь размещалось достаточно обстоятельное описание возможностей использования ассемблерного кода в Linux. Но сейчас я оставлю здесь лишь некоторые вводные замечания — очень краткие, но позволяющие интересующимся углубиться в этот предмет самостоятельно.

Как уже отмечалось, в некоторых сложных случаях иногда бывает нужно изучить ассемблерный код, генерируемый GCC (в приложении пользователя) на промежуточном этапе компиляции. Увидеть сгенерированный GCC ассемблерный код можно, компилируя командой с ключами:

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

Примечание

Посмотреть результат еще 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 (*x*86, PPC, MIPS, AVR, ARM, ...), поддерживаемых GCC, — но синтаксис записи этого кода будет радикально *различаться* для разных платформ (регистры, их число и обозначение, мнемоника команд ... все-все-все).

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

```
$ as --version
GNU ассемблер (GNU Binutils for Ubuntu) 2.34
Copyright (C) 2020 Free Software Foundation, Inc.
Эта программа является открытым программным обеспечением; вы можете
распространять ее согласно условиям GNU General Public License версии 3 или
более новой версии.
Эта программа не имеет абсолютно никаких гарантий.
Ассемблер настроен на цель x86_64-linux-gnu.
```

Или так (совершенно разные аппаратные платформы):

```
$ as --version
GNU ассемблер (GNU Binutils for Raspbian) 2.35.2
Copyright (C) 2020 Free Software Foundation, Inc.
Эта программа является открытым программным обеспечением; вы можете
распространять ее согласно условиям GNU General Public License версии 3 или
более новой версии.
Эта программа не имеет абсолютно никаких гарантий.
Ассемблер настроен на цель arm-linux-gnueabihf.
```

Помимо нативного ассемблера Linux `as`, вы можете установить другие популярные у любителей этого дела приложения ассемблеров:

```
$ nasm --version
NASM version 2.14.02
$ yasm --version
yasm 1.3.0
Compiled on Apr  1 2020.
Copyright (c) 2001-2014 Peter Johnson and other Yasm developers.
Run yasm --license for licensing overview and summary.
```

Устанавливаются они из одноименных пакетов стандартного репозитория вашего дистрибутива типовым для дистрибутива способом. Помимо минимального числа ассемблерных файлов (*.s или *.S), в коде (ядра) Linux гораздо шире используется такое специфическое расширение компилятора GCC, как инлайновые ассемблерные вставки — когда внутрь C-кода включается по необходимости несколько ассемблерных строк, компилируемых в монолите C-кода.

Но если вы вдруг решили поэкспериментировать в ассемблере в Linux, то прежде всего обратите внимание на такую вещь, как *нотация* записи ассемблерного кода...

Нотация AT&T

В отличие от нотации Intel (которую используют, например, все инструменты Microsoft, компилятор C/C++ Intel и многоплатформенный ассемблер NASM) [26, 27], ассемблер GCC использует синтаксическую нотацию AT&T.

> ***Примечание***
>
> Обоснование этого простое — все названные инструменты, ориентированные на нотацию Intel, используют ее исключительно к процессорам архитектуры *x*86. Но GCC является многоплатформенным инструментом, поддерживающим не один десяток аппаратных платформ, и ассемблерный код каждой из этих множественных платформ может быть записан в AT&T-нотации.

В нотации AT&T строка, записанная так:

```
movl %ebx, %eax           // EBX -> EAX
```

выглядит в Intel-нотации так:

```
mov eax, ebx              // EAX <- EBX
```

Основные принципы AT&T-нотации, принципиально отличающие ее от более привычной многим Intel-нотации (еще идущей из известной MS-DOS):

- порядок операндов (направление выполнения операции): `<Операция>`, `<Источник>`, `<Приемник>` (слева направо). В Intel-нотации порядок обратный (справа налево);
- 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: для различных процессорных архитектур это может быть самая различная запись — например: `%1`, `%2`, ... (VAX, Motorola 68000);
- названия констант начинаются с `$` и могут быть выражением. Например: `movl $1,%eax`;
- обязательное *явное* задание размеров операндов в суффиксах команд (последнем символе мнемоники команды): `b` — это `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`;
- значение без префикса означает адрес. Это еще один камень преткновения новичков. Просто следует запомнить, что:
  - `movl $123, %eax` — означает 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`;

- для косвенной адресации необходимо использовать круглые скобки. Например: `movl (%ebx), %eax` — загрузить в регистр `%eax` значение переменной, по адресу, находящемуся в регистре `%ebx`;
- SIB-адресация смещения (база, индекс, множитель).

Однострочные примеры:

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

А вот как выглядит последовательность ассемблерных инструкций для реализации системного вызова `exit(EXIT_SUCCESS)` на архитектуре *x*86:

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

Все, что записано в программе на ассемблере, будь это ассемблерный отдельно компилируемый файл или описываемые далее инлайновые ассемблерные вставки, — все это будет *платформенно зависимо*! Начиная с самых основ, с синтаксиса фрагмента — символьные обозначения регистров на каждой платформе будут различаться. Более того, близкие платформы, неразделяемые обычно на пользовательском уровне, — такие как *x*86 32-бит (IA-32) и 64-бит (IA-64), а также AMD64 — будут все между собой различаться в ассемблерной записи! Поэтому ассемблерный код более пригоден для изучения и иллюстрации происходящего, чем для практических целей. Если же это потребуется в рабочем проекте (что нужно тщательно продумать), то такие фрагменты прописываются раздельно для всех платформ, где проект может эксплуатироваться, и «обкладываются» соответствующими препроцессорными директивами `#ifdef`.

> ***Примечание***
>
> Во всех рассматриваемых примерах, здесь и везде далее, где для ясности понимания привлекается ассемблерный код, он будет записан в синтаксической нотации 32-битной платформы *x*86 — где иное не будет подчеркнуто явно.

Инлайновый ассемблер GCC

GCC Inline Assembly — встроенный ассемблер компилятора GCC, представляющий собой язык *макроописания* интерфейса компилируемого высокоуровнего кода с ассемблерной вставкой. Синтаксис инлайн-вставки в С-код — это оператор вида (в терминологии синтаксиса языка С — оператор, *отделяемый* от следующего символом `;`):

```
asm [volatile] ("команды и директивы ассемблера" /* как последовательность текстовых строк */
                  : [<выходные параметры>] : [<входные параметры>] : [<изменяемые параметры>]
                );
```

Здесь:

- `выходные параметры` — это запись того, как значения регистров будут записаны в выходные переменные С-кода *после* выполнения фрагмента. Например: `"=a"(res)` — содержимое регистра `%eax` будет записано в целочисленную переменную `res`;
- `входные параметры` — это запись того, как переменные С-кода будут занесены в регистры процессора *перед* выполнением фрагмента. Например: `"b"((long)(str))`, `"c"((long)(len))` — значение указателя (адрес) строки `str` будет загружено в регистр `%ebx`, а целочисленное значение длины строки `len` — в регистр `%ecx`;
- `изменяемые параметры` — это запись списка (через запятую) тех регистров, содержимое которых может портиться при выполнении фрагмента, — чтобы компилятор мог принять меры по их хранению. Например: `"%ecx", "%edi"` — два регистра процессора объявлены модифицируемыми в процессе выполнения фрагмента;
- `команды и директивы ассемблера` — это записанный в формате текстовой строки языка С фрагмент (может быть достаточно большой) программы, подлежащей макропреобразованию в язык ассемблера. Если это многострочный фрагмент программы, то его строки завершаются ограничителями `\n` или `\n\t`.

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

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

Примеры:

- запись нескольких строк инструкций ассемблера:

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

- выполнение системного вызова `write()` (только в 32-битной системе) — определение *функции*:

```
int write_call(int fd, const char* str, int len) {
   long __res;           // это еще С-определение
   __asm__ volatile ("int $0x80"
           : "=a" (__res)
           : "0"(__NR_write),"b"((long)(fd)),"c"((long)(str)),"d"((long)(len)));
   return (int) __res;   // а это уже С-оператор
}
```

- так в форме инлайнового включения в программе определяется *макровызов* с параметрами (произвольный в общем случае — умножения на 5 конкретно в показанном примере). Этим вводится определение нового функционального вызова, который реализуется позже из С-кода простой и привычной записью `times5(n, n)`(это макрос!):

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

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

Еще один, более сложный пример — запись в ядре Linux макроса системного вызова с четырьмя параметрами, как это было сделано в одной из предыдущих версий ядра (очень рекомендовал бы здесь остановиться на мгновение и проанализировать написанное, чтобы понять, какую «ядерную» смесь представляет собой код ядра Linux[3]):

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

Создание среды сборки модулей ядра

Ранее уже было вскользь сказано, что в свежеинсталлированной системе Linux у вас *может не быть* установлено инструментов для выполнения работ с модулями ядра (я встречал такое в некоторых из многих инсталлированных дистрибутивах). Начиная с того, что зачастую и сам компилятор GCC, и утилита `make` могут отсутствовать в системе... Мы проверим это так:

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

У вас должны присутствовать:

```
$ gcc --version
gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

[3] Спасает такой код от самораспада только непрерывное и массовое тестирование и «вылизывание» в течение более чем 30 лет.

```
$ make --version
GNU Make 4.2.1
Эта программа собрана для x86_64-pc-linux-gnu
Copyright (C) 1988-2016 Free Software Foundation, Inc.
Лицензия GPLv3+: GNU GPL версии 3 или новее <http://gnu.org/licenses/gpl.html>
Это свободное программное обеспечение: вы можете свободно изменять его и
распространять. НЕТ НИКАКИХ ГАРАНТИЙ вне пределов, допустимых законом.
```

Но сделанного нами мало — этого достаточно для прикладного программирования, но недостаточно для программирования модулей ядра. Нам нужно создать инфраструктуру для сборки модулей ядра (главным образом это заголовочные файлы ядра, но не только). Заголовочные файлы ядра *по умолчанию* не устанавливаются практически ни в одном дистрибутиве Linux... и это резонно — далеко не каждому пользователю системы понадобится пересобирать компоненты ядра, а места, бесцельно растранжиренного, это потребует немало:

```
$ sudo du -hs /lib/modules/`uname -r`
278M    /lib/modules/5.4.0-117-generic
```

В Debian-дистрибутивах (Debian, Ubuntu, Mint, ...) вам необходимо установить пакет:

```
$ apt show linux-headers-`uname -r`
Package: linux-headers-5.4.0-117-generic
Version: 5.4.0-117.132
Priority: optional
Section: devel
Source: linux
Origin: Ubuntu
Maintainer: Ubuntu Kernel Team <kernel-team@lists.ubuntu.com>
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Installed-Size: 15,0 MB
Provides: linux-headers, linux-headers-3.0
Depends: linux-headers-5.4.0-117, libc6 (>= 2.14), libelf1 (>= 0.142), libssl1.1 (>= 1.1.0)
Download-Size: 1.392 kB
APT-Manual-Installed: no
APT-Sources: http://ubuntu.volia.net/ubuntu-archive focal-updates/main amd64 Packages
Description: Linux kernel headers for version 5.4.0 on 64 bit x86 SMP
 This package provides kernel header files for version 5.4.0 on
 64 bit x86 SMP.
 .
 This is for sites that want the latest kernel headers.  Please read
 /usr/share/doc/linux-headers-5.4.0-117/debian.README.gz for details.
```

В итоге вы должны получить (проверить) состояние, подобное следующему:

```
$ aptitude search header | grep ^i
i A linux-headers-5.4.0-113 - Header files related to Linux kernel version 5.4.0
i A linux-headers-5.4.0-113-generic - Linux kernel headers for version 5.4.0 on 64 bit x86 SMP
i A linux-headers-5.4.0-117 - Header files related to Linux kernel version 5.4.0
```

```
i A linux-headers-5.4.0-117-generic - Linux kernel headers for version 5.4.0 on 64 bit x86 SMP
i  linux-headers-generic - Generic Linux kernel headers
```

Для тех, кто предпочитает (или в силу служебных корпоративных требований) RPM-дистрибутивы (Fedora, CentOS, RedHat, ...), вам необходимы дополнительные пакеты `kernel-headers.*` (обычно устанавливаются вместе с `kernel-devel.*`):

```
$ dnf info kernel-headers
Последняя проверка окончания срока действия метаданных: 0:02:44 назад, Вт 14 июн 2022 12:26:15.
Установленные пакеты
Имя          : kernel-headers
Версия       : 5.17.11
Выпуск       : 200.fc35
Архитектура  : x86_64
Размер       : 5.7 M
Источник     : kernel-headers-5.17.11-200.fc35.src.rpm
Репозиторий  : @System
Из репозитор : updates
Краткое опис : Header files for the Linux kernel for use by glibc
URL          : http://www.kernel.org/
Лицензия     : GPLv2
Описание     : Kernel-headers includes the C header files that specify the interface
             : between the Linux kernel and userspace libraries and programs.
             : The header files define structures and constants that are needed
             : for building most standard programs and are also needed for rebuilding
             : the glibc package.
```

И в итоге вы будете иметь что-то подобное:

```
$ dnf list installed '*kernel*'
Установленные пакеты
abrt-addon-kerneloops.x86_64                    2.14.6-9.fc35
kernel-core.x86_64                              5.17.11-200.fc35
kernel-core.x86_64                              5.17.12-200.fc35
kernel-core.x86_64                              5.17.13-200.fc35
kernel-devel.x86_64                             5.17.11-200.fc35
kernel-devel.x86_64                             5.17.12-200.fc35
kernel-devel.x86_64                             5.17.13-200.fc35
kernel-headers.x86_64                           5.17.11-200.fc35
kernel-modules.x86_64                           5.17.11-200.fc35
kernel-modules.x86_64                           5.17.12-200.fc35
kernel-modules.x86_64                           5.17.13-200.fc35
kernel-modules-extra.x86_64                     5.17.11-200.fc35
kernel-modules-extra.x86_64                     5.17.12-200.fc35
kernel-modules-extra.x86_64                     5.17.13-200.fc35
kernel-srpm-macros.noarch                       1.0-6.fc35
kernel-tools.x86_64                             5.17.11-200.fc35
kernel-tools-libs.x86_64                        5.17.11-200.fc35
libreport-plugin-kerneloops.x86_64              2.15.2-8.fc35
```

> ***Примечание***
>
> Обратите внимание, что суффикс подверсии `kernel-headers` отличается от `uname -r` — это означает, что хедеры не обновляются с каждым обновлением ядра. Но важно не это, а то, что такое различие может ввести в заблуждение при поиске нужной информации по ядру.

В любом случае (независимо от дистрибутива) вы должны получить и проверить *ссылку*:

```
$ ls -l /lib/modules/`uname -r`/build
lrwxrwxrwx 1 root root 40 июн  2 02:07 /lib/modules/5.4.0-117-generic/build -> /usr/src/linux-
headers-5.4.0-117-generic
```

Она в разных дистрибутивах может указывать на несколько различающиеся каталоги:

```
$ ls -l /lib/modules/`uname -r`/build
lrwxrwxrwx 1 root root 40 июн  6 18:10 /lib/modules/5.17.13-200.fc35.x86_64/build ->
/usr/src/kernels/5.17.13-200.fc35.x86_64
```

... но важно не это, а то, что такая ссылка *должна быть*, и она указывает на дерево каталогов, содержащих *всё* необходимое для сборки модулей ядра:

```
$ tree -L 1 /lib/modules/`uname -r`/build
/lib/modules/5.4.0-117-generic/build
├── arch
├── block -> ../linux-headers-5.4.0-117/block
├── certs -> ../linux-headers-5.4.0-117/certs
├── crypto -> ../linux-headers-5.4.0-117/crypto
├── Documentation -> ../linux-headers-5.4.0-117/Documentation
├── drivers -> ../linux-headers-5.4.0-117/drivers
├── fs -> ../linux-headers-5.4.0-117/fs
├── include
├── init -> ../linux-headers-5.4.0-117/init
├── ipc -> ../linux-headers-5.4.0-117/ipc
├── Kbuild -> ../linux-headers-5.4.0-117/Kbuild
├── Kconfig -> ../linux-headers-5.4.0-117/Kconfig
├── kernel
├── lib -> ../linux-headers-5.4.0-117/lib
├── Makefile -> ../linux-headers-5.4.0-117/Makefile
├── mm -> ../linux-headers-5.4.0-117/mm
├── Module.symvers
├── net -> ../linux-headers-5.4.0-117/net
├── samples -> ../linux-headers-5.4.0-117/samples
├── scripts
├── security -> ../linux-headers-5.4.0-117/security
├── sound -> ../linux-headers-5.4.0-117/sound
├── tools
├── ubuntu -> ../linux-headers-5.4.0-117/ubuntu
├── usr -> ../linux-headers-5.4.0-117/usr
└── virt -> ../linux-headers-5.4.0-117/virt

22 directories, 4 files
```

Работа над кодом

Для создания и редактирования кода модуля пригоден любой текстовый редактор, не привносящий разметки в редактируемый текст. Ряд разработчиков традиционно предпочитают редакторы vi, vim или emacs. Вполне приемлемым выбором, даже для достаточно крупных проектов («крупных» применительно именно к области модулей ядра), будет и встроенный визуальный редактор файлового менеджера Midnight Commander (**http://www.midnight-commander.org/**), показанный на рис. 3.1. Редактор активируется в MC по функциональной клавише <F4>.

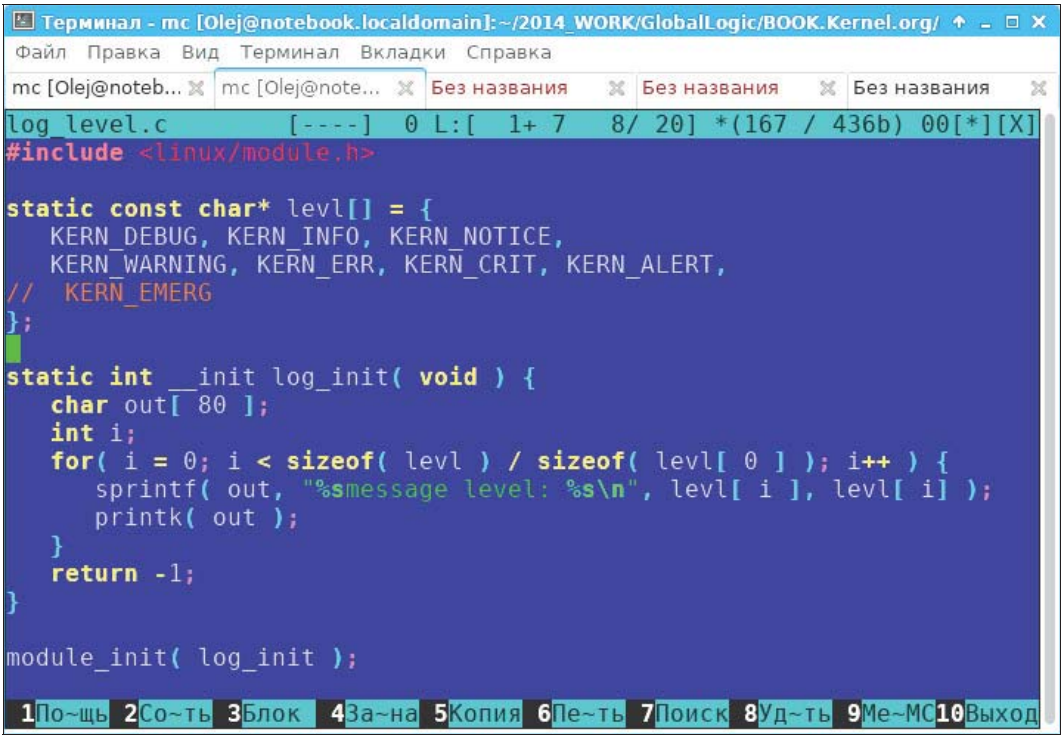


Рис. 3.1. Простейший редактор кода файлового менеджера MC с цветовой разметкой кода

Все упомянутые ранее инструменты редактирования кода (vi, vim, emacs, mc) могут использоваться для работы над кодом как в графическом *терминале*, так и в текстовой *консоли*, — при периодической смене (для контроля) одного из этих режимов на другой. В этом их большое преимущество, и именно поэтому их в любом случае нужно держать в своем арсенале.

Любители работать в более комфортных средах вполне могут использовать некоторые (но далеко не все) из множества предлагаемых в Linux интегрированных сред разработки (IDE). Но вся эта группа средств работает лишь в графическом окружении, и в некоторые моменты их использование может стать невозможным при отработке модулей ядра. Это нужно иметь в виду.

Первым несомненным кандидатом здесь будет легковесная среда Geany (**http://www.geany.org/**), представленная во всех дистрибутивах Linux. Geany фактически является не IDE, а высокопроизводительной терминальной системой, предоставляющей превосходную цветовую разметку кода, возможности перекрестного поиска по файлам, выполнения сборки (команды `make`) непосредственно, не покидая среду редактирования, и другие удобства (рис. 3.2)[4].

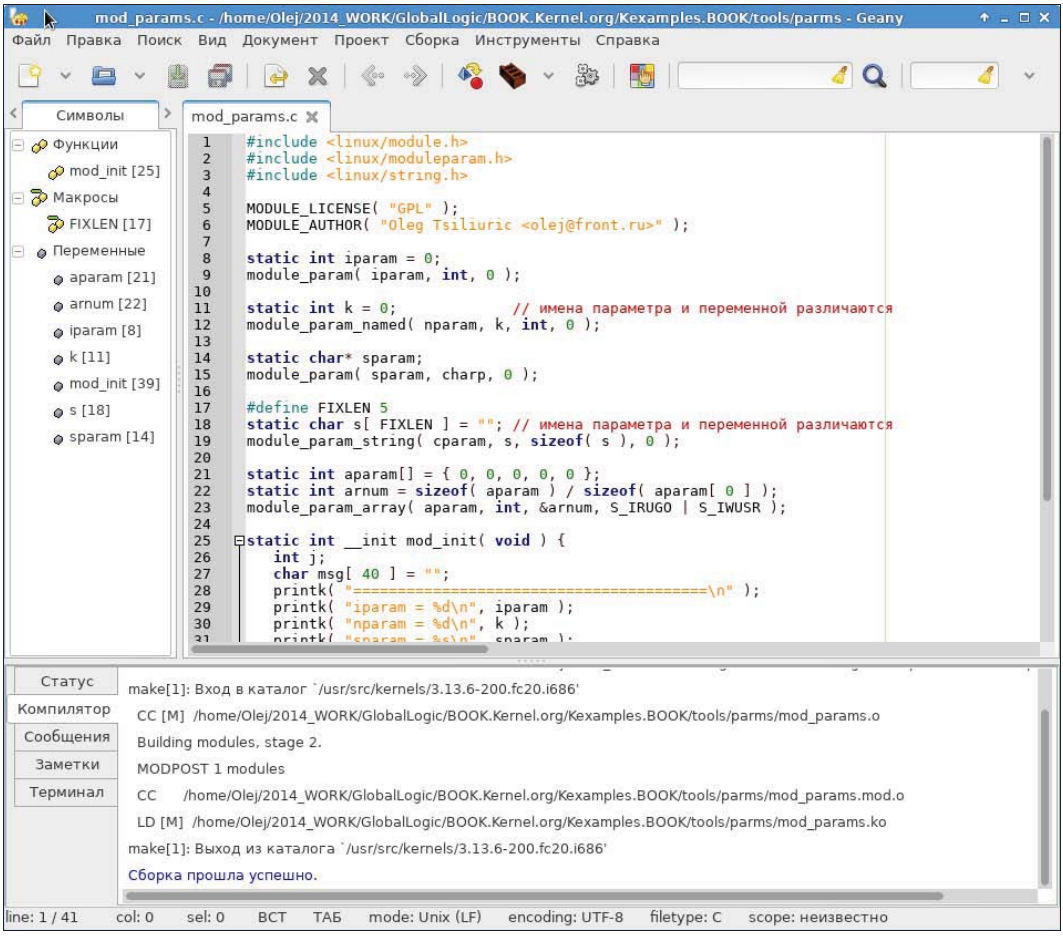


Рис. 3.2. Geany в качестве простого IDE

Здесь же у вас есть вкладка **Терминал** (слева внизу), где вы можете только что созданные модули загружать (`insmod`), выгружать (`rmmod`), анализировать результат (`dmesg`, `lsmod`, ...) — не покидая оболочки (все «в одном флаконе»: и редактирование кода, и компиляция/сборка, и отладка и наблюдение).

Еще одна удачная среда разработки (которую успешно использовал автор в одном из реальных проектов) — IDE менеджера рабочего стола KDE (KDevelop Integrated

[4] На всех снимках экранов показаны реальные примеры из архива примеров, сопровождающего книгу.

Development Environment, **http://www.kdevelop.org/**). Такое решение может оказаться выигрышным, когда модуль создается как один из компонентов в составе более сложного проекта, включающего целый набор приложений пользовательского пространства. Нужно иметь в виду, что установка этого инструмента может потянуть большой объем инсталляций, особенно если вы ведете разработку в среде, отличной от KDE (GNOME, Xfce, MATE, Cinnamon ...), — это вообще один из самых крупных пакетов для Linux (хотя при сегодняшних дисковых объемах размеры именно Linux-пакетов никого не должны волновать):

```
$ sudo apt install kdevelop
Чтение списков пакетов... Готово
Построение дерева зависимостей
...
Обновлено 0 пакетов, установлено 84 новых пакета, для удаления отмечено 2 пакета,
и 0 пакетов не обновлено.
Необходимо скачать 89,5 MB архивов.
После данной операции объем занятого дискового пространства возрастет на 515 MB.
Хотите продолжить? [Д/н] y
...
```

Вот как может выглядеть проект модуля и его сборка в KDevelop — проект получен *импортированием* существующего каталога примера модуля *вместе* с его Makefile (рис. 3.3).

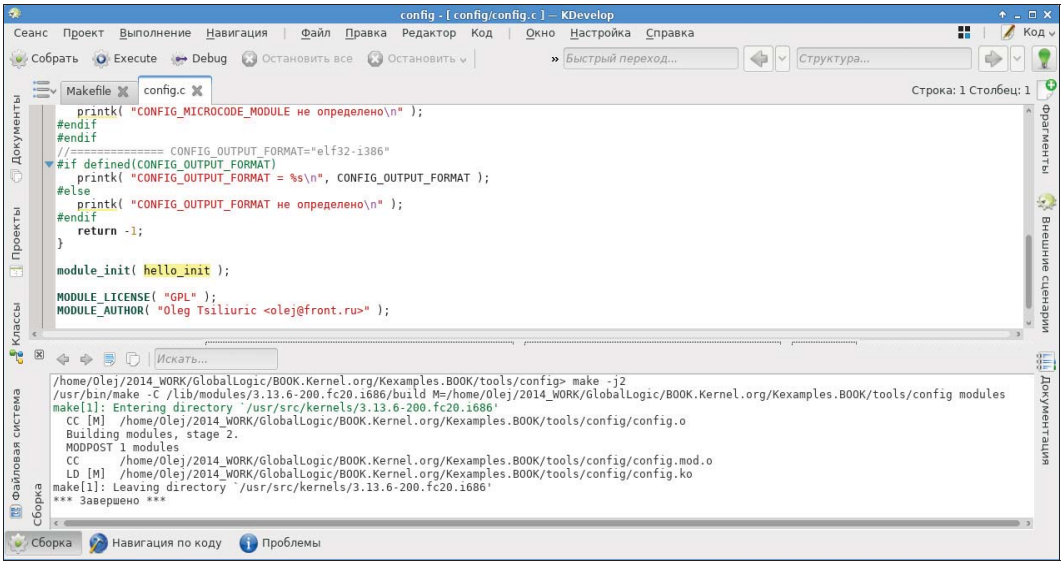


Рис. 3.3. Использование KDevelop — одной из множества доступных в Linux IDE

Сообщается о весьма успешном использовании в проектах по созданию модулей ядра для Linux известной среды разработки Eclipse (**http://www.eclipse.org/**).

Выбирайте инструмент работы с исходным кодом на свой вкус...

В деталях о сборке

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

Переменные периода компиляции

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

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

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

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

> ***Пояснение***
>
> Откуда берутся переменные, не описанные явно по тексту файла Makefile, — как, например, упоминаемая EXTRA_CFLAGS? Или откуда берутся правила сборки по умолчанию (для файлов *.c)? И как посмотреть эти правила? Все это вытекает из правил работы утилиты make — в конце книги в *приложении 1* приведена краткая справка по этим вопросам, там же имеется ссылка на детальное описание утилиты make.

Некоторые важные переменные компиляции, используемые в системном скрипте сборки модуля, могут быть переопределены непосредственно в команде сборки. Например, в одном из обсуждавшихся ранее примеров было:

```
$ uname -r
5.4.0-122-generic

$ head -n3 Makefile
CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)

$ ls /lib/modules/
5.4.0-121-generic  5.4.0-122-generic  5.4.0-97-generic  5.4.0-99-generic
```

Теперь собираем модуль под ядро версии, отличающейся от загруженной системы:

```
$ make KDIR=/lib/modules/5.4.0-121-generic/build
make -C /lib/modules/5.4.0-121-generic/build
M=/home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello modules
make[1]: вход в каталог «/usr/src/linux-headers-5.4.0-121-generic»
  CC [M]  /home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello/hello_printk.o
  Building modules, stage 2.
```

```
  MODPOST 1 modules
  CC [M]  /home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello/hello_printk.mod.o
  LD [M]  /home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello/hello_printk.ko
make[1]: выход из каталога «/usr/src/linux-headers-5.4.0-121-generic»
```

И убеждаемся, что модуль собран именно для той версии ядра, которая нас интересует (подпись `vermagic`):

```
$ modinfo hello_printk.ko
filename:       /home/olej/2022/own.BOOKs/BHV.kernel/examples/first_hello/hello_printk.ko
author:         Oleg Tsiliuric <olej.tsil@gmail.com>>
license:        GPL
srcversion:     9526D45F847C0B4EF5BBBDC
depends:
retpoline:      Y
name:           hello_printk
vermagic:       5.4.0-121-generic SMP mod_unload modversions
```

Здесь мы явно переопределили пути к корневому каталогу исходных файлов сборки и таким образом получили возможность собирать модуль для версии ядра, *отличной* от текущей, — например, для загрузки во встраиваемую конфигурацию. Совершенно естественно, что мы должны сами (*вручную* или инсталляцией пакета) создать дерево сборки для требуемой версии ядра (или просто *перенести* его копированием из интересующей нас инсталляции). Иначе команда сборки завершится терминальной ошибкой.

Дополнительные параметры периода компиляции

При сборке ядра и модулей ядра предусматриваются (разработчиками ядра) *дополнительные* параметры компиляции/сборки, которые учитываются макрокомандами сборки. Эти параметры описаны в файле source/Documentation/kbuild/kbuild.rst в дереве кодов ядра:

- `KCFLAGS` — дополнительные опции компилятора C (for built-in and modules, т. е. и для ядра, и для модулей);
- `CFLAGS_KERNEL` — additional options for $(CC) when used to compile code that is compiled as built-in, дополнительные параметры для $(CC) при компиляции кода, который скомпилирован как встроенный;
- `CFLAGS_MODULE` — additional module specific options to use for $(CC), дополнительные специальные параметры модуля, используемые для $(CC);
- `LDFLAGS_MODULE` — additional options used for $(LD) when linking modules, дополнительные опции, используемые для $(LD) при линковке модулей.

Версионность ядра в коде модуля

Предположим, вы создали в рамках своего долгосрочного проекта модуль ядра для поддержки проприетарного оборудования, производимого вашей компанией, отла-

дили его в версии ядра 4.19 и передали в производство и тиражирование продукта. Но к ядру 5.4 API ядра изменился, модуль не компилируется, а на продукт поступают потоком рекламации... Как быть? (Тем более что и потребителей, которые продолжают работать с с ядром 5.0 и не спешат обновляться до 5.4, тоже нужно обеспечивать поддержкой.)

Для этого в коде модуля (в хронологии его существования и развития) можно определить фрагменты, которые будут работать (точнее, собираться) только в определенном диапазоне версий ядра:

```
#include <linux/version.h>
...
#if LINUX_VERSION_CODE < KERNEL_VERSION(4,19,3)
...
#elif LINUX_VERSION_CODE <= KERNEL_VERSION(5,4)
...
#else
...
```

Такой способ довольно широко применялся в коде ранних версий ядра Linux, да и в последующих такие препроцессорные определения также можно нередко увидеть. Но позже, с переходом развития ядра под контроль системы контроля версий GIT, в этом отпала необходимость — каждая новая ветвь (бранч, срез) относится *только* к одной конкретной версии (подверсии) как ядра, так и строго *соответствующего* этому ядру дерева кодов модулей. В следующей ветви будет как новое ядро, так и новый набор модулей...

В связи с этим складываются и две разные линии поведения в поддержании кода модулей ядра, входящих в состав корпоративного прикладного проекта:

- отслеживать в ходе развития проекта изменения в API ядра и при возникновении несоответствий отрабатывать *разные фрагменты кода*, относящиеся к разным версиям ядра, т. к. это показано ранее;
- выполнить всю процедурную часть, требуемую командой разработчиков ядра, и *включить* свою разработку в дерево исходных кодов ядра Linux. А далее отрабатывать изменения в коде модуля синхронно с изменениями в API ядра.

Участники разработки ядра в своих публикациях, рекомендациях и интервью настаивают (и достаточно требовательно) на следовании по второму пути. Но этому могут препятствовать многие ограничения — например, такие: а) модули поддерживают узко специальное, мелкотиражное оборудование или программные модели; б) проект планируется как недостаточно долговременный; в) команда разработчиков недостаточно многочисленна и не располагает резервами трудоемкости; г) корпоративный проект не на 100% соответствует лицензии GNU.

Как собрать одновременно несколько модулей?

В уже привычного нам вида Makefile может быть описана сборка сколь угодно большого количества одновременно собираемых модулей (см. папку tools/export-data в сопровождающем книгу файловом архиве[5]):

Makefile

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

Как собрать модуль и используемые 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 (см. папку dev/ioctl в сопровождающем книгу файловом архиве) :

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 той же папке dev/ioctl):

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

[5] К этой папке мы еще раз вернемся в самом конце главы — при анализе экспортирования имен, а пока только отметим в отношении нее то, как собирается несколько модулей одновременно.

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 привносит то, что при сборке приложений и модулей (использующих совместные определения) задействуются разные дефолтные 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 всем подобии имен 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 репозитория дистрибутива) или при сборке и установке нового ядра из исходных кодов.

Пользовательские библиотеки

В дополнение к набору приложений, обсуждавшихся ранее, удобно целый ряд совместно используемых этими приложениями функций собрать в виде единой библиотеки (так устраняется дублирование кода, упрощается внесение изменений, да и вообще улучшается структура проекта). Фрагмент Makefile из папки time сопровождающего книгу файлового архива демонстрирует, как это записать, не указывая в явном виде все цели сборки (приведенные списком в переменной `OBJLIST`) для каждого такого объектного файла, включаемого в библиотеку (реализующего от-

дельную функцию библиотеки). В нашем 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`, объединенные 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 лишь 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 она будет найдена динамическим загрузчиком, размещение «текущий каталог» для этого случая неприемлемо — относительные путевые имена не применяются для по-

иска 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?

Соберем (см. папку tools/mobj в сопровождающем книгу файловом архиве) модуль из нескольких файлов исходного кода: основного файла mod.c и трех отдельно транслируемых файлов: mf1.c, mf2.c, mf3.c, — содержащих по одной отдельной функции, экспортируемой модулем (весьма общий случай). Это наше первое пересечение с понятием *экспорта имен ядра*:

mod.c

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

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

mf1.c

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

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

Заголовочный файл, включаемый в текст модулей:

mf.h

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

Ну и, наконец, в той же папке 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 нашем 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)
...
```

Примечание

Привычные из предыдущих примеров Makefile все те же определения переменных компиляции опущены.

Теперь мы можем испытать то, что мы получили:

```
$ nm mobj.ko | grep T
00000000 T cleanup_module
00000000 T init_module
00000000 T mod_func_A
00000010 T mod_func_B
00000020 T mod_func_C
$ sudo insmod ./mobj.ko
```

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

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

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

Рекурсивная сборка

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

```
$ ls -w80
$ ls -w80
blkdev  dma   first_hello  Makefile  pci     sys_call_table  time      udev
chdev   exec  IRQ          memory    procfs  sysfs           tools     usb
DKMS    file  load_module  network   signal  thread          tree.txt  user_io
```

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

Makefile

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

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

```
$ ls -l
итого 100
drwxrwxr-x 7 olej olej 4096 Янв 26 02:36 dev
drwxrwxr-x 2 olej olej 4096 Авг 27 21:57 dma
...
s ls -l
total 96
drwxrwxr-x 7 user user 4096 2011-07-02 13:11 dev
drwxrwxr-x 2 user user 4096 2011-08-27 21:57 dma
...
```

В этом случае разница обусловлена различным форматом даты и различием в числе полей. Для такой ситуации можно использовать другой вариант[6] (здесь есть пространство для изобретательства):

[6] Предложен одним из читателей рукописи книги.

Makefile

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

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

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

Подписывание модулей

С какого-то времени (несколько лет назад, раньше этого практически не встречалось) при загрузке модуля, собранного традиционным способом, появляются такие вот сообщения (будем оперировать с уже собранным нами простейшим hello_printk.ko):

```
$ dmesg | tail -n3
[ 2911.727484] EXT4-fs (sda1): mounted filesystem with ordered data mode. Opts: (null)
[ 5397.637583] hello_printk: loading out-of-tree module taints kernel.
[ 5397.637613] hello_printk: module verification failed: signature and/or required key missing
- tainting kernel
[ 5397.637804] Hello, world!
```

При этом можем видеть, что модуль загружен, как и предполагалось:

```
$ lsmod | head -n2
Module                  Size  Used by
hello_printk           16384  0
```

Природа этих сообщений связана с тем ~~безумием~~[7] движением, которое развернулось вокруг secure boot, защищенности программного обеспечения, и определяется тем, *подписан* (signed) ли или *не подписан* модуль ядра публичным ключом ядра, в которое загружается модуль. Поведение ядра и модулей в отношении подписывания определяется следующей группой конфигурационных констант периода сборки, как они определены сборщиками ядра вашего дистрибутива:

```
$ grep CONFIG_MODULE_SIG /boot/config-`uname -r`
CONFIG_MODULE_SIG_FORMAT=y
CONFIG_MODULE_SIG=y
```

[7] Это твердое убеждение автора (сугубо *субъективное* мнение!), что истинной целью всего движения «secure boot» является не столько защита от вредоносного программного обеспечения, сколько стремление крупнейших брендов IT-индустрии ограничить участие сторонних разработчиков в этой индустрии, то есть защитить собственные коммерческие интересы, свою рыночную долю.

```
# CONFIG_MODULE_SIG_FORCE is not set
CONFIG_MODULE_SIG_ALL=y
# CONFIG_MODULE_SIG_SHA1 is not set
# CONFIG_MODULE_SIG_SHA224 is not set
# CONFIG_MODULE_SIG_SHA256 is not set
# CONFIG_MODULE_SIG_SHA384 is not set
CONFIG_MODULE_SIG_SHA512=y
CONFIG_MODULE_SIG_HASH="sha512"
CONFIG_MODULE_SIG_KEY="certs/signing_key.pem"

$ grep CONFIG_SYSTEM_TRUSTED_KEYS /boot/config-`uname -r`
CONFIG_SYSTEM_TRUSTED_KEYS="debian/canonical-certs.pem"
```

Смысл этих констант (не вникая глубоко в детали криптографии, шифрования, техники асимметричных ключей и всего такого прочего) хорошо бы представлять хотя бы поверхностно (описано в документации к ядру — см. [52]):

- `CONFIG_MODULE_SIG` — средства подписи модулей включить;
- `CONFIG_MODULE_SIG_FORCE` — требование, чтобы модули были обязательно правильно подписаны. Указывает, как ядро должно работать с модулем, имеющим сигнатуру, ключ которой неизвестен, или модулем, который вообще не подписан. Если выключено (permissive), то модули, для которых ключ недоступен, и модули, которые не подписаны, разрешены, но ядро будет помечено как испорченное, и соответствующие модули будут помечены как испорченные (показано символом `y`). Если включено (restrictive), то будут загружены только модули, имеющие действительную подпись, которую можно проверить с помощью открытого ключа, находящегося во владении ядра. Все остальные модули будут генерировать ошибку. Независимо от этой настройки, если модуль имеет блок подписи, который не может быть проанализирован, он будет немедленно отклонен;
- `CONFIG_MODULE_SIG_ALL` — автоматически подписывать все модули. Если этот параметр включен, модули будут автоматически подписаны на этапе сборки `modules_install`. Если он выключен, то модули должны подписываться вручную с помощью скрипта: scripts/sign-file;
- `CONFIG_MODULE_SIG_SHA1`, `CONFIG_MODULE_SIG_SHA224`, `CONFIG_MODULE_SIG_SHA256`, `CONFIG_MODULE_SIG_SHA384`, `CONFIG_MODULE_SIG_SHA512` — выбор алгоритма хеширования, которым этап установки будет подписывать модули. Алгоритм, выбранный здесь, также будет встроен в ядро (а не в модуль), чтобы модули, подписанные с помощью этого алгоритма, могли проверять свои подписи, не вызывая циклические зависимости.

Условия в отношении подписи модулей могут отличаться самым разительным образом даже в ближайших дистрибутивах... Сравните в этом смысле Mint 20.3 и LMDE 5 — дистрибутивы практически от одной команды разработчиков (для сравнения вид подписи встроенных в дерево ядра модулей взят из вывода команды `modinfo` — первый наугад модуль):

- для Mint 20.3:

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Linuxmint
Description:    Linux Mint 20.3
Release:        20.3
Codename:       una
$ grep CONFIG_MODULE_SIG /boot/config-`uname -r`
CONFIG_MODULE_SIG_FORMAT=y
CONFIG_MODULE_SIG=y
# CONFIG_MODULE_SIG_FORCE is not set
CONFIG_MODULE_SIG_ALL=y
# CONFIG_MODULE_SIG_SHA1 is not set
# CONFIG_MODULE_SIG_SHA224 is not set
# CONFIG_MODULE_SIG_SHA256 is not set
# CONFIG_MODULE_SIG_SHA384 is not set
CONFIG_MODULE_SIG_SHA512=y
CONFIG_MODULE_SIG_HASH="sha512"
CONFIG_MODULE_SIG_KEY="certs/signing_key.pem"
$ modinfo i2c_i801 | grep ^sig
sig_id:         PKCS#7
signer:         Build time autogenerated kernel key
sig_key:        71:A4:C5:68:1B:4C:C6:E2:6D:11:37:59:EF:96:3F:B0:D9:4B:DD:F7
sig_hashalgo:   sha512
signature:      B0:A6:31:B9:BB:1B:64:30:6F:F0:95:5A:CB:C8:DA:87:75:19:1F:FB:
```

- и то же самое для LMDE 5:

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Linuxmint
Description:    LMDE 5 (elsie)
Release:        5
Codename:       elsie
$ grep CONFIG_MODULE_SIG /boot/config-`uname -r`
CONFIG_MODULE_SIG_FORMAT=y
CONFIG_MODULE_SIG=y
# CONFIG_MODULE_SIG_FORCE is not set
# CONFIG_MODULE_SIG_ALL is not set
# CONFIG_MODULE_SIG_SHA1 is not set
# CONFIG_MODULE_SIG_SHA224 is not set
CONFIG_MODULE_SIG_SHA256=y
# CONFIG_MODULE_SIG_SHA384 is not set
# CONFIG_MODULE_SIG_SHA512 is not set
CONFIG_MODULE_SIG_HASH="sha256"
CONFIG_MODULE_SIG_KEY=""
$ modinfo i2c_i801 | grep ^sig
sig_id:         PKCS#7
```

```
signer:         Debian Secure Boot CA
sig_key:        4B:6E:F5:AB:CA:66:98:25:17:8E:05:2C:84:66:7C:CB:C0:53:1F:8C
sig_hashalgo:   sha256
signature:      6B:23:98:84:6B:F1:A2:4B:D4:60:30:B9:E9:7F:45:FA:8D:8C:ED:B6:
```

Во втором варианте (`CONFIG_MODULE_SIG_ALL` не установлено) вам будут сыпаться сообщения-предупреждения, с которых мы начали разговор в этом разделе. В любом случае модули, которые вы соберете традиционным способом, будут *не подписаны*:

```
$ modinfo hello_printk.ko
filename:       /home/olej/2022/own.BOOKs/BHV.kernel/examples/tools/signed/hello_printk.ko
author:         Oleg Tsiliuric <olej.tsil@gmail.com>>
license:        GPL
srcversion:     9526D45F847C0B4EF5BBBDC
depends:
retpoline:      Y
name:           hello_printk
vermagic:       5.4.0-122-generic SMP mod_unload modversions
```

Для того чтобы не «ловить» занудных сообщений о неподписанности модуля при экспериментах, вы можете *вставить* первой строкой в Makefile, используемый для сборки модуля, строку:

```
$ head -n1 Makefile
CONFIG_MODULE_SIG=n
```

А если вы хотите создавать «правильные» модули, подписанные по всем правилам своего дистрибутива, то вы можете подписывать эти модули *вручную*, используя исполнимый скрипт:

```
$ ls -l /lib/modules/`uname -r`/build/scripts/sign-file
-rwxr-xr-x 1 root root 27152 июн 22 16:00 /lib/modules/5.4.0-122-generic/build/scripts/sign-file
```

Описание его использования (параметры и разъяснения) для ручной подписи модулей приводится в текстовом документе, на который мы ссылаемся в этом разделе (или здесь: **https://askubuntu.com/questions/483283/module-verification-failed-signature-and-or-required-key-missing**). Но это уже объемный и обстоятельный предмет, выходящий за рамки наших намерений.

Инсталляция модуля

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

1. Скопировать собранный модуль (*.ko) в его местоположение в иерархии модулей в исполняющейся файловой системе — часто это, например, каталог /lib/modules/`uname -r`/misc — в один из подкаталогов, где операционная система ищет доступные ей модули ядра.
2. Обновить информацию о взаимных зависимостях модулей (в связи с добавлением нового), что делает утилита `depmod`.

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

Нужна ли новая сборка ядра?

Это первейший вопрос всякого, кто приступает к разработке собственного модуля ядра. Из сказанного ранее (и что не раз будет подтверждаться деталями дальнейшего изложения) должно уже быть понятно: для компиляции и сборки самого элементарного модуля необходима некоторая информация, генерируемая собственно в ходе *компиляции и сборки ядра*. А именно: *абсолютные адреса* экспортируемых имен *ядра и модулей*, по которым происходит связывание имен в компилируемом модуле (те адреса, которые мы наблюдаем в /proc/kallsyms).

Для того чтобы объяснить происходящее, возьмем в исследование произвольный символ экспортируемый ядром... Символы, *экспортируемые* ядром, вместе с их абсолютными *адресами в пространстве ядра* после сборки ядра записаны *статически* в нескольких местах (файлах), например:

```
$ cat /lib/modules/`uname -r`/build/Module.symvers | grep vmlinux | grep EXPORT_SYMBOL | head -n5
0x5b03880d      ipv6_chk_custom_prefix      vmlinux  EXPORT_SYMBOL
0xf7c57701      cros_ec_check_result        vmlinux  EXPORT_SYMBOL
0x4a03bd0f      sata_pmp_error_handler      vmlinux  EXPORT_SYMBOL_GPL
0x55417264      unregister_vt_notifier      vmlinux  EXPORT_SYMBOL_GPL
0x691d503e      set_anon_super              vmlinux  EXPORT_SYMBOL
```

Сравним это с точно таким же файлом *ближайшей* предыдущей версии (после последнего обновления системы):

```
$ diff /lib/modules/`uname -r`/build/Module.symvers /lib/modules/5.4.0-113-generic/build/
Module.symvers | tail -n21
> 0x17db64a3     bpf_prog_create       vmlinux             EXPORT_SYMBOL_GPL
23833c23827
< 0x50c53593     inet_sk_diag_fillnet/ipv4/inet_diag       EXPORT_SYMBOL_GPL
---
> 0xeadb21c8     inet_sk_diag_fillnet/ipv4/inet_diag       EXPORT_SYMBOL_GPL
23835c23829
< 0xff4794a9     rdma_link_unregister      drivers/infiniband/core/ib_core   EXPORT_SYMBOL
---
> 0x5ec3f1d2     rdma_link_unregister      drivers/infiniband/core/ib_core   EXPORT_SYMBOL
23842,23844c23836,23838
< 0x305c508d        tcp_ca_openreq_child          vmlinux      EXPORT_SYMBOL_GPL
< 0x65dfba2b        clean_acked_data_enable       vmlinux      EXPORT_SYMBOL_GPL
< 0x40cfb8b5        bpf_prog_destroy              vmlinux      EXPORT_SYMBOL_GPL
---
> 0x6429a071        tcp_ca_openreq_child          vmlinux      EXPORT_SYMBOL_GPL
> 0x489b9ee1        clean_acked_data_enable       vmlinux      EXPORT_SYMBOL_GPL
```

```
> 0x03896d27        bpf_prog_destroy              vmlinux       EXPORT_SYMBOL_GPL
23850c23844
< 0xa1089f2f        pci_enable_atomic_ops_to_root vmlinux       EXPORT_SYMBOL
---
> 0xb82fbec4        pci_enable_atomic_ops_to_root vmlinux       EXPORT_SYMBOL
```

Таких *отличающихся* адресов тех же имен:

```
$ diff /lib/modules/`uname -r`/build/Module.symvers /lib/modules/5.4.0-113-generic/build/
Module.symvers | grep "\-\-\-" | wc -l
3315
```

Это *совершенно другие адреса* для более чем 3000 имен (точек входа и др.), и не важно, отличаются ли они от исходного на 2 байта или переместятся в другую страницу физической RAM... Ясно только, что если мы по данным одной системы соберем модуль, использующий вызов в другой, то выполнение любого такого вызова просто вдребезги разнесет ядро Linux. Более того, та же история произойдет, если мы заново соберем *то же самое* ядро, но изменим при его конфигурировании один из великого множества параметров конфигурации ядра `CONFIG_*` (мы говорили о них раньше), — все адреса точек входов «поплывут». В итоге мы приходим к выводу, что для сборки любого своего простейшего модуля ядра необходимы *данные сборки* самого ядра. Это плата за монолитность ядра Linux!

Компиляция текущего ядра из исходных кодов — трудоемкий процесс, который может потребовать от 20–30 минут непрерывного процессорного времени (на самых быстрых процессорах) до нескольких часов (на более старых компьютерах). Поэтому дистрибьюторы основных пакетных дистрибутивов Linux включают в репозитарии *пакеты*, достаточные для работы с модулями ядра. При этом для сборки и отработки модулей ядра перекомпиляция самого ядра (и загружаемого образа системы) в обязательном порядке *не нужна*. Один из таких пакетов (RPM-дистрибутивы) — `kernel-devel-*` (который вы должны установить для работы с модулями):

```
$ rpm -ql kernel-devel-* | grep System
/usr/src/kernels/5.17.11-200.fc35.x86_64/System.map
/usr/src/kernels/5.17.12-200.fc35.x86_64/System.map
/usr/src/kernels/5.17.13-200.fc35.x86_64/System.map
```

Этот пакет устанавливает всю иерархию дерева сборки ядра и все необходимые результирующие данные сборки (System.map и др.) такого ядра.

Кроме того, как уже было сказано, для работы с модулями необходимо наличие *заголовочных файлов ядра* (в точности соответствующих загруженной версии ядра!), сам же исходный программный код системы *не нужен*. Эти заголовочные файлы устанавливаются другим пакетом из той же группы: `kernel-headers-*`.

Тем не менее новая сборка ядра Linux, с чего и началось обсуждение этой главы, может оказаться полезной и необходимой в некоторых случаях: для сборки ядра с некоторыми специальными качествами — например, с повышенными отладочными уровнями. Для сложных комплексных и долгосрочных проектов сборка рабочей версии ядра может оказаться желательной. Все новые специальные качества ядра, о которых упоминалось ранее, определяются исключительно *конфигурацион-*

ными параметрами ядра, задаваемыми пользователем в ходе диалога конфигурирования ядра, предшествующего сборке. О том, как использовать значения конфигурационных параметров ядра в коде модулей, рассказывалось ранее.

Динамическая сборка модулей (DKMS)

Из приведенных рассуждений должно уже быть понятно: подгружаемый модуль ядра связывается с экспортируемыми именами ядра (функциями и данными) по абсолютным адресам в памяти этих имен. Как следствие, в Linux *не может быть в принципе* бинарных драйверов, «готовых к употреблению» в самых разных ядрах системы. Модуль ядра должен предоставляться исключительно в форме исходного кода на языке C, который подлежит в дальнейшем компиляции в окружении той системы, где он будет использоваться.

Как следствие, при самом стандартном обновлении ядра из репозитория дистрибутива установленный вами модуль ядра становится не пригодным к использованию, и должен быть перекомпилирован из своего исходного кода. Модули, которые включены в дерево исходных кодов, будут обновляться пакетной системой вместе с обновлением ядра, но вот к проприетарным вручную установленным драйверам это не относится.

Для пользователей, которые вовсе не «на ты» с системой и техникой компиляции, разобраться с такими ситуациями сложно и трудоемко. Для разрешения этого казуса и предложена система автоматической перекомпиляции модулей ядра DKMS (Dynamic Kernel Module Support) — фреймворк, который используется для динамической генерации тех модулей ядра Linux, которые в общем случае не включены в дерево исходного кода.

Этот инструмент по умолчанию может быть и не установлен, что потребует установки его из пакетного репозитория стандартным способом:

```
$ apt show dkms
Package: dkms
Version: 2.8.1-5ubuntu2
Priority: optional
Section: admin
Origin: Ubuntu
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer: Dynamic Kernel Modules Support Team <dkms@packages.debian.org>
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Installed-Size: 296 kB
Pre-Depends: lsb-release
Depends: kmod | kldutils, gcc | c-compiler, dpkg-dev, make | build-essential, coreutils
(>= 7.4), patch, dctrl-tools
Recommends: fakeroot, sudo, linux-headers-686-pae | linux-headers-amd64 | linux-headers-
generic | linux-headers
Suggests: menu, e2fsprogs
Breaks: shim-signed (<< 1.34~)
Homepage: https://github.com/dell-oss/dkms
Download-Size: 66,8 kB
```

```
APT-Manual-Installed: yes
APT-Sources: http://ubuntu.volia.net/ubuntu-archive focal-updates/main amd64 Packages
Description: инфраструктура для поддержки динамически загружаемых модулей ядра
 DKMS — инфраструктура, позволяющая обновлять модули ядра без изменения
 всего ядра. Также позволяет легко пересобирать модули при обновлении ядра.
```

Смотрим возможности этого инструмента:

```
$ dkms --help
Error!  Unknown option: --help
Usage: /usr/sbin/dkms [action] [options]
  [action]  = { add | remove | build | install | uninstall | match | autoinstall |
                                                                        mkdriverdisk |
               mktarball | ldtarball | mkrpm | mkkmp | mkdeb | mkdsc | mkbmdeb | status }
  [options] = [-m module] [-v module-version] [-k kernel-version] [-a arch]
              [-d distro] [-c dkms.conf-location] [-q] [--force] [--force-version-override]
                                                                              [--all]
              [--templatekernel=kernel] [--directive='cli-directive=cli-value']
              [--config=kernel-.config-location] [--archive=tarball-location]
              [--kernelsourcedir=source-location] [--no-prepare-kernel] [--no-initrd]
              [--binaries-only] [--source-only] [-r release (SuSE)] [--verbose]
              [--size] [--spec=specfile] [--media=floppy|iso|tar] [--legacy-postinst=0|1]
              [--no-depmod] [-j number] [--version]
```

```
$ ls -l /var/lib/dkms/
итого 8
-rw-r--r-- 1 root root    6 окт 31  2019 dkms_dbversion
drwxr-xr-x 3 root root 4096 июл 11 23:03 virtualbox
```

Пока у меня услугами DKMS пользуется, что совершенно естественно, только VirtualBox для сборки своих модулей. А теперь мы отправим под динамический контроль свой пробный модуль (см. папку first_hello в сопровождающем книгу файловом архиве), который уже собирали вручную в качестве упражнения начального уровня... Создаем (добавляем) под него *каталог* с указанием *версии* проекта (модуля) вот в этом месте:

```
$ ls /usr/src/
linux-headers-5.4.0-121  linux-headers-5.4.0-121-generic  linux-headers-5.4.0-122  linux-
headers-5.4.0-122-generic  virtualbox-6.1.34
$ sudo mkdir /usr/src/hello_printk-3.1/
$ ls -l /usr/src/
итого 24
drwxr-xr-x  2 root root 4096 авг  8 09:17 hello_printk-3.1
drwxr-xr-x 24 root root 4096 июн 23 00:52 linux-headers-5.4.0-121
drwxr-xr-x  7 root root 4096 июн 23 00:52 linux-headers-5.4.0-121-generic
drwxr-xr-x 24 root root 4096 июл 11 23:03 linux-headers-5.4.0-122
drwxr-xr-x  7 root root 4096 июл 11 23:03 linux-headers-5.4.0-122-generic
drwxr-xr-x 11 root root 4096 июл  4 18:26 virtualbox-6.1.34
```

Скопируем туда *все* файлы, необходимые для ручной сборки модуля. В нашем простейшем случае это файлы hello_printk.c и Makefile, уже предварительно *выверенные*

ручной сборкой. И добавим туда же файл конфигурации DKMS dkms.conf, наполнив его соответствующим содержимым, — например, так:

```
$ touch dkms.conf
$ cat dkms.conf
PACKAGE_NAME="hello_printk"
PACKAGE_VERSION="3.1"
BUILT_MODULE_NAME[0]="hello_printk"
DEST_MODULE_LOCATION[0]="/kernel/lib/hello_printk/"
AUTOINSTALL="yes"
```

А исходный код модуля дополним двумя строками (макросами) описаний, которые до этого нам казались там не обязательными (но вполне могли уже и присутствовать), — подсистема DKMS строится на базе *версий модулей*:

```
$ grep MODULE_ hello_printk.c
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>>");
MODULE_DESCRIPTION("DKMS module hello_printk");
MODULE_VERSION("3.1");
```

Теперь у нас уже все готово для того, чтобы *подключить* свой написанный модуль под контроль динамической сборки DKMS:

```
$ sudo dkms add -m hello_printk -v 3.1

Creating symlink /var/lib/dkms/hello_printk/3.1/source ->
                 /usr/src/hello_printk-3.1

DKMS: add completed.

$ sudo dkms build -m hello_printk -v 3.1

Kernel preparation unnecessary for this kernel.  Skipping...

Building module:
cleaning build area...
make -j40 KERNELRELEASE=5.4.0-122-generic -C /lib/modules/5.4.0-122-generic/build
M=/var/lib/dkms/hello_printk/3.1/build...
cleaning build area...

DKMS: build completed.

$ tree /var/lib/dkms/hello_printk/3.1
/var/lib/dkms/hello_printk/3.1
├── 5.4.0-122-generic
│   └── x86_64
│       ├── log
│       │   └── make.log
│       └── module
│           └── hello_printk.ko
└── source -> /usr/src/hello_printk-3.1

5 directories, 2 files
```

```
$ sudo dkms install -m hello_printk -v 3.1

hello_printk.ko:
Running module version sanity check.
 - Original module
   - No original module exists within this kernel
 - Installation
   - Installing to /lib/modules/5.4.0-122-generic/kernel/lib/hello_printk//

depmod....

DKMS: install completed.

$ ls -l /lib/modules/`uname -r`/kernel/lib/hello_printk
итого 8
-rw-r--r-- 1 root root 4544 авг  8 09:25 hello_printk.ko
```

Я специально определил (см. файл dkms.conf) местоположение инсталляции своего модуля в достаточно произвольное место: /lib/modules/`uname -r`/kernel/lib/hello_printk. Но сам проект DKMS рекомендует размещать такие модули в /lib/modules/`uname -r`/updates или в /lib/modules/`uname -r`/extra:

```
$ tree /lib/modules/`uname -r`/updates
/lib/modules/5.4.0-122-generic/updates
├── vboxdrv.ko
├── vboxnetadp.ko
└── vboxnetflt.ko

0 directories, 3 files
```

Теперь наш написанный модуль hello_printk.ko находится под контролем подсистемы динамической сборки. И мы из любого места файловой системы, без привязки к местоположению, можем выполнить диагностику и загрузку инсталлированного модуля без указания расширения файла модуля ko:

```
$ modinfo hello_printk
filename:       /lib/modules/5.4.0-122-generic/kernel/lib/hello_printk/hello_printk.ko
version:        3.1
description:    DKMS module hello_printk
author:         Oleg Tsiliuric <olej.tsil@gmail.com>>
license:        GPL
srcversion:     6D5ACE57C7E17A1DF4E30B5
depends:
retpoline:      Y
name:           hello_printk
vermagic:       5.4.0-122-generic SMP mod_unload modversions
$ lsmod | head -n3
Module                  Size  Used by
vboxnetadp             28672  0
vboxnetflt             28672  0
```

```
$ sudo modprobe hello_printk
$ echo $?
0
$ lsmod | head -n3
Module                  Size  Used by
hello_printk           16384  0
vboxnetadp             28672  0
$ dmesg | tail -n1
[ 2439.063870] Hello, world!
```

И наконец, проверить, что у нас находится под контролем DKMS:

```
$ dkms status
hello_printk, 3.1, 5.4.0-122-generic, x86_64: installed
virtualbox, 6.1.34, 5.4.0-121-generic, x86_64: installed
virtualbox, 6.1.34, 5.4.0-122-generic, x86_64: installed
```

В завершение нужно убрать за собой — деинсталлировать и удалить из-под контроля DKMS модуль hello_printk.ko ... потому что, пока мы с ним только играли:

```
$ sudo dkms uninstall -m hello_printk -v 3.1

-------- Uninstall Beginning --------
Module:  hello_printk
Version: 3.1
Kernel:  5.4.0-122-generic (x86_64)
-------------------------------------

Status: Before uninstall, this module version was ACTIVE on this kernel.

hello_printk.ko:
 - Uninstallation
   - Deleting from: /lib/modules/5.4.0-122-generic/kernel/lib/hello_printk//
 - Original module
   - No original module was found for this module on this kernel.
   - Use the dkms install command to reinstall any previous module version.

depmod...

DKMS: uninstall completed.
$ ls -l /lib/modules/`uname -r`/kernel/lib/hello_printk
ls: невозможно получить доступ к '/lib/modules/5.4.0-122-generic/kernel/lib/hello_printk':
Нет такого файла или каталога

$ sudo dkms remove hello_printk/3.1 --all

-------- Uninstall Beginning --------
Module:  hello_printk
Version: 3.1
Kernel:  5.4.0-122-generic (x86_64)
-------------------------------------
```

```
Status: This module version was INACTIVE for this kernel.
depmod...

DKMS: uninstall completed.

------------------------------
Deleting module version: 3.1
completely from the DKMS tree.
------------------------------
Done.
```

Затем можно выполнить проверку того, что мы вернулись к состоянию, которое предшествовало нашим экспериментам с DKMS:

```
$ dkms status
virtualbox, 6.1.34, 5.4.0-121-generic, x86_64: installed
virtualbox, 6.1.34, 5.4.0-122-generic, x86_64: installed
```

В заключение нужно заметить, что работа DKMS ясно *подтверждает*, что в Linux драйверы (модули) могут предоставляться *только* в виде исходных кодов на языке программирования C, требуемых для компиляции (сборки), и при наличии необходимого для такой сборки инструментария. DKMS лишь скрывает это под покровом обновления пакетной системы. И если в коде вашего модуля использованы какие-то замысловатые API ядра, которые могут стать недействительными после обновления ядра, то последовательность динамических обновлений модуля на этом месте обломается.

Обсуждение

- В этой главе мы очень кратко рассмотрели минимальный набор инструментальных средств, требуемых для создания и отладки модулей ядра. Это именно тот «необходимый и достаточный» комплект инструментов, который позволяет вам выполнять такую работу. Есть еще весьма много средств, расширяющих ваши возможности в этой области или существенно повышающих производительность труда. Но это все *дополнительные* инструменты, и вы соберете оптимальный для себя набор инструментов, экспериментируя с ними.
- Представляющие интерес системные файлы могут несколько различаться в разных дистрибутивных группах Linux (Debian, RPM и др.), но никаких принципиальных различий каждый используемый дистрибутив Linux не привносит.

- Глава 4 -
Внешние интерфейсы модуля

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

Скотт Максвелл, «Ядро Linux в комментариях»

Под внешними интерфейсами модуля мы будем понимать, как уже было сказано ранее, те связи, которые может и должен установить модуль с «внешним пространством» Linux, видимым пользователю, и с которыми пользователь может взаимодействовать либо из своего программного кода, либо посредством консольных команд системы. Такими интерфейсами-связями являются, например, имена в файловых системах (в /dev, /proc, /sys), сетевые интерфейсы, сетевые протоколы... Понятно, что регистрация этих механизмов взаимодействия со стороны модуля не есть программирование в смысле алгоритмов и структур данных, а лишь строго формализованное (регламентированное как по номенклатуре, так и по порядку вызова) использование предоставляемых для этих целей API ядра. Это занятие скучное, но оно представляет собой первейшую фазу проектирования всякого модуля (драйвера) — создание тех связей, через которые с ним можно взаимодействовать. Чем мы и будем заниматься на протяжении всей этой главы.

Драйверы: интерфейс устройства

Модель «устройств» Linux унаследовала из самых ранних реализаций UNIX, где его авторы попытались представить «всё как файл». Каждое устройство представляется его уникальным именем в каталоге /dev, это же имя и называется устройством.

> ***Примечание***
>
> Как будет показано вскоре, имя устройства может быть создано не только в /dev, но и в любом произвольном месте файловой системы Linux, — важно только, чтобы это имя было связано с уникальными номерами `major` и `minor`. Но такое использование, вне каталога /dev, нарушает традиции UNIX, и если к нему и прибегать, то с большой осторожностью.

Все устройства Linux делятся на две большие группы: *символьные устройства* (они же устройства с последовательным доступом) и *блочные устройства* (устройства с произвольным доступом). Устройства с последовательным доступом читают

(пишут) данные *последовательно* байт за байтом, причем ранее уже считанный байт более не может быть повторно считан (за редким исключением ограниченной внутренней буферизации). В противоположность этому устройство с произвольным доступом может читать *любые* байты — от нулевого до максимальной *емкости* устройства, в произвольном порядке и сколько угодно раз, просто указывая номер интересующего байта. Характерными представителями символьных устройств являются: терминальные линии /dev/tty* и псевдоустройства /dev/zero, /dev/null, /dev/random, а также все физические линии связи: RS-232, Modbus, CAN... Как видите, символьные устройства многолики и разнообразны. Устройства с произвольным доступом выступают в роли устройств хранения (дисковых, твердотельных, флеш-накопителей USB, ...), почему их и называют еще *блочными* устройствами. Логика их использования более однообразная — *над* блочными устройствами (как над устройствами хранения) *надстраиваются* (размечаются, форматируются) файловые системы, которые и являются конечной целью организации таких устройств. Здесь мы храним наши каталоги и файлы.

Смысл операций с интерфейсом /dev состоит в связывании *именованного устройства* в каталоге /dev с разрабатываемым модулем, а в самом коде модуля — в реализации нескольких типовых операций на этом устройстве (таких как `open()`, `read()`, `write()` и нескольких других). После этого пользовательский код может выполнять с таким устройством весь спектр POSIX API, а ваш модуль будет обслуживать эти операции в ядре. В таком качестве модуль ядра и называется *драйвером устройства*. Некоторую путаницу в проектировании драйвера создает то, что для этого действа предлагается несколько альтернативных, совершенно исключающих друг друга техник написания. Связано это с давней *историей* развития подсистемы /dev (одна из самых старых подсистем UNIX и Linux) и с тем, что на протяжении этой истории отрабатывалось несколько *различающихся моделей реализации*, а удачные решения на этом пути закреплялись как *альтернативы*. Все исторически складывающиеся альтернативы сохранялись далее для обеспечения совместимости.

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 (под парой номеров `major` и `minor`), каким образом станет известно системе, что у нее появился в распоряжении новый драйвер и новое устройство;
- каким образом драйвер создает (или использует созданное внешними средствами) имя соответствующего ему устройства в каталоге /dev и как он (драйвер) *связывает* это имя с номерами `major` и `minor` этого устройства;
- после того как драйвер однозначно увязан с устройством, какие будут использованы особенности в реализации основных *операций обслуживания* устройства (`open()`, `read()`, ...)?

Но прежде, чем перейти к созданию интерфейса устройства, очень коротко вспомним философию устройств, общую не только для Linux, но и для всех UNIX/POSIX-систем. Каждому устройству в системе соответствует имя этого устройства в каталоге /dev. Каждое именованное устройство в Linux однозначно характеризу-

ется двумя номерами: старшим номером (major), отвечающим за отдельный *класс* устройств, и младшим номером (minor) — конкретного *экземпляра* 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 — младший номер для второго (sda2) раздела первого (sda) 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 со многими типами оборудования жестко регламентирована (особенно в отношении старших номеров). В предыдущих версиях ядра она определялась содержимым файла Documentation/devices.txt в исходных кодах ядра (больше 100 Кбайт текста приведено для примера в каталоге /dev/).

Номера major для *символьных* и для *блочных* устройств составляют совершенно различные пространства номеров и могут использоваться независимо, пример чему — набор разных устройств с одинаковым номером major:

```
$ ls -l /dev | grep ' 7,'
brw-rw----   1 root disk       7,   0 июн 14 10:45 loop0
brw-rw----   1 root disk       7,   1 июн 14 10:45 loop1
brw-rw----   1 root disk       7,   2 июн 14 10:45 loop2
brw-rw----   1 root disk       7,   3 июн 14 10:45 loop3
brw-rw----   1 root disk       7,   4 июн 14 10:45 loop4
brw-rw----   1 root disk       7,   5 июн 14 10:45 loop5
brw-rw----   1 root disk       7,   6 июн 14 10:45 loop6
brw-rw----   1 root disk       7,   7 июн 14 10:45 loop7
crw-rw----   1 root tty        7,   0 июн 14 10:45 vcs
crw-rw----   1 root tty        7,   1 июн 14 10:45 vcs1
crw-rw----   1 root tty        7,   2 июн 14 10:45 vcs2
crw-rw----   1 root tty        7,   3 июн 14 10:45 vcs3
crw-rw----   1 root tty        7,   4 июн 14 10:45 vcs4
crw-rw----   1 root tty        7,   5 июн 14 10:45 vcs5
crw-rw----   1 root tty        7,   6 июн 14 10:45 vcs6
crw-rw----   1 root tty        7,   7 июн 14 10:45 vcs7
crw-rw----   1 root tty        7, 128 июн 14 10:45 vcsa
crw-rw----   1 root tty        7, 129 июн 14 10:45 vcsa1
crw-rw----   1 root tty        7, 130 июн 14 10:45 vcsa2
crw-rw----   1 root tty        7, 131 июн 14 10:45 vcsa3
crw-rw----   1 root tty        7, 132 июн 14 10:45 vcsa4
crw-rw----   1 root tty        7, 133 июн 14 10:45 vcsa5
```

```
crw-rw----   1 root tty         7, 134 июн 14 10:45 vcsa6
crw-rw----   1 root tty         7, 135 июн 14 10:45 vcsa7
crw-rw----   1 root tty         7,  64 июн 14 10:45 vcsu
crw-rw----   1 root tty         7,  65 июн 14 10:45 vcsu1
crw-rw----   1 root tty         7,  66 июн 14 10:45 vcsu2
crw-rw----   1 root tty         7,  67 июн 14 10:45 vcsu3
crw-rw----   1 root tty         7,  68 июн 14 10:45 vcsu4
crw-rw----   1 root tty         7,  69 июн 14 10:45 vcsu5
crw-rw----   1 root tty         7,  70 июн 14 10:45 vcsu6
crw-rw----   1 root tty         7,  71 июн 14 10:45 vcsu7
```

Примечание

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

Многие годы в UNIX и Linux предполагалось что `major` и `minor` — это байтовые величины. Но на сегодня (начиная с 2.6.0) номера устройства определяются в <linux/types.h> как 32-битные значения `dev_t`:

```
typedef u32 __kernel_dev_t;
typedef __kernel_fd_set         fd_set;
typedef __kernel_dev_t          dev_t;
```

где 12 битов занимает старший номер (0...4096-1), а 20 — младший номер (0...1048576-1). Конечно, ваш исходный код не обязан делать предположений о внутренней организации номеров устройств, а должен использовать макросы из файла <linux/kdev_t.h>, где для старшей и младшей частей `dev_t` используются соответственно:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

И, наоборот, `dev_t` получается из двух частей макросом:

```
MKDEV(int major, int minor);
```

Символьные устройства

Как правило, на практике наиболее часто встречается задача создания именно символьных устройств, которые позволяют читать и писать последовательный *поток* байтов (символов). При этом зачастую в прикладных проектах в качестве символьных устройств оформляются разнообразные *псевдоустройства*, не имеющие отношения к аппаратному оборудованию, моделирующие программную обработку, но предоставляющие ее результаты стандартными POSIX C операциями ввода/вывода. Классическими примерами таких псевдоустройств являются, например, широко использующиеся в Linux устройства:

```
$ ls -l /dev/{null,zero}
crw-rw-rw- 1 root root 1, 3 июн 15 09:59 /dev/null
crw-rw-rw- 1 root root 1, 5 июн 15 09:59 /dev/zero
$ ls -l /dev/random
crw-rw-rw- 1 root root 1, 8 июн 15 09:59 /dev/random
```

Практически вся полезная работа любого модуля символьного устройства в интерфейсе /dev (точно так же, как и в интерфейсах /proc и /sys, рассматриваемых позже) реализуется через таблицу (структуру) файловых операций `file_operations`, которая определена в файле <linux/fs.h> и содержит *указатели на функции* драйвера, которые отвечают за выполнение различных операций с устройством. Эта большая структура настолько важна, что она стоит того, чтобы быть приведенной полностью (в том виде, как это имеет место в ядре 5.4):

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iopoll)(struct kiocb *kiocb, bool spin);
        int (*iterate) (struct file *, struct dir_context *);
        int (*iterate_shared) (struct file *, struct dir_context *);
        __poll_t (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        unsigned long mmap_supported_flags;
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
                                           unsigned long, unsigned long);
```

```
        int (*check_flags)(int);
        int (*setfl)(struct file *, unsigned long);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
                                                                        unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t,
                                                                        unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
#endif
        ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
                        loff_t, size_t, unsigned int);
        loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
                                   struct file *file_out, loff_t pos_out,
                                   loff_t len, unsigned int remap_flags);
        int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

Как понятно из вида приведенной таблицы, каждое определение задает *прототип* функции, которая выполняется при системном вызове для этой операции. Если мы переопределяем в своем коде модуля какую-то из функций таблицы, то эта функция становится обработчиком, вызываемым для обслуживания этой операции. Если мы не переопределяем операцию, то для большинства операций (`llseek`, `flush` и др.) используется *обработчик по умолчанию*. Такой обработчик может вообще не выполнять никаких действий или выполнять некоторый минимальный набор простейших действий. Эта ситуация встречается достаточно часто — например, при отсутствии определений для операций `open` и `release` на устройстве, хотя тем не менее устройства замечательно открываются и закрываются. Но для некоторых операций (`mmap` и др.) обработчик по умолчанию будет всегда возвращать код ошибки (not implemented yet), и поэтому такой обработчик имеет смысл только тогда, когда он переопределен.

Вот еще одна структура (`inode_operations`), которая менее значима, чем `file_operations`, но также широко используется:

```
struct inode_operations {
        struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
        const char * (*get_link) (struct dentry *, struct inode *, struct delayed_call *);
        int (*permission) (struct inode *, int);
        struct posix_acl * (*get_acl)(struct inode *, int);

        int (*readlink) (struct dentry *, char __user *,int);

        int (*create) (struct inode *,struct dentry *, umode_t, bool);
        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 *,umode_t);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
        int (*rename) (struct inode *, struct dentry *,
                       struct inode *, struct dentry *, unsigned int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (const struct path *, struct kstat *, u32, unsigned int);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start, u64 len);
        int (*update_time)(struct inode *, struct timespec64 *, int);
        int (*atomic_open)(struct inode *, struct dentry *,
                           struct file *, unsigned open_flag,
                           umode_t create_mode);
        int (*tmpfile) (struct inode *, struct dentry *, umode_t);
        int (*set_acl)(struct inode *, struct posix_acl *, int);
} ____cacheline_aligned;
```

Примечание

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

Варианты реализации

Возвращаемся к регистрации драйвера в системе. Некоторую путаницу в этом вопросе создает 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 конкретно для этого способа. Именно на этапе связывания устройства и возникает отмечаемое многими изобилие операторов `goto` в коде драйвера, когда при неудаче очередного шага установки приходится последовательно отменять результаты всех проделанных шагов в обратном порядке. Для регистрации устройства и создания связи (интерфейса) модуля с /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:

- 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;

- способ полностью динамического создания именованных устройств — так называемая техника misc drivers (от miscellaneous, разнородный);
- старый способ (использующий `register_chrdev()`), статически связывающий модуль со старшим номером, тем самым отдавая под контроль модуля *весь* диапазон допустимых младших номеров [0...255]. Название способа *старым* не отменяет его актуальности и на сегодня.

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

Ручное создание имени

Наш первый вариант модуля символьного устройства предоставляет пользователю только операцию чтения из устройства (операция записи реализуется абсолютно симметрично и не приводится здесь, чтобы не перегружать текст. Аналогичная реализация будет показана и 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 объема изложения):

dev.h

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");
MODULE_VERSION("7.4");

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

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

```
   printk(KERN_INFO "=== read return : %d\n", len);
   return len;
}

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

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

Тогда наш первый вариант драйвера (см. папку chdev/cdev в сопровождающем книгу файловом архиве), использующий структуру `struct cdev`, примет следующий вид (рассмотренный общий файл dev.h включен в качестве преамбулы этого кода — так будет и в дальнейших примерах):

fixdev.c

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

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

#define EOK 0
static int device_open = 0;

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

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

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

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

```
static struct cdev hcdev;

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

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

Здесь на шаге инсталляции модуля показан только один (для краткости) уход на метку ошибки выполнения (`err:`), но в коде реальных модулей вы увидите целые цепочки подобных конструкций для отработки возможных ошибок на каждом шаге инсталляции: ошибка на каждом шаге должна вызывать переход на свою метку обработки ошибки с последовательным откатом в обратном порядке всех уже произведенных ранее шагов.

Наш 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 пара-

метра 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.

Вызовы API ядра `register_chrdev_region()` и `alloc_chrdev_region()` различаются тем, что первый из них запрашивает `DEVICE_COUNT` младших номеров, начиная с указанного `dev_t`, а второй выделяет эти же `DEVICE_COUNT` младших номеров, но в удобном для системы диапазоне старшего и начального младшего номера. Как должно быть понятно, вероятность получить ошибку в первом варианте намного выше.

Далее, путем экспериментирования мы проверяем работоспособность написанного модуля, и эти эксперименты очень много проясняют по части драйверов устройств Linux. Первым делом мы попытаемся привязать драйвер (и проверим это) к номеру `major`, который уже занят в системе:

```
$ ls -l /dev | grep ^c | head -n15
crw-r--r--   1 root root       10, 235 июн 14 10:45 autofs
crw-rw----   1 root disk       10, 234 июн 14 10:45 btrfs-control
crw--w----   1 root tty         5,   1 июн 14 10:45 console
crw-------   1 root root       10,  59 июн 14 10:45 cpu_dma_latency
crw-------   1 root root       10, 203 июн 14 10:45 cuse
crw-------   1 root root       10,  62 июн 14 10:45 ecryptfs
crw-rw----   1 root video      29,   0 июн 14 10:45 fb0
crw-rw-rw-   1 root root        1,   7 июн 14 10:45 full
crw-rw-rw-   1 root root       10, 229 июн 14 10:45 fuse
crw-------   1 root root      240,   0 июн 14 10:45 hidraw0
crw-------   1 root root      240,   1 июн 14 10:45 hidraw1
crw-------   1 root root      240,   2 июн 14 10:45 hidraw2
crw-------   1 root root      240,   3 июн 14 10:45 hidraw3
crw-------   1 root root      240,   4 июн 14 10:45 hidraw4
crw-------   1 root root      240,   5 июн 14 10:45 hidraw5
$ sudo insmod fixdev.ko major=240
insmod: ERROR: could not insert module fixdev.ko: Device or resource busy
$ dmesg | tail -n1
[31044.639034] === Can not register char device region
```

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

В конечном итоге самый простой способ — поручить самому драйверу найти первый свободный `major` в системе (в вашей системе он может быть совершенно другой):

```
$ sudo insmod fixdev.ko
$ dmesg | tail -n1
[31255.113928] ========== module installed 235:0 ==============
$ lsmod | grep fix
fixdev                 16384  0
$ cat /proc/devices | grep my_
235 my_cdev_dev
```

Теперь драйвер установлен в системе по всем правилам. Он связан для обслуживания устройства с парой номеров (235:0 и далее)... но он не привязан к имени устройства в /dev в общепринятом смысле — мы не знаем этого имени, чтобы выполнять операции ввода/вывода. Но мы можем сделать это вручную, связав *произвольное* имя устройства с известными нам двумя номерами (старшим и младшим):

```
$ sudo mknod -m0666 /dev/abc c 235 0
$ ls -l /dev/abc
crw-rw-rw- 1 root root 235, 0 июн 14 19:30 /dev/abc
$ cat /dev/abc
Hello, world!
```

Экспериментируя с модулем, не забываем его периодически выгружать время от времени перед очередным туром экспериментов, причем мы должны как отдельно удалить модуль, так и уничтожить имя устройства:

```
$ sudo rm /dev/abc
$ sudo rmmod fixdev
$ lsmod | grep fix
$
```

Особое внимание обращаем на то, каким образом функция, обрабатывающая запросы `read()`, сообщает вызывающей программе об исчерпании потока доступных данных (признак завершения операции: `EOF`) — это важнейшая функция операции чтения (и в разных реализациях это может быть по-разному). Смотрим по диагностике изнутри модуля на то, как стандартная утилита `cat` запрашивала его данные, и что она получала в результате:

```
$ ls -l /dev/abc
crw-rw-rw- 1 root root 235, 0 июн 14 20:32 /dev/abc
$ cat /dev/abc
Hello, world!
$ dmesg | tail -n5
[35171.409549] ========== module installed 235:0 ==============
[35277.402110] === read : 131072
[35277.402117] === read return : 14
[35277.402137] === read : 131072
[35277.402138] === read return : 0
```

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

```
$ sudo insmod fixdev.ko
$ dmesg | tail -n1
[ 7425.221782] ========== module installed 235:0 ==============
$ sudo mknod -m0666 /dev/abc0 c 235 0
$ sudo mknod -m0666 /dev/abc1 c 235 1
$ sudo mknod -m0666 /dev/abc2 c 235 2
$ sudo mknod -m0666 /dev/abc3 c 235 3
```

```
$ cat /dev/abc*
Hello, world!
Hello, world!
Hello, world!
cat: /dev/abc3: Нет такого устройства или адреса
```

Мы создали 4 устройства с последовательными младшими номерами, но только 3 из них обслуживаются только что созданным нами драйвером. А это и есть именно то, чего мы добивались!

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

Интересно, что система, вообще-то говоря, *не различает*, на каком пути (каталоге) мы создаем имя устройства, и это не обязательно должно быть /dev, — смысл команды mknod состоит просто в связывании произвольного имени в файловой системе с парой чисел: старшим и младшим номерами устройства. Создадим имя «устройства» для все того же модуля прямо в рабочем каталоге отработки самого модуля (например):

```
$ pwd
/home/olej/2022/own.BOOKs/BHV.kernel/examples/dev/cdev
$ cat /proc/devices | grep my_
235 my_cdev_dev
$ sudo mknod -m0666 ./z2 c 235 2
$ ls -l | grep ^c
crw-rw-rw- 1 root root 235, 2 июн 15 12:54 z2
$ cat ./z2
Hello, world!
```

Подобные трюки могут оказаться полезными в прикладных проектах — например, для временного создания (скрытых) псевдоустройств, чтобы не загружать общесистемный каталог /dev.

Использование *udev*

Со времени создания файловой системы /sys (sysfs) каждое действие, связанное с устройствами (горячее подключение, отключение) или модулями ядра (загрузка, выгрузка), вызывает целый всплеск асинхронных *широковещательных* уведомлений через дейтаграммный сокет специально для этого случая спроектированного

протокола обмена netlink. Для большей ясности: *любой* пользовательский процесс может создать такой сокет вызовом вида:

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

> ***Примечание***
>
> Отметим, что такой вид сокета и обмен через него практически нигде не упоминаются и не описаны в общих публикациях по сетевому программированию — это исключительно «придумка» Linux.

И любой пользовательский процесс может обрабатывать информацию, передаваемую **из** *ядра* через сокет netlink, и реагировать на эту информацию. Но программа (демон) пользовательского пространства udevd (подсистема udev[1]) делает это по умолчанию, даже если никто другой такую информацию не обрабатывает. В этом и смысл *широковещания*:

```
$ ps -A | grep udevd
    715 ?        00:00:01 systemd-udevd
```

После загрузки обсуждавшегося ранее модуля fixdev.ko мы можем увидеть активную взаимосвязь этого действа с каталогом /sys, например:

```
$ cat /proc/devices | grep my_cdev_dev
235 my_cdev_dev
$ lsmod | head -n2
Module                  Size  Used by
fixdev                 16384  0
$ ls -l /sys/module/fixdev
итого 0
-r--r--r-- 1 root root 4096 июн 15 13:31 coresize
drwxr-xr-x 2 root root    0 июн 15 13:31 holders
-r--r--r-- 1 root root 4096 июн 15 13:31 initsize
-r--r--r-- 1 root root 4096 июн 15 13:31 initstate
drwxr-xr-x 2 root root    0 июн 15 13:31 notes
drwxr-xr-x 2 root root    0 июн 15 13:31 parameters
-r--r--r-- 1 root root 4096 июн 15 13:31 refcnt
drwxr-xr-x 2 root root    0 июн 15 13:31 sections
-r--r--r-- 1 root root 4096 июн 15 13:31 srcversion
-r--r--r-- 1 root root 4096 июн 15 13:31 taint
--w------- 1 root root 4096 июн 15 12:02 uevent
-r--r--r-- 1 root root 4096 июн 15 13:31 version
$ cat /sys/module/fixdev/parameters/major
235
```

Характерно, что здесь мы можем извлечь значение параметра модуля major, даже в том случае, когда мы загружаем модуль без явного указания этого параметра при

[1] На сегодня подсистема udev, как и многие другие, «отдана» под управление systemd, но от этого сущность udev никак не поменялась.

загрузке (что и было сделано специально в показанном примере). Таким образом, через /sys мы можем извлекать все интересующие нас параметры загрузки модулей.

Но возвратимся к асинхронным уведомлениям через дейтаграммный сокет протокола netlink. Увидеть содержимое уведомлений, передаваемых от ядра к демону udevd, мы можем, запустив на время наблюдаемого события (подключение/отключение устройства, загрузка/выгрузка модуля) *в отдельном терминале* программу udevadm. Загрузите и выгрузите (любой) модуль:

```
$ sudo insmod fixdev.ko
$ sudo rmmod fixdev
```

И в мониторе udevadm вы увидите:

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

KERNEL[13886.300015] add      /module/fixdev (module)
ACTION=add
DEVPATH=/module/fixdev
SUBSYSTEM=module
SEQNUM=10311

UDEV  [13886.303232] add      /module/fixdev (module)
ACTION=add
DEVPATH=/module/fixdev
SUBSYSTEM=module
SEQNUM=10311
USEC_INITIALIZED=13886303097

KERNEL[13899.353422] remove   /module/fixdev (module)
ACTION=remove
DEVPATH=/module/fixdev
SUBSYSTEM=module
SEQNUM=10312

UDEV  [13899.357567] remove   /module/fixdev (module)
ACTION=remove
DEVPATH=/module/fixdev
SUBSYSTEM=module
SEQNUM=10312
USEC_INITIALIZED=13899353504
...
```

Здесь показан именно момент загрузки и выгрузки тестового модуля fixdev.ko (это напоминает по смыслу для сокета netlink аналогию с тем, как это же делает общеизвестный сетевой снифер tcpdump для сетевого трафика IP-протоколов).

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

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

Или непосредственно *создавая имя* устройства в /dev (что бывает гораздо чаще):

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

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

Вот еще один альтернативный способ извлечения параметров *события* (а это может оказаться иногда не так просто) для успешного написания эффективных *правил* udev (при загруженном модуле ядра):

```
$ sudo insmod fixdev.ko
$ udevadm info --path=/module/fixdev --export
P: /module/fixdev
L: 0
E: DEVPATH=/module/fixdev
E: SUBSYSTEM=module
```

Подсистема udev все еще находится в активном развитии и изменении, хотя основные принципы ее и устоялись. Тем не менее *синтаксис* составления ее правил все еще изменяется (публикации 2012 года, например, описывают другой синтаксис — им руководствоваться нельзя). Поэтому обо всём, что касается синтаксиса записи правил udev, нужно справиться накануне их написания:

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

Но общая схема при всех детализациях синтаксиса записи правил остается неизменной. Это на сегодня основная схема создания символьных устройств в Linux.

Динамические имена

Вариацией на тему использования того же API является вариант предыдущего модуля (в том же каталоге /dev/cdev), но динамически создающий имя устройства в каталоге /dev с заданным старшим и младшим номерами (это обеспечивается использованием в ручном режиме механизмов системы sysfs, рассмотренных ранее). Далее показаны только принципиальные различия (дополнения) относительно предыдущего варианта:

dyndev.c

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

static struct cdev hcdev;
static struct class *devclass;

static int __init dev_init(void) {
   int ret, i;
   dev_t dev;
   if(major != 0) {
      dev = MKDEV(major, DEVICE_FIRST);
      ret = register_chrdev_region(dev, DEVICE_COUNT, MODNAME);
   }
   else {
      ret = alloc_chrdev_region(&dev, DEVICE_FIRST, DEVICE_COUNT, MODNAME);
      major = MAJOR(dev);  // не забыть зафиксировать!
   }
   if(ret < 0) {
      printk(KERN_ERR "=== Can not register char device region\n");
      goto err;
   }
   cdev_init(&hcdev, &dev_fops);
   hcdev.owner = THIS_MODULE;
   ret = cdev_add(&hcdev, dev, DEVICE_COUNT);
   if(ret < 0) {
      unregister_chrdev_region(MKDEV(major, DEVICE_FIRST), DEVICE_COUNT);
      printk(KERN_ERR "=== Can not add char device\n");
      goto err;
   }
   devclass = class_create(THIS_MODULE, "dyn_class"); /* struct class* */
   for(i = 0; i < DEVICE_COUNT; i++) {
```

```
#define DEVNAME "dyn"
      dev = MKDEV(major, DEVICE_FIRST + i);
#if LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,26)
/* struct device *device_create(struct class *cls, struct device *parent,
                               dev_t devt, const char *fmt, ...); */
      device_create(devclass, NULL, dev, "%s_%d", DEVNAME, i);
#else              // прототип device_create() изменился!
/* struct device *device_create(struct class *cls, struct device *parent,
                               dev_t devt, void *drvdata, const char *fmt, ...); */
      device_create(devclass, NULL, dev, NULL, "%s_%d", DEVNAME, i);
#endif
   }
   printk(KERN_INFO "======== module installed %d:[%d-%d] ==========\n",
          MAJOR(dev), DEVICE_FIRST, MINOR(dev));
err:
   return ret;
}

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

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

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

Теперь исчезает необходимость вручную создавать имя устройства в /dev и отслеживать соответствие его номеров — соответствующие имена возникают автоматически после загрузки модуля. Вот исчерпывающая диагностика такого модуля:

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

```
Module                  Size  Used by
dyndev                 16384  0
$ dmesg | tail -n1
[18699.439749] ======== module installed 234:[0-2] ===========
$ ls -l /dev/dyn*
crw------- 1 root root 234, 0 июн 15 15:10 /dev/dyn_0
crw------- 1 root root 234, 1 июн 15 15:10 /dev/dyn_1
crw------- 1 root root 234, 2 июн 15 15:10 /dev/dyn_2
$ sudo cat /dev/dyn_2
Hello, world!
$ ls -ld /sys/class/d*
drwxr-xr-x 2 root root 0 июн 15 09:59 /sys/class/dax
drwxr-xr-x 2 root root 0 июн 15 09:59 /sys/class/devcoredump
drwxr-xr-x 2 root root 0 июн 15 09:59 /sys/class/devfreq
drwxr-xr-x 2 root root 0 июн 15 09:59 /sys/class/devfreq-event
drwxr-xr-x 2 root root 0 июн 15 09:59 /sys/class/dma
drwxr-xr-x 2 root root 0 июн 15 09:59 /sys/class/dmi
drwxr-xr-x 2 root root 0 июн 15 09:59 /sys/class/drm
drwxr-xr-x 2 root root 0 июн 15 09:59 /sys/class/drm_dp_aux_dev
drwxr-xr-x 2 root root 0 июн 15 15:10 /sys/class/dyn_class
$ ls -l /sys/class/dyn*
итого 0
lrwxrwxrwx 1 root root 0 июн 15 15:13 dyn_0 -> ../../devices/virtual/dyn_class/dyn_0
lrwxrwxrwx 1 root root 0 июн 15 15:13 dyn_1 -> ../../devices/virtual/dyn_class/dyn_1
lrwxrwxrwx 1 root root 0 июн 15 15:13 dyn_2 -> ../../devices/virtual/dyn_class/dyn_2
$ tree /sys/class/dyn_class/dyn_0
/sys/class/dyn_class/dyn_0
├── dev
├── power
│   ├── async
│   ├── autosuspend_delay_ms
│   ├── control
│   ├── runtime_active_kids
│   ├── runtime_active_time
│   ├── runtime_enabled
│   ├── runtime_status
│   ├── runtime_suspended_time
│   └── runtime_usage
├── subsystem -> ../../../../class/dyn_class
└── uevent

2 directories, 11 files
$ cat /proc/modules | grep dyn
dyndev 16384 0 - Live 0x0000000000000000 (OE)
$ sudo cat /proc/modules | grep dyn
dyndev 16384 0 - Live 0xffffffffc0dc6000 (OE)
$ cat /proc/devices | grep dyn
234 my_dyndev_mod
```

И точно так же имена в /dev ликвидируются после выгрузки модуля:

```
$ ls -l /dev/dyn*
crw------- 1 root root 234, 0 июн 15 15:10 /dev/dyn_0
crw------- 1 root root 234, 1 июн 15 15:10 /dev/dyn_1
crw------- 1 root root 234, 2 июн 15 15:10 /dev/dyn_2
$ sudo rmmod dyndev
$ dmesg | tail -n1
[19687.748235] ============== module removed ==================
$ ls -l /dev/dyn*
ls: невозможно получить доступ к '/dev/dyn*': Нет такого файла или каталога
```

Этот же модуль может задействоваться и для создания диапазона устройств с принудительным указанием `major` (если использование этого номера возможно с точки зрения системы), но с динамическим созданием имен таких устройств. Мы специально запустим этот драйвер с `major=300`, чтобы опровергнуть бытующее заблуждение, идущее от ранних версий ядра Linux, что числа `major` и `minor` имеют байтовую размерность и диапазон [0...255]:

```
$ sudo insmod dyndev.ko major=300
$ cat /proc/devices | grep dyn
300 my_dyndev_mod
$ dmesg | tail -n1
[19816.725419] ======== module installed 300:[0-2] ==========
$ ls -l /dev | grep 300
crw-------   1 root root      300,   0 июн 15 15:29 dyn_0
crw-------   1 root root      300,   1 июн 15 15:29 dyn_1
crw-------   1 root root      300,   2 июн 15 15:29 dyn_2
drwxr-xr-x   4 root root           300 июн 15 09:59 input
$ sudo cat /dev/dyn_1
Hello, world!
```

Отметим: в версии ядра 2.6.32 все происходит в точности, как и предполагалось. Но в версии ядра 3.13 (я не могу пока сказать, по какой точно версии здесь проходит раздел) простое прямое чтение не удастся из-за *прав* (флагов доступа), с которым и создаются устройства в /dev:

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

Легко убедиться, что эта сложность снимается **ручным** изменением флагов, но это совершенно не удобный способ:

```
$ sudo chmod 0444 /dev/dyn*
$ ls -l /dev/dyn*
```

```
cr--r--r-- 1 root root 246, 0 апр 22 13:45 /dev/dyn_0
cr--r--r-- 1 root root 246, 1 апр 22 13:45 /dev/dyn_1
cr--r--r-- 1 root root 246, 2 апр 22 13:45 /dev/dyn_2
$ cat /dev/dyn_2
Hello, world!
```

> ***Примечание***
>
> В предварительных обсуждениях рукописи с читателями неоднократно отмечалось, что изменение флагов устройства можно (нужно?) устанавливать (изменять) в правилах подсистемы `udev` для этих устройств. Но если прибегать к `udev`, то тогда мы можем и вообще создавать устройства с ее правилами, вовсе не прибегая к услугам `device_create()`. Наверняка, флаги создания имени устройства *по умолчанию* могут и должны переопределяться где-то в атрибутной записи устройства... но пока я не готов сказать, где именно, и оставляю это для экспериментов читателям.

Описывамое динамическое создание устройств сильно упрощает работу над драйвером. Но всегда ли хорош такой способ распределения номеров устройств? Все зависит от решаемой задачи. Если номера реального *физического устройства* (изделия) в системе «гуляют» от одного экземпляра компьютера к другому и даже просто при изменениях в конфигурациях системы, то это вряд ли понравится разработчикам этого устройства. С другой стороны, во множестве задач удобно создавать псевдоустройства — некоторые моделирующие сущности для каналов ввода/вывода. Для таких случаев совершенно уместным будет полностью динамическое распределение параметров этих устройств. Одним из красивых иллюстрирующих примеров, поясняющих сказанное, является ставший уже стандартом дефакто интерфейс zaptel/DAHDI к оборудованию передачи VoIP, используемый во многих проектах коммутаторов IP-телефонии: Asterisk, FreeSWITCH, YATE и других. В архитектуре этой драйверной подсистемы для синхронных цифровых линий связи E1/T1/J1 (и E3/T3/J3) создается много (возможно, до нескольких сотен) фиктивных устройств-имен в /dev, взаимно однозначно соответствующих *виртуальным* каналам временно́го уплотнения (тайм-слотам) реальных линий связи. Вся дальнейшая работа с созданными динамическими именами множества устройств обеспечивается традиционными POSIX `read()` или `write()` — в точности так же, как это делается с реальным физическим оборудованием.

Разнородные (смешанные) устройства

Практика динамически перераспределяемых псевдоустройств приобрела настолько широкое распространение, что для упрощения реализации таких устройств был предложен специальный интерфейс (<linux/miscdevice.h>). Эту технику регистрации драйвера устройства часто называют в литературе misc drivers (от miscellaneous, разнородный) — интерфейс смешанных, разнородных устройств. Совершенно нет необходимости создавать целую группу устройств, когда вам необходимо создать единственное специфическое устройство. Примером таких индивидуальных устройств являются, например, традиционные в Linux:

```
$ ls -l /dev/{null,zero,random}
crw-rw-rw- 1 root root 1, 3 июн 15 09:59 /dev/null
```

```
crw-rw-rw- 1 root root 1, 8 июн 15 09:59 /dev/random
crw-rw-rw- 1 root root 1, 5 июн 15 09:59 /dev/zero
```

Техника misc drivers — это *самая простая* в кодировании техника регистрации символьного устройства. Каждое такое устройство создается с единым `major` значением 10, но может выбирать свой уникальный `minor` (который либо задается принудительно, либо устанавливается системой). В этом варианте драйвер регистрируется и создает символическое имя устройства в /dev одним-единственным вызовом `misc_register()` (см. папку chdev/misc в сопровождающем книгу файловом архиве):

misc_dev.c

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

static int minor = -1;
module_param(minor, int, S_IRUGO);

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

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

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

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

Вот, собственно, и весь код драйвера. Вызов `misc_register()` регистрирует *единичное* устройство с одним значением `minor`, определенным в `struct miscdevice`. Поэтому, если драйвер предполагает обслуживать группу однотипных устройств, различающихся по `minor`, то это не есть лучший выбор для использования. Хотя, конечно,

драйвер может поочередно зарегистрировать *несколько* структур `struct miscdevice`. Вот как вся эта теория выглядит в примере:

```
$ sudo insmod misc_dev.ko
$ lsmod | head -n2
Module                  Size  Used by
misc_dev               16384  0
$ cat /proc/devices  | grep misc
 10 misc
$ ls -l /dev/*misc*
crw------- 1 root root 10, 55 июн 15 16:41 /dev/own_misc_dev
$ sudo cat /dev/own_misc_dev
Hello, world!
$ sudo rmmod misc_dev
$ ls -l /dev/*misc*
ls: невозможно получить доступ к '/dev/*misc*': Нет такого файла или каталога
```

Операционная система (и прикладные проекты — как, например, Oracle VirtualBox в коде, показанном далее, — имена вида /dev/vbox*) регистрирует с `major` значением 10 достаточно много разносортных устройств:

```
$ ls -l /dev | grep 10,
crw-r--r--   1 root root       10, 235 июн 15 09:59 autofs
crw-rw----   1 root disk       10, 234 июн 15 09:59 btrfs-control
crw-------   1 root root       10,  59 июн 15 09:59 cpu_dma_latency
crw-------   1 root root       10, 203 июн 15 09:59 cuse
crw-------   1 root root       10,  62 июн 15 09:59 ecryptfs
crw-rw-rw-   1 root root       10, 229 июн 15 09:59 fuse
crw-------   1 root root       10, 228 июн 15 09:59 hpet
crw-------   1 root root       10, 183 июн 15 09:59 hwrng
crw-rw----+  1 root kvm        10, 232 июн 15 09:59 kvm
crw-rw----   1 root disk       10, 237 июн 15 09:59 loop-control
crw-------   1 root root       10, 227 июн 15 09:59 mcelog
crw-------   1 root root       10, 144 июн 15 09:59 nvram
crw-------   1 root root       10,   1 июн 15 09:59 psaux
crw-rw-r--+  1 root netdev     10, 242 июн 15 09:59 rfkill
crw-------   1 root root       10, 231 июн 15 09:59 snapshot
crw-rw----   1 root kvm        10,  60 июн 15 09:59 udmabuf
crw-------   1 root root       10, 239 июн 15 09:59 uhid
crw-------   1 root root       10, 223 июн 15 09:59 uinput
crw-------   1 root root       10, 240 июн 15 09:59 userio
crw-------   1 root root       10,  58 июн 15 09:59 vboxdrv
crw-rw-rw-   1 root root       10,  57 июн 15 09:59 vboxdrvu
crw-------   1 root root       10,  56 июн 15 09:59 vboxnetctl
crw-------   1 root root       10,  63 июн 15 09:59 vga_arbiter
crw-------   1 root root       10, 137 июн 15 09:59 vhci
crw-------   1 root root       10, 238 июн 15 09:59 vhost-net
crw-------   1 root root       10, 241 июн 15 09:59 vhost-vsock
crw-------   1 root root       10, 249 июн 15 09:59 zfs
```

Все такие устройства (смешанные, разнородные) регистрируются в sysfs в *едином* подклассе (под именем из модуля) в классе `misc` — где среди них находится и созданный нами модуль:

```
$ ls /sys/class/misc -w80
autofs           fuse       loop-control  rfkill    vboxdrv
btrfs-control    hpet       mcelog        snapshot  vboxdrvu
cpu_dma_latency  hw_random  microcode     tun       vboxnetctl
device-mapper    kvm        own_misc_dev  udmabuf   vfio
ecryptfs         lightnvm   psaux         uinput    vga_arbiter
$ ls /sys/class/misc/own_misc_dev
dev  power  subsystem  uevent
$ tree /sys/devices/virtual/misc/own_misc_dev/
/sys/devices/virtual/misc/own_misc_dev/
├── dev
├── power
│   ├── async
│   ├── autosuspend_delay_ms
│   ├── control
│   ├── runtime_active_kids
│   ├── runtime_active_time
│   ├── runtime_enabled
│   ├── runtime_status
│   ├── runtime_suspended_time
│   └── runtime_usage
├── subsystem -> ../../../../class/misc
└── uevent

2 directories, 11 files
```

Этот модуль может использовать и принудительно задаваемый номер `minor` устройства (если он не совпадает с `minor` уже загруженного устройства этого же типа с `major=10`):

```
$ sudo insmod misc_dev.ko minor=60
insmod: ERROR: could not insert module misc_dev.ko: Device or resource busy
$ dmesg | tail -n1
[25656.480808] === Unable to register misc device

$ sudo insmod misc_dev.ko minor=0
$ ls -l /dev/*misc*
crw------- 1 root root 10, 0 июн 15 17:07 /dev/own_misc_dev
$ sudo cat /dev/own_misc_dev
Hello, world!
```

Управляющие операции устройства

То, что рассматривалось до сих пор, делалось на примере единственной операции `read()`. Операция `write()`, как понятно интуитивно, симметричная и реализуется так

же. Она пока не включалась в обсуждение, чтобы не перегружать текст, и будет показана позже — при рассмотрении операций с /proc. Но, кроме этих операций, которые часто упоминают как операции ввода/вывода *в основном потоке* данных, в Linux достаточно широко используется операция `ioctl()`, применяемая для управления устройством и осуществляющая обмен с устройством вне основного потока данных (иногда говорят об *обмене внеполосовыми данными*).

В следующем примере мы обсудим (см. папку tools/ioctl в сопровождающем книгу файловом архиве) реализацию таких управляющих операций. Но для реализации операций *регистрации* устройства воспользуемся попутно так называемым *старым методом* регистрации символьного устройства (`register_chrdev()`). Эта техника не потеряла актуальности и используется сегодня — это и будет наш *третий*, последний, *альтернативный* способ *регистрации* модуля драйвера любого символьного устройства:

ioctl_dev.c

```
#include "ioctl.h"
#include "../dev.h"
#include <linux/version.h>      /* LINUX_VERSION_CODE */

#define HELLO_MAJOR 200

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

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

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,35)
static long dev_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {
#else
static int dev_ioctl(struct inode *n, struct file *f,
                     unsigned int cmd, unsigned long arg) {
#endif
   if((_IOC_TYPE(cmd) != IOC_MAGIC)) return -EINVAL;
   switch(cmd) {
      case IOCTL_GET_STRING:
         if(copy_to_user((void*)arg, hello_str, _IOC_SIZE(cmd))) return -EFAULT;
         break;
```

```
      default:
         return -ENOTTY;
   }
   return 0;
}

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

#define HELLO_MODNAME "my_ioctl_dev"

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

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

Как видно по препроцессорным условным определениям, вид кода будет отличаться на границе версии ядра 2.6.35, — и это мы обсудим вскоре.

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

ioctl.h

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

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

Для единообразного определения кодов в `ioctl`-командах используются (см. файл определений /usr/include/asm-generic/ioctl.h в пространстве пользователя) макросы следующего вида (показаны основные и еще некоторые):

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

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

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

ioctl.c

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

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

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

Испытаем полученное устройство:

```
$ sudo insmod ioctl_dev.ko
$ cat /proc/devices | grep ioctl
200 my_ioctl_dev
242 megaraid_sas_ioctl
$ ls -l /dev/ioctl
crw-rw-rw- 1 root root 200, 0 июн 15 18:25 /dev/ioctl
$ cat /dev/ioctl
Hello, world!
$ ./ioctl
Hello, world!
```

Что произошло в двух последних командах? Мы читаем данные из устройства, но двумя совершенно разными операциями (в пространстве пользователя): `read()` — в случае утилиты `cat`, и `ioctl()` — когда мы читаем из собственного приложения ioctl (то, что операции читают совпадающие данные, ничего не опровергает — они для упрощения читают совершенно разными механизмами, но из одной и той же области данных). В коде модуля запрос `read()` обеспечивает функция `dev_read()` (из файла ../dev.h), а запрос `ioctl()` — функция `dev_ioctl()`.

Обратим внимание на одну особенность, которая была вскользь отмечена ранее: регистрируя устройство вызовом `register_chrdev()`, драйвер регистрирует только номер `major` и получает под свой контроль *весь* диапазон [0...255] номеров `minor`:

```
$ sudo mknod -m0666 /dev/ioctl5 c 200 5
$ sudo mknod -m0666 /dev/ioctl255 c 200 255
$ sudo mknod -m0666 /dev/ioctl256 c 200 256
$ sudo mknod -m0666 /dev/ioctl10000 c 200 10000
$ ls -l /dev/ioctl*
crw-rw-rw- 1 root root 200, 10000 июн 15 18:55 /dev/ioctl10000
crw-rw-rw- 1 root root 200,   255 июн 15 19:14 /dev/ioctl255
crw-rw-rw- 1 root root 200,   256 июн 15 19:13 /dev/ioctl256
crw-rw-rw- 1 root root 200,     5 июн 15 19:00 /dev/ioctl5
$ cat /dev/ioctl5
Hello, world!
$ cat /dev/ioctl255
Hello, world!
$ cat /dev/ioctl256
cat: /dev/ioctl256: Нет такого устройства или адреса
$ cat /dev/ioctl10000
cat: /dev/ioctl10000: Нет такого устройства или адреса
```

Хотя ранее уже было сказано, что байтовый размер и диапазон [0...255] — это рудимент старых соглашений UNIX/Linux и на сегодня под младший номер выделяется 20 битов из 32 (оставшиеся 12 битов — это старший номер), но наш — третий — метод регистрации символьного устройства `register_chrdev()` (потому он и назван «старый») регистрирует именно группу из 256 младших номеров за устройством.

Сам номер устройства в группе [0...255] в этом случае регистрации устройства должен перехватываться в самом коде модуля, в обработчике операции `open()` по своему *первому* полученному параметру `struct inode*`:

```
static int dev_open(struct inode *n, struct file *f) {
     printk(KERN_INFO "=== open device %u:%u\n", HELLO_MAJOR, iminor(n));
     ...
```

Вот как это выглядит:

```
$ cat /dev/ioctl5
Hello, world!
$ dmesg | tail -n5
[33249.800620] === open device 200:5
[33249.800641] === read : 131072
[33249.800645] === read return : 14
[33249.800661] === read : 131072
[33249.800662] === read return : 0
$ cat /dev/ioctl255
Hello, world!
$ dmesg | tail -n5
[33331.065008] === open device 200:255
[33331.065032] === read : 131072
[33331.065037] === read return : 14
[33331.065056] === read : 131072
[33331.065057] === read return : 0
```

Тот код, который показан (как вариант) в файле ioctl_dev.old.c, был работоспособным в ядрах до версии 2.6.34. А в версии 2.6.35 (и везде далее) указатель функции (поле) `ioctl` в таблице `struct file_operations` был разделен на два обработчика: `unlocked_ioctl()` и `compat_ioctl()`. Прежний обработчик был описан с прототипом:

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

Для новых обработчиков прототип разделен на два и *изменен*:

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

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

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

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

ioctl_dev.new.c

```
...

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

static const struct file_operations hello_fops = {
   ...
   .unlocked_ioctl = dev_ioctl
};
```

При этом весь остальной код примера останется в неизменном виде.

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

В архиве примеров имеется объединенная версия (см. файл ioctl_dev.c), работоспособная для любых ядер за счет использования условной препроцессорной компиляции:

ioctl_dev.c

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

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

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

Множественное открытие устройства

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

- драйвер вообще никак *не контролирует* возможности параллельного использования (то, что было показано во всех рассмотренных ранее примерах);
- драйвер допускает только *единственное* открытие устройства — попытки параллельного открытия будут завершаться ошибкой до тех пор, пока использующий устройство процесс (блокирующий) не завершит свою операцию и не закроет устройство. Это совершенно реалистический сценарий — например, для устройств передачи данных по *физическим* линиям передачи (Modbus, CAN и др.);
- драйвер допускает много *параллельных* сессий использования устройства. При этом драйвер должен реализовать индивидуальный *экземпляр* данных для каждой копии открытого устройства. Это очень частый случай для *псевдоустройств* — логических устройств, реализующих на программном уровне *модель* устройства. Примерами их могут служить, например, устройства /dev/random, /dev/zero, /dev/null или, скажем, «точки подключения данных» в разнообразных SCADA-системах при проектировании АСУТП.

Подробно это проще рассмотреть на примере (см. папку chdev/mopen в сопровождающем книгу файловом архиве). Мы создаем модуль, реализующий все три названных варианта (а то, какой вариант он будет использовать, определяется параметром `mode` запуска модуля — соответственно `0`, `1` или `2`):

mmopen.c

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");
MODULE_VERSION("7.5");

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

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

static int dev_open = 0;

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

static int mopen_open(struct inode *n, struct file *f) {
   LOG("open - node: %p, file: %p, refcount: %ld", n, f, (long)module_refcount(THIS_MODULE));
   if(dev_open) {
      LOG("device /dev/%s is busy", DEVNAM);
      return -EBUSY;
   }
   if(1 == mode) dev_open++;
   if(2 == mode) {
      struct mopen_data *data;
      f->private_data = kmalloc(sizeof(struct mopen_data), GFP_KERNEL);
      if(NULL == f->private_data) {
         LOG("memory allocation error");
         return -ENOMEM;
      }
      data = (struct mopen_data*)f->private_data;
      strcpy(data->buf, "dynamic: not initialized!");  // динамический буфер
      data->odd = 0;
   }
   return 0;
}
```

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

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

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

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

static const struct file_operations mopen_fops = {
   .owner  = THIS_MODULE,
   .open =    mopen_open,
```

```
   .release = mopen_release,
   .read    =  mopen_read,
   .write   =  mopen_write,
};

static struct miscdevice mopen_dev = {
   MISC_DYNAMIC_MINOR, DEVNAM, &mopen_fops
};

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

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

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

Для тестирования полученного модуля мы воспользуемся стандартными командами чтения и записи устройства: `cat` и `echo`, но их нам будет недостаточно, и мы запустим сделанное на этот случай тестовое приложение, которое выполняет *одновременное* открытие двух дескрипторов нашего устройства и делает на них поочередные 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 <unistd.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       // длина 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 открытие устройства (параметр mode здесь можно не указывать — это значение по умолчанию, и я делаю это только для наглядности)[2]:

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

Записываем на устройство произвольную символьную строку:

```
$ sudo echo 777777777 > /dev/mopen
bash: /dev/mopen: Отказано в доступе
$ ls -l /dev/mopen
crw------- 1 root root 10, 55 июн 15 20:30 /dev/mopen
$ echo 777777777 | sudo tee /dev/mopen
777777777
$ sudo cat /dev/mopen
777777777
$ sudo ./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 (последовательными) утилитами POSIX (echo и 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 безуспешными:

---

[2] Мы используем для записи форму команды: echo … | sudo tee … — потому что устройство (по флагам доступа) требует привилегий root, но простое перенаправление echo … > в рассматриваемом случае здесь не сработает. Того же можно добиться, выполняя это тестирование в терминале, открытом от имени root. Или вручную изменить флаги доступа к устройству командой chmod.

```
$ sudo insmod ./mmopen.ko debug=1 mode=1
$ sudo cat /dev/mopen
static: not initialized!
$ echo 777777777 | sudo tee /dev/mopen
777777777
$ sudo cat /dev/mopen
777777777
$ sudo ./pmopen
prepared 7 bytes: 1111111
open device error: Device or resource busy
write device error: Bad file descriptor
-----------------------------------
read 8 bytes: 1111111

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

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

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

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

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

```
$ sudo insmod ./mmopen.ko mode=2 debug=1
$ sudo cat /dev/mopen
dynamic: not initialized!
$ echo 987654321 | sudo tee /dev/mopen
987654321
$ dmesg | tail -n10
```

```
[52608.573231] ! installed device /dev/mopen in mode 2
[52644.371784] ! open - node: 00000000a0d09f6b, file: 0000000052cd11a5, refcount: 1
[52644.371796] ! read - file: 0000000052cd11a5, read from 000000007a95e1ed bytes 131072;
refcount: 1
[52644.371798] ! return bytes :  26
[52644.371809] ! read - file: 0000000052cd11a5, read from 000000007a95e1ed bytes 131072;
refcount: 1
[52644.371810] ! return : EOF
[52644.371821] ! close - node: 00000000a0d09f6b, file: 0000000052cd11a5, refcount: 1
[52669.029744] ! open - node: 00000000a0d09f6b, file: 00000000442313cf, refcount: 1
[52669.029790] ! write - file: 00000000442313cf, write to 000000002ab3bcd3 bytes 10; refcount: 1
[52669.029792] ! put bytes : 10
```

Каждая тестовая операция `echo` и `cat` открывает *свой экземпляр* устройства, выполняет требуемую операцию и закрывает устройство. Следующая выполняемая команда работает с *совершенно другим экземпляром* устройства и соответственно с другой копией данных! Поэтому именно то, для чего мы готовили драйвер, — множественное открытие дескрипторов устройства — срабатывает отменно:

```
$ sudo ./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
-----------------------------------
$ dmesg | tail -n15
[52953.484737] ! open - node: 00000000a0d09f6b, file: 00000000e3632089, refcount: 1
[52953.484748] ! write - file: 00000000e3632089, write to 00000000cf2a1986 bytes 7; refcount: 1
[52953.484750] ! put bytes : 7
[52953.484930] ! open - node: 00000000a0d09f6b, file: 00000000af8dd9cb, refcount: 2
[52953.484936] ! write - file: 00000000af8dd9cb, write to 000000009e6cd3eb bytes 5; refcount: 2
[52953.484937] ! put bytes : 5
[52953.484950] ! read - file: 00000000e3632089, read from 00000000cf2a1986 bytes 256; refcount: 2
[52953.484952] ! return bytes :  8
[52953.484961] ! read - file: 00000000e3632089, read from 00000000cf2a1986 bytes 256; refcount: 2
[52953.484962] ! return : EOF
[52953.484992] ! read - file: 00000000af8dd9cb, read from 000000009e6cd3eb bytes 256; refcount: 2
[52953.484993] ! return bytes :  6
[52953.485005] ! read - file: 00000000af8dd9cb, read from 000000009e6cd3eb bytes 256; refcount: 2
[52953.485005] ! return : EOF
[52953.485018] ! close - node: 00000000a0d09f6b, file: 00000000e3632089, refcount: 2
```

В итоге этого рассмотрения нельзя не задаться вопросом: всегда ли последний вариант (`mode=2`) лучше других (`mode=0` или `mode=1`)? Это не так, и категорично это 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 надежности и живучести (но и само это утверждение тоже может вызывать сомнения). В любом случае способ открытия устройства может реализоваться по самым различным алгоритмам и должен соответствовать логике решаемой задачи. Он накладывает требования на реализацию всех прочих операций на устройстве и в итоге заслуживает самой пристальной проработки при начале нового проекта.

Счетчик ссылок использования модуля

О счетчике ссылок использования модуля и о том, что это один из важнейших элементов контроля безопасности загрузки модулей и целостности ядра системы, уже было сказано ранее. Вернемся еще раз к вопросу счетчика ссылок использования модуля и на примере только что спроектированного модуля внесем для себя окончательную ясность в этот вопрос. Для этого изготовим еще 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 которых будет ожидать ввода с терминала. Загрузим модуль со следующими его параметрами загрузки:

```
$ lsmod | head -n2
Module                  Size  Used by
mmopen                 16384  0
$ ls /sys/module/mmopen/parameters
debug  mode
$ cat /sys/module/mmopen/parameters/*
1
2
```

Запустим три копии тестового процесса:

```
$ sudo ./simple
>
$ sudo ./simple
>
$ sudo ./simple
>
```

То, что показано, выполняется на четырех независимых терминалах, и это достаточно сложно объяснить в линейном протоколе, но будем считать, что мы оставили три тестовых процесса заблокированными на ожидании ввода строки (символ приглашения >). Выполним в этом месте:

```
$ lsmod | grep mmopen
mmopen                 16384  3
```

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

```
$ dmesg | tail -n9
[54303.569834] ! open - node: 00000000a0d09f6b, file: 00000000ea599a59, refcount: 1
[54319.257250] ! open - node: 00000000a0d09f6b, file: 0000000049c2c7d7, refcount: 2
[54328.069939] ! open - node: 00000000a0d09f6b, file: 00000000918ffeaf, refcount: 3
```

Хорошо видно, как счетчик ссылок использования пробежал диапазон от 0 до 3. После этого введем строки (разной длины) на двух копиях тестового процесса, а последний завершим по комбинации клавиш <Ctrl>+<C> (SIGINT), чтобы знать, как счетчик использования отреагирует на завершение (аварийное) клиента по сигналу:

```
$ sudo ./simple
> 1234
read 5 bytes: 1234
1234
$
$ sudo ./simple
> 5678
read 5 bytes: 5678
5678
$
$ sudo ./simple
> ^C
$
```

И вот что мы примерно увидим в системном журнале в качестве протокола всех этих манипуляций — уменьшение числа ссылок по ходу завершения приложений, запрашивающих доступ к модулю:

```
$ dmesg | tail -n14
[54789.287543] ! write - file: 00000000424e3044, write to 000000008e1b4d52 bytes 5; refcount: 3
[54789.287546] ! put bytes : 5
[54789.287550] ! read - file: 00000000424e3044, read from 000000008e1b4d52 bytes 160; refcount: 3
[54789.287552] ! return bytes :  5
[54789.287600] ! read - file: 00000000424e3044, read from 000000008e1b4d52 bytes 160; refcount: 3
[54789.287601] ! return : EOF
[54789.287610] ! close - node: 00000000a0d09f6b, file: 00000000424e3044, refcount: 3
[54795.351883] ! write - file: 0000000049c2c7d7, write to 00000000f2c7d547 bytes 5; refcount: 2
[54795.351889] ! put bytes : 5
[54795.351908] ! read - file: 0000000049c2c7d7, read from 00000000f2c7d547 bytes 160; refcount: 2
[54795.351909] ! return bytes :  5
[54795.351943] ! read - file: 0000000049c2c7d7, read from 00000000f2c7d547 bytes 160; refcount: 2
[54795.351943] ! return : EOF
[54795.351953] ! close - node: 00000000a0d09f6b, file: 0000000049c2c7d7, refcount: 2
```

И в конечном итоге (после завершения всех запущенных в разных терминалах экземпляров simple):

```
$ lsmod | head -n2
Module                  Size  Used by
mmopen                 16384  0
```

Примечание

На всем протяжении выполнения функции, реализующей операцию release() устройства, счетчик использования еще не декрементирован, т. к. сессия файлового открытия еще не завершена!

Нужно также подчеркнуть еще один момент: из протокола системного журнала следует, что после выполнения `open()` все другие операции из той же таблицы файловых операций (`read()`, `write()`, ...) сами уже никоим образом не влияют на значение счетчика ссылок.

Все это говорит о том, что отслеживание ссылок использования при выполнении `open()` и `close()` (именно они отслеживают число файловых операций, выполняющихся на устройстве) на сегодня корректно выполняется ядром *самостоятельно* (и мне не совсем понятно, каким путем, поскольку мы полностью подменяем реализующие операции для `open()` и `close()`, не оставляя места ни для каких умалчиваемых функций). И еще о том, что неоднократно рекомендуемая в публикациях необходимость корректировки числа ссылок из кода при выполнении обработчиков для `open()` и `close()` на сегодня отпала.

Режимы выполнения операций ввода/вывода

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

> ***Примечание***
>
> Простейшей иллюстрацией блокирующего и неблокирующего режимов ввода могут служить UNIX-операции ввода с терминала: в каноническом режиме терминала (блокирующем) операция `gets()` будет бесконечно ожидать ввода с клавиатуры и его завершения по <Enter>, а в неканоническом режиме операция `getchar()` будет либо считывать введенный символ (если он есть), либо просто «пробегать» операцию ввода (разве что, отмечая для себя признак отсутствия ввода).

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

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

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

Неблокирующий ввод/вывод и мультиплексирование

> ***Примечание***
>
> Показанная в этом разделе реализация сопровождалась автором (редактировалась под изменение версий ядра) только до ядра 3.14. Реставрировать ее под ядра 5.*x* оставлено на самостоятельную проработку, что не должно вызвать больших трудностей.

Здесь имеется в виду реализация поддержки (в модуле, драйвере) операций мультиплексированного ожидания возможности выполнения операций ввода/вывода `select()` и `poll()`. Примеры этого раздела получились много объемнее и сложнее, чем все предыдущие, и в них задействованы механизмы ядра, которые мы еще не затрагивали и которые рассмотрим далее... Но сложность эта обусловлена тем, что здесь мы начинаем вторгаться в обширную и сложную область — неблокирующих и асинхронных операций ввода/вывода. При первом прочтении этот раздел можно пропустить — на него никак не опирается все последующее изложение.

Сложность описания подобных механизмов и написания демонстрирующих их примеров состоит в том, чтобы придумать модель-задачу, которая: а) достаточно адекватно использует рассматриваемый механизм и б) была бы до примитивного простой, чтобы ее код был не громоздким, легко анализировался и мог использоваться для дальнейшего развития. В этом разделе мы реализуем драйвер (см. папку chdev/_old_vers/poll.le.3.14 в сопровождающем книгу файловом архиве) устройства (и тестовое окружение к нему), которое функционирует следующим образом:

- устройство допускает неблокирующие операции *записи* (в буфер) в любом количестве, последовательности и в любое время. Операция записи обновляет содержимое буфера устройства и устанавливает указатель чтения в начало нового содержимого;
- операция *чтения* может запрашивать любое число байтов в последовательных операциях (от 1 до 32 767). 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, мс
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, а принципиально новые моменты относятся к реализации операций `poll()` и блокирования:

- операция `poll()` вызывает (всегда) `poll_wait()` для одной (в нашем случае это `qwait`) или нескольких очередей ожидания (часто это одна очередь для чтения и одна для записи);
- далее производится анализ доступности условий для выполнения операций записи и чтения и на основе этого анализа 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`, переданный параметром `struct file`;
- если же затребована блокирующая операция чтения, а данные для ее выполнения недоступны, вызывающий процесс блокируется;
- разблокирован читающий процесс будет при выполнении более поздней операции записи (в условиях теста — с другого терминала).

Теперь что касается процессов пространства пользователя. Вот обещанный общий включаемый файл:

user.h

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

struct parm {
   int blk, vis, mlt;
};
struct parm parms(int argc, char *argv[], int par) {
   int c;
   struct parm p = { 0, 0, 0 };
   while((c = getopt(argc, argv, "bvm")) != EOF)
      switch(c) {
         case 'b': p.blk = 1; break;
         case 'm': p.mlt = 1; break;
```

```
         case 'v': p.vis++; break;
         default: goto err;
   }
   // par > 0 - pecho; par < 0 - pcat
   if((par != 0 && (argc - optind) != abs(par))) goto err;
   if(par < 0 && atoi(argv[ optind ]) <= 0) goto err;
   return p;
err:
   ERR("usage: %s [-b][-m][-v] %s\n", argv[ 0 ], par < 0 ?
        "<read block size>" : "<write string>");
}

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

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

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

Тест записи:

pecho.c

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

```
   int dfd = opendev();                        // дескриптор устройства
   if(0 == p.blk) nonblock(dfd);
   struct pollfd client[ 1 ] = {
      { .fd = dfd,
        .events = POLLOUT | POLLWRNORM,
      }
   };
   struct timeval t1, t2;
   gettimeofday(&t1, NULL);
   int res;
   if(0 == p.mlt) res = poll(client, 1, -1);
   res = write(dfd, sout, strlen(sout)); // запись
   gettimeofday(&t2, NULL);
   fprintf(stdout, "interval %s write %d bytes: ", interval(t1, t2), res);
   if(res < 0) ERR("write error: %m\n");
   else if(0 == res) {
      if(errno == EAGAIN)
         fprintf(stdout, "device NOT READY!\n");
   }
   else fprintf(stdout, "%s\n", sout);
   close(dfd);
   return EXIT_SUCCESS;
};
```

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

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

где:

- `-b` — установить блокирующий режим операции (по умолчанию неблокирующий);
- `-m` — не использовать ожидание на `poll()` (по умолчанию используется);
- `-v` — увеличить степень детализации вывода (для отладки).

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

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

pcat.c

```
#include "poll.h"
int main(int argc, char *argv[]) {
   struct parm p = parms(argc, argv, -1);
   int blk = LEN_MSG;
   if(optind < argc && atoi(argv[ optind ]) > 0)
      blk = atoi(argv[ optind ]);
   if(p.vis > 0)
      fprintf(stdout, "nonblocked: %s, multiplexed: %s, read block size: %s bytes\n",
         (0 == p.blk ? "yes" : "no"),
         (0 == p.mlt ? "yes" : "no"),
         argv[ optind ]);
   int dfd = opendev();                  // дескриптор устройства
   if(0 == p.blk) nonblock(dfd);
   struct pollfd client[ 1 ] = {
      { .fd = dfd,
        .events = POLLIN | POLLRDNORM,
      }
   };
   while(1) {
      char buf[ LEN_MSG + 2 ];           // буфер данных
      struct timeval t1, t2;
      int res;
      gettimeofday(&t1, NULL);
      if(0 == p.mlt) res = poll(client, 1, -1);
      res = read(dfd, buf, blk);       // чтение
      gettimeofday(&t2, NULL);
      fprintf(stdout, "interval %s read %d bytes: ", interval(t1, t2), res);
      fflush(stdout);
      if(res < 0) {
         if(errno == EAGAIN) {
            fprintf(stdout, "device NOT READY\n");
            if(p.mlt != 0) sleep(3);
         }
         else
            ERR("read error: %m\n");
      }
      else if(0 == res) {
         fprintf(stdout, "read EOF\n");
         break;
      }
      else {
         buf[ res ] = '\0';
         fprintf(stdout, "%s\n", buf);
      }
   }
```

```
    close(dfd);
    return EXIT_SUCCESS;
};
```

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

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

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

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

И окончательно наблюдаем, как это все работает...

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

Запись производим сколько угодно раз последовательно:

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

А вот чтение можем произвести только один раз:

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

При повторной операции чтения:

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

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

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

И, как легко можно видеть, заблокированная операция `cat` после разблокирования выводит уже обновленное 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 циклического чтения 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 с) в другом терминале:

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

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

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

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

```
$ ./pcat -v 3 -m
nonblocked: yes, multiplexed: no, read block size: 3 bytes
interval 00:000 read -1 bytes: device NOT READY
interval 00:000 read -1 bytes: device NOT READY
interval 00:000 read -1 bytes: device NOT READY
interval 00:000 read -1 bytes: device NOT READY
interval 00:000 read 3 bytes: 123
interval 00:000 read 3 bytes: 45
interval 00:000 read 3 bytes: 678
interval 00:000 read 3 bytes: 90
interval 00:000 read 0 bytes: read EOF
```

Опять же, делающая доступными данные операция с другого терминала:

```
$ ./pecho '12345 67890'
interval 00:099 write 11 bytes: 12345 67890
```

Здесь нулевые задержки на неблокирующих операциях указывают на время между операциями. Они разрежены по времени, чтобы не засорять код, и его работу можно было бы наблюдать во *время выполнения самой операции* `read()`, что соответствует действительности.

Блочные устройства

Прежде всего — некоторые терминологические детали... Блочные устройства в UNIX (и в Linux, в частности) — это устройства хранения, устройства с *произвольным доступом*. Для дальнейшего использования над такими устройствами *надстраиваются* файловые системы. Само непосредственное использование блочных устройств (в «сыром» виде) случается, но крайне редко — например, когда их *копированием* мы создаем *дубликаты* дисковых устройств вместе с их содержимым.

Принципиальное отличие этих устройств от рассмотренных ранее символьных устройств, известных как устройства с *последовательным доступом*, состоит в том, что символьные устройства позволяют считывать данные только последовательно байт за байтом, и нет возможности вернуться к ранее считанным байтам. В противовес им устройства с произвольным доступом позволяют в произвольном порядке считывать любые байты в пределах всего диапазона, называемого *емкостью устройства*.

Необходимость в разработке драйвера блочных устройств возникает куда реже, чем для символьных устройств. Более того, в последние годы получило очень широкое распространение простое в использовании решение — файловая система FUSE (проект файловых систем в пространстве пользователя, который будет рассмотрен позже). В связи с изложенным сопровождение примеров кода этого раздела (про блочные устройства) велось только до версии ядра 3.13. Тем не менее весь этот раздел проясняет построение и использование блочных устройств идеологически, что не меняется от версии к версии, и сохранен в книге для полноты понимания вопроса.

Все блочные устройства одной природы поддерживаются единым модулем ядра и в /dev будут представляться именами с одинаковыми префиксами: например, все дисковые устройства на интерфейсах IDE и EIDE — именами `hda`, `hdb`, `hdc`, ... (с единым *префиксом* имени). Точно так же все дисковые устройства, представленные в одной *логической* модели (независимо от их аппаратной реализации), поддерживаются единым драйвером — например, все диски на интерфейсе SATA, флеш-диски на интерфейсе USB и внешние дисковые накопители на интерфейсе USB будут представлены в одной модели SCSI-интерфейса и именоваться в /dev как `sda`, `sdb`, ... А близкие флеш-накопители, выполненные в различных конструктивах SD-карт, будут поддерживаться другим модулем блочного устройства и иметь префиксы имени `mmcblk`. Кроме того, над структурой физического блочного устройства

(диска) может быть *надстроена* логическая структура — разбиение на разделы (partition), каждый из которых будет рассматриваться системой как отдельное блочное устройство. Каждый раздел получит свой индивидуальный *суффикс*, и уже именно на нем (а не на базовом блочном устройстве) будет создаваться файловая система. Например:

```
$ ls -l /dev/sd*
brw-rw---- 1 root disk 8,  0 апр 26 14:06 /dev/sda
brw-rw---- 1 root disk 8,  1 апр 26 14:06 /dev/sda1
brw-rw---- 1 root disk 8,  2 апр 26 14:06 /dev/sda2
brw-rw---- 1 root disk 8,  3 апр 26 14:06 /dev/sda3
brw-rw---- 1 root disk 8, 16 апр 26 14:51 /dev/sdb
brw-rw---- 1 root disk 8, 17 апр 26 14:51 /dev/sdb1
brw-rw---- 1 root disk 8, 32 апр 26 14:51 /dev/sdc
brw-rw---- 1 root disk 8, 33 апр 26 14:51 /dev/sdc1
brw-rw---- 1 root disk 8, 34 апр 26 14:51 /dev/sdc2
```

Здесь присутствуют три достаточно разнородных блочных устройства: `sda` — встроенный HDD-накопитель ноутбука на SATA, `sdb` — съемный флеш-диск на USB и `sdc` — внешний HDD-накопитель Transcend на USB, причем на каждом из *физических* устройств созданы *логические* разделы. А в следующем примере показано представление SD-карты:

```
$ ls -l /dev/mmc*
brw-rw---- 1 root disk 179, 0 апр 26 14:06 /dev/mmcblk0
brw-rw---- 1 root disk 179, 1 апр 26 14:06 /dev/mmcblk0p1
brw-rw---- 1 root disk 179, 2 апр 26 14:06 /dev/mmcblk0p2
```

Причем интересно, что:

- префикс имени устройства (определяемый модулем ядра) вовсе не обязательно двухсимвольный (как чаще всего бывает) — для SD-карты это `mmcblk`;
- суффиксы *порядка* представления устройств не всегда литерно-алфавитные (`hda`, `hdb`, ...) — для SD-карт это число;
- суффиксы разделов (partition) в разбиении устройства не всегда числовые (`hda1`, `hda2`, ...) — для SD-карт это `p1`, `p2`, ...

Из показанного можно предположить, что все префиксы и суффиксы именования блочных устройств определяются кодом поддерживающего модуля, и очень скоро мы убедимся, что это именно так.

Блочное устройство (как и явствует из его названия) обеспечивает обмен блоками данных. *Блок* — единица данных фиксированного размера, которыми осуществляется обмен в системе. Размер блока данных определяется ядром (версией, аппаратной платформой) — чаще всего размер блока совпадает с размером страницы аппаратной архитектуры и для 32-битной архитектуры *x*86 составляет 4096 байтов. Оборудование же хранит данные на физических носителях, разбитых в единицах, называемых *сектор*. Исторически сложилось так, что несколько десятилетий аппаратное обеспечение создавалось для работы с размером сектора 512 байтов. В по-

следние годы существует тенденция в новых устройствах оперировать с бóльшими секторами (4096 байтов).

Вообще-то любое блочное устройство в UNIX (и в Linux, в частности) представляется как последовательность байтов (raw, сырое представление) — например: /dev/sda. Именно такое представление и должен обеспечить модуль-драйвер блочного устройства. Именно поэтому мы можем легко, как с символьного устройства, считать загрузочный сектор диска в файл:

```
$ sudo dd if=/dev/sda of=MBR bs=512 count=1
1+0 записей считано
1+0 записей написано
 скопировано 512 байт (512 B), 0,0295399 c, 17,3 kB/c
$ hexdump -C MBR
00000000  eb 63 90 33 2e 30 fa fc  be 00 7c bf 00 06 8c c8  |.c.3.0....|.....|
00000010  8e d0 89 f4 8e c0 8e d8  51 b9 00 01 f3 a5 59 e9  |........Q.....Y.|
00000020  00 8a fb b4 02 cd 16 24  03 3c 03 75 05 c6 06 61  |.......$.<.u...a|
00000030  07 01 bb be 07 b9 04 00  80 3f 80 74 0e 83 03 02  |.........?.t....|
00000040  ff 00 00 20 01 00 00 00  00 02 fa 90 90 f6 c2 80  |... ............|
00000050  75 02 b2 80 ea 59 7c 00  00 31 00 80 01 00 00 00  |u....Y|..1......|
00000060  00 00 00 00 ff fa 90 90  f6 c2 80 74 05 f6 c2 70  |...........t...p|
00000070  74 02 b2 80 ea 79 7c 00  00 31 c0 8e d8 8e d0 bc  |t....y|..1......|
00000080  00 20 fb a0 64 7c 3c ff  74 02 88 c2 52 be 80 7d  |. ..d|<.t...R..}|
00000090  e8 17 01 be 05 7c b4 41  bb aa 55 cd 13 5a 52 72  |.....|.A..U..ZRr|
000000a0  3d 81 fb 55 aa 75 37 83  e1 01 74 32 31 c0 89 44  |=..U.u7...t21..D|
000000b0  04 40 88 44 ff 89 44 02  c7 04 10 00 66 8b 1e 5c  |.@.D..D.....f..\|
000000c0  7c 66 89 5c 08 66 8b 1e  60 7c 66 89 5c 0c c7 44  ||f.\.f..`|f.\..D|
000000d0  06 00 70 b4 42 cd 13 72  05 bb 00 70 eb 76 b4 08  |..p.B..r...p.v..|
000000e0  cd 13 73 0d f6 c2 80 0f  84 d8 00 be 8b 7d e9 82  |..s..........}..|
000000f0  00 66 0f b6 c6 88 64 ff  40 66 89 44 04 0f b6 d1  |.f....d.@f.D....|
00000100  c1 e2 02 88 e8 88 f4 40  89 44 08 0f b6 c2 c0 e8  |.......@.D......|
00000110  02 66 89 04 66 a1 60 7c  66 09 c0 75 4e 66 a1 5c  |.f..f.`|f..uNf.\|
00000120  7c 66 31 d2 66 f7 34 88  d1 31 d2 66 f7 74 04 3b  ||f1.f.4..1.f.t.;|
00000130  44 08 7d 37 fe c1 88 c5  30 c0 c1 e8 02 08 c1 88  |D.}7....0.......|
00000140  d0 5a 88 c6 bb 00 70 8e  c3 31 db b8 01 02 cd 13  |.Z....p..1......|
00000150  72 1e 8c c3 60 1e b9 00  01 8e db 31 f6 bf 00 80  |r...`......1....|
00000160  8e c6 fc f3 a5 1f 61 ff  26 5a 7c be 86 7d eb 03  |......a.&Z|..}..|
00000170  be 95 7d e8 34 00 be 9a  7d e8 2e 00 cd 18 eb fe  |..}.4...}.......|
00000180  47 52 55 42 20 00 47 65  6f 6d 00 48 61 72 64 20  |GRUB .Geom.Hard |
00000190  44 69 73 6b 00 52 65 61  64 00 20 45 72 72 6f 72  |Disk.Read. Error|
000001a0  0d 0a 00 bb 01 00 b4 0e  cd 10 ac 3c 00 75 f4 c3  |...........<.u..|
000001b0  00 00 00 00 00 00 00 00  61 d9 61 d9 00 00 00 20  |........a.a.... |
000001c0  21 00 83 fe ff ff 00 08  00 00 00 00 5f 05 00 fe  |!..........._...|
000001d0  ff ff 82 fe ff ff 00 08  5f 05 00 c0 5d 00 80 fe  |........_...]...|
000001e0  ff ff 83 fe ff ff 00 c8  bc 05 00 00 40 01 00 00  |............@...|
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
00000200
```

Или полностью переносить (структуру и содержимое) одного физического носителя на другой простым копированием блочного устройства как байтового потока:

```
$ sudo cp /dev/sda /dev/sdf
...
```

Позже на последовательность байтов, созданную драйвером блочного устройства, может быть *наложена* структура разделов (partition) в формате MBR (Master Boot Record) — программой fdisk или в новом, идущем ему на смену формате GPT (GUID Partition Table) — программами parted, gparted, gdisk). Далее на сам диск или любой его раздел может быть *наложена* структура любой из *многих* файловых систем, поддерживаемых Linux, что делается программами:

```
$ ls -w80 /sbin/mkfs*
/sbin/mkfs          /sbin/mkfs.ext3     /sbin/mkfs.gfs2     /sbin/mkfs.ntfs
/sbin/mkfs.btrfs    /sbin/mkfs.ext4     /sbin/mkfs.hfsplus  /sbin/mkfs.reiserfs
/sbin/mkfs.cramfs   /sbin/mkfs.ext4dev  /sbin/mkfs.minix    /sbin/mkfs.vfat
/sbin/mkfs.ext2     /sbin/mkfs.fat      /sbin/mkfs.msdos    /sbin/mkfs.xfs
```

Здесь обратим внимание на то, что все многочисленные приведенные в таком выводе программы (`*disk`, `mkfs*` и др.) являются процессами *пользовательского* адресного пространства и никакого отношения к ядру (и модулям ядра) уже не имеют. Задача модуля блочного устройства заключается только в создании сырого устройства /dev/*xxx*, позволяющего выполнять на нем блочные операции. В этом состоит еще одно отличие блочных устройств: после загрузки модуля поддержки устройства нужно еще произвести достаточно много манипуляций пользовательскими программами, чтобы выполнить подготовку файловой структуры устройства и сделать его пригодным для использования.

Поскольку работа блочного устройства в значительной мере завязана на его последующее структурирование уже в пользовательском пространстве, а отдельные характеристики для возможностей такого структурирования вообще ограничиваются кодом модуля, то рассмотрение техники написания модулей блочных устройств будет перемежаться со связанными вопросами структурирования и использования таких блочных устройств.

В принципе, модуль блочного устройства мог бы во многом наследовать технику символьных устройств, детально рассмотренную на примерах ранее, и для некоторых операций это так и происходит. Но сами операции, осуществляющие поддержку обмена данными (ввода/вывода) строятся по-другому. Так что начнем мы с рассмотрения именно аналогий между символьными и блочными устройствами, и только затем перейдем к специфике.

Особенности драйвера блочного устройства

Из различий использования и происходят различия между модулями блочных устройств и рассматриваемыми ранее символьными. Основные факторы, порождающие эти различия, следующие:

- для блочных устройств первостепенным вопросом является вопрос производительности (скорости). Для символьных устройств производительность может

быть не основным фактором — многие символьные устройства могут работать ниже своей максимальной скорости, и производительность системы в целом от этого не страдает. Но для блочных вопрос производительности — это вопрос вообще конкурентных преимуществ операционной системы на рынке. Поэтому бóльшая часть разработки модуля блочного уровня сосредоточена на обеспечении производительности;

- запросы на ввод/вывод символьные устройства получают из процессов пространства пользователя, явно выполняющих операции `read()` или `write()`. Блочные устройства могут получать запросы ввода/вывода как из процессов пространства пользователя, так и из кода ядра (модулей ядра), — например, при монтировании дисковых устройств или виртуализации страниц оперативной памяти на диск. Исходя из этого, запросы ввода/вывода должны прежде попадать в блочную подсистему ввода/вывода ядра и только затем передаваться ею непосредственно драйверу, осуществляющему обмен с устройством;
- блочное устройство, как будет рассмотрено вскоре, не выполняет непосредственно запросы `read()` и `write()`, как это делает символьное устройство. Напротив, запросы на обработку драйверу поступают через очередь запросов на обслуживание, которую поддерживает ядро. И ядро формирует вызовом метода `request()` характер, последовательность и объем операций, выполняемых драйвером. Это (обработка очереди ядра) основной и наиболее часто задействуемый механизм работы блочного драйвера, но могут быть и более редкие в использовании варианты, которые тоже будут рассмотрены далее;
- блочное устройство, представленное модулем как имя в /dev, еще далеко не пригодно непосредственно для дальнейшего использования. Над ним предварительно необходимо выполнить целый ряд подготовительных манипуляций пространства пользователя, прежде чем устройство можно будет использовать (создание структуры разделов, *форматирование* файловых систем на разделах).

Обзор примеров реализации

Техника блочных устройств гораздо более громоздкая, чем устройств символьных. Архив кода примеров для блочных устройств представляет несколько совершенно различных реализаций, а поэтому требует кратких комментариев:

```
$ ls -l | grep ^d
drwxrwxr-x 2 Olej Olej  4096 апр 24 13:00 block_mod.ELDD_14
drwxr-xr-x 2 Olej Olej  4096 апр 24 13:00 block_mod.LDD3
drwxr-xr-x 2 Olej Olej  4096 апр 24 13:00 block_mod.LDD_35
drwxr-xr-x 2 Olej Olej  4096 апр 24 13:00 dubfl
```

Первые три приведенных варианта — это реализации RAM-дисков (структуры блочного устройства в памяти). Такая организация памяти наиболее гибкая для отработки и экспериментов, а потом может быть перенесена на реальный физический носитель. Эти три реализации отталкивались в начале развития от примеров, более или менее полно описанных в трех книгах (все детально названы в библиографии), откуда за ними и остались такие имена тестовых каталогов:

- ELDD_14 — [5] Sreekrishnan Venkateswaran. Essential Linux Device Drivers (глава 14), имя этого модуля `block_mod_e.ko`;
- LDD3 — [2] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Linux Device Drivers, имя этого модуля `block_mod_s.ko`;
- LDD_35 — [6] Jerry Cooperstein. Writing Linux Device Drivers, в 2 томах (глава 35), имя этого модуля `block_mod_c.ko`

  Именно под такими именами эти модули и будут показываться в примерах их использования без дальнейших разъяснений. Все три реализации в исходном своем виде (такой вид вложен в подпапки папки blkdev сопровождающего книгу файлового архива) даже не компилируются в ядре, уже начиная с 2.6.32 (из-за большой волатильности API новых ядер), поэтому они радикально трансформированы, а в дальнейшем развивались в нужном мне направлении.
- последний вариант (dubfl) использует в качестве физического носителя предварительно создаваемый файл (командой `dd` или любым другим образом) в файловой системе Linux (конечно, достаточно большого для этих целей размера). Информация на таком «устройстве» хранится перманентно (между выключениями питания компьютера), поэтому такой вариант гораздо удобнее для определенных групп экспериментов.

Регистрация устройства

Регистрация блочных устройств во многом напоминает регистрацию символьных — с коррекцией на присущую им специфику.

Подготовка к регистрации

Начинается регистрация устройства (и соответственно завершается по окончании работы) с вызовов API:

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

Точно так же, как это было для символьных устройств, в качестве `major` в вызове `register_blkdev()` может быть нулевое значение, что позволит системе присвоить `major` первое свободное значение.

В любом случае вызов `register_blkdev()` возвратит текущее значение `major` или сигнализирующее об ошибке отрицательное значение (как это всегда принято в ядре) — обычно это бывает, когда в вызове для `major` задается принудительное значение, уже занятое ранее другим устройством в системе.

В качестве `name` в этот вызов передается родовое имя устройств — например, для SCSI устройств это было бы `sd`, а для блочных устройств, создаваемых в примерах далее (`xda`, `xdb`, ...), — это будет `xd`. Регистрация имени устройства создает соответствующую запись в файле /proc/devices, но не создает еще самого устройства в /dev:

```
$ cat /proc/devices | grep xd
252 xd
```

В ядре 2.6 и старше, как пишут, в принципе регистрацию с помощью `register_blkdev()` можно и не проводить, но обычно она делается, и это больше дань традиции.

Здесь же (сразу перед `register_blkdev()`или после) *для каждого* устройства (привода) драйвера проделывают инициализацию *очереди обслуживания*, связанной с устройством (об этом мы вскоре поговорим, когда перейдем к рассмотрению операций чтения и записи). Делается это кодом, подобным следующему:

```
spinlock_t xda_lock;
struct request_queue* xda_request_queue;
...
spin_lock_init(&xda_lock);
if(!(xda_request_queue = blk_init_queue(&xda_request_func, &xda_lock))) {
   // ошибка и завершение
}
```

Показанный фрагмент:

- создает очередь обслуживания запросов `xda_request_queue` в ядре;
- увязывает ее с примитивом синхронизации `xda_lock`, который будет использоваться для монополизации выполняемых операций с этой очередью;
- определяет для очереди `xda_request_queue` функцию обработки запросов, которая будет вызываться *ядром* при каждом требовании на обработку запроса чтения или записи. Функция должна иметь прототип вида:

  ```
  static void xda_request_func(struct request_queue *q) { ... }
  ```

Забегая вперед, отметим, что обработка запросов обслуживания с помощью очереди ядра является не единственным способом обеспечения операций чтения и записи. Но такой способ используют 95% драйверов. Остающиеся 5% реализуют прямое выполнение запросов по мере их поступления, и такой способ также будет рассмотрен далее.

Но все эти подготовительные операции пока не приблизили нас к созданию отображения блочного устройства в каталог /dev. Для реального создания отображения *каждого диска* в /dev нужно: а) создать для этого устройства структуру `struct gendisk`; б) заполнить эту структуру и в) зарегистрировать ее в ядре. Структура `struct gendisk` (описана в <linux/genhd.h>) является описанием каждого диска в ядре.

> ***Примечание***
>
> Экземпляром точно такой же структуры в ядре будет описываться каждый раздел (partition) диска, создаваемый с помощью fdisk или parted. Но разработчику не нужно беспокоиться об этих экземплярах — их позже нормально создаст ядро при работе все тех же утилит, производящих разбиение диска.

Структуру `struct gendisk` нельзя создать просто так, как мы создаем другие структуры (размещая память `kmalloc()` или подобными вызовами), — она представляет собой динамически создаваемую структуру, которая сильно завязана в ядре и требует специальных манипуляций со стороны ядра для инициализации. Поэтому создаем ее (а в конце работы уничтожаем) вызовами:

```
struct gendisk* alloc_disk(int minors);
void del_gendisk(struct gendisk*);
```

Диски с разметкой MBR и GPT

Параметр `minors` в вызове `alloc_disk()` должен быть числом младших номеров, *резервируемых* для отображения в /dev этого диска, и всех его разделов в будущем.

> ***Примечание***
>
> Не следует рассчитывать, что позже можно будет просто изменить соответствующее поле `minors` в структуре `struct gendisk`, которое и заполняется вызовом `alloc_disk()`, и ожидать, что все будет нормально работать: там инициализация существенно сложнее — похоже, что под размер `minors` выделяется соответствующее число слотов (дочерних структур `struct gendisk`).

Параметр `minors` в вызове `alloc_disk()` заслуживает отдельного рассмотрения. Если вы зададите для `minors` значение 4, то получите возможность с помощью fdisk создавать до четырех *первичных* (primary) разделов в MBR этого диска (/dev/xda1 ... /dev/xda4). Но вы не сможете в этом случае создать *ни единого расширенного* (extended) раздела, потому что первый же созданный логический (logical) раздел внутри расширенного получит номер 5:

```
# fdisk -l /dev/xda
...
Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/xda1               1        4000        2000   83  Linux
/dev/xda2            4001        8191        2095+   5  Расширенный
/dev/xda5            4002        8191        2095   83  Linux
# ls -l /dev/xda*
brw-rw---- 1 root disk 252, 0 нояб. 12 10:44 /dev/xda
brw-rw---- 1 root disk 252, 1 нояб. 12 10:44 /dev/xda1
brw-rw---- 1 root disk 252, 2 нояб. 12 10:44 /dev/xda2
brw-rw---- 1 root disk 252, 5 нояб. 12 10:46 /dev/xda5
```

Обычно в качестве `minors` указывают значение 16. Некоторые драйверы указывают это значение как 64.

> ***Примечание***
>
> В обсуждениях пользователи часто изумляются, почему никакими сторонними средствами разбиения диска не удается получить на диске больше 16 разделов (хотя сам fdisk и не умеет этого делать, но по определению цепочка вложенных extended-разделов может быть безграничной, и сторонние средства разбиения позволяют создавать такие вложенные структуры расширенных разделов). Как мы здесь видим, невозможно *не создать* большее число разделов на физическом диске, а невозможно *отобразить* эти созданные разделы драйвером, поддерживающим этот диск.

Все сказанное относится к способу разбиения диска в стандарте разметки MBR (Master Boot Record), который привычен и использовался как минимум на протяжении последних 35 лет. Но в настоящее время происходит массированный переход к разметке диска в стандарте GPT (GUID Partition Table), который давно уже на-

зрел. Основные отличительные стороны GPT в контексте нашего рассмотрения модулей блочных устройств (GPT в целом — слишком обширная тема) следующие:

- используются только первичные разделы;
- число таких разделов может быть до 128;
- полностью изменены идентификаторы типов разделов, и они теперь 32-битные (например, для Linux файловых систем — 8300 вместо прежних 83).

Утилита fdisk используется для работы с форматом MBR — она не понимает формат диска GPT и будет сообщать :

```
$ sudo fdisk -l /dev/sdc
WARNING: GPT (GUID Partition Table) detected on '/dev/sdc'! The util fdisk doesn't support
GPT. Use GNU Parted.
...
```

Для работы с дисками GPT используются утилиты parted (gparted) или ставшая уже популярной утилита gdisk. Вот как видит parted один из GPT-дисков:

```
$ sudo parted /dev/sdb
GNU Parted 3.0
Используется /dev/sdb
(parted) print
Модель: Ut163 USB2FlashStorage (scsi)
Диск /dev/sdb: 1011MB
Размер сектора (логич./физич.): 512B/512B
Таблица разделов: gpt
Disk Flags:.

Номер Начало  Конец   Размер  Файловая система Имя                  Флаги
1     1049kB  53,5MB  52,4MB  ext2             EFI System           загрузочный, legacy_boot
10    53,5MB   578MB   524MB                   Microsoft basic data
20    578MB   1011MB   433MB                   Linux filesystem
```

Утилитами для работы с дисками GPT можно создать до 128 разделов (или любое число дисков с номерами разделов из диапазона 1...128). Вот как утилитой gdisk созданы 18 разделов на диске и как они ею отображаются:

```
$ sudo gdisk -l /dev/sdf
GPT fdisk (gdisk) version 0.8.4

Partition table scan:
  MBR: protective
  BSD: not present
  APM: not present
  GPT: present

Found valid GPT with protective MBR; using GPT.
Disk /dev/sdf: 7827456 sectors, 3.7 GiB
Logical sector size: 512 bytes
```

```
Disk identifier (GUID): C9533CB0-A119-429D-84D8-2B5C1DEA7E30
Partition table holds up to 128 entries
First usable sector is 34, last usable sector is 7827422
Partitions will be aligned on 2048-sector boundaries
Total free space is 5984189 sectors (2.9 GiB)

Number  Start (sector)    End (sector)    Size   Code  Name
 21           2048          104447    50.0 MiB   8300  Linux filesystem
 22         104448          206847    50.0 MiB   8300  Linux filesystem
 23         206848          309247    50.0 MiB   8300  Linux filesystem
 24         309248          411647    50.0 MiB   8300  Linux filesystem
 25         411648          514047    50.0 MiB   8300  Linux filesystem
 26         514048          616447    50.0 MiB   8300  Linux filesystem
 27         616448          718847    50.0 MiB   8300  Linux filesystem
 28         718848          821247    50.0 MiB   8300  Linux filesystem
 29         821248          923647    50.0 MiB   8300  Linux filesystem
101        1128448         1230847    50.0 MiB   8300  Linux filesystem
102        1230848         1333247    50.0 MiB   8300  Linux filesystem
103        1333248         1435647    50.0 MiB   8300  Linux filesystem
104        1435648         1538047    50.0 MiB   8300  Linux filesystem
105        1538048         1640447    50.0 MiB   8300  Linux filesystem
106        1640448         1742847    50.0 MiB   8300  Linux filesystem
107        1742848         1845247    50.0 MiB   8300  Linux filesystem
108        1845248         1947647    50.0 MiB   8300  Linux filesystem
109        1947648         2050047    50.0 MiB   8300  Linux filesystem
```

И вот как такая структура GPT-диска отображается в Linux хорошо написанным (Fedora 17, ядро 3.5) модулем поддержки:

```
$ ls /dev/sdf*
/dev/sdf     /dev/sdf103  /dev/sdf106  /dev/sdf109  /dev/sdf23  /dev/sdf26  /dev/sdf29
/dev/sdf101  /dev/sdf104  /dev/sdf107  /dev/sdf21   /dev/sdf24  /dev/sdf27
/dev/sdf102  /dev/sdf105  /dev/sdf108  /dev/sdf22   /dev/sdf25  /dev/sdf28
```

Из показанного очевидно, что грядут большие перемены. И это должно быть учтено при создании модулей ядра блочных устройств. В частности, параметр `minors` в вызове `alloc_disk()`, наверное, должен указываться как 128.

Заполнение структуры

Мы остановились на *создании* структуры `struct gendisk`, после чего самое время перейти к *заполнению* ее полей. Это весьма большая структура (описана в <linux/genhd.h>), и мы отметим только те поля, которые нужно заполнить под свой драйвер (есть еще несколько полей, которые требуют «технического» заполнения, их можно видеть в примере далее):

```
struct gendisk {
        int major;                      /* major number of driver */
        int first_minor;
```

```
        int minors;                     /* maximum number of minors, =1 for
                                         * disks that can't be partitioned. */
        char disk_name[DISK_NAME_LEN];  /* name of major driver */
...
        const struct block_device_operations *fops;
...
}
```

Здесь для нас важны:

- `major` — уже встречавшийся нам старший номер для устройств такого *класса* (поддерживаемых этим модулем ядра). Мы его ранее присвоили устройству принудительно или получили в результате вызова `register_blkdev(0, ...)`;
- `minors` — максимальное число *разделов*, обслуживаемое на диске. Это значение обсуждалось ранее — его поле заполнено вызовом `alloc_disk()`, оно главным образом для чтения;
- `first_minor` — номер `minor`, представляющий сам диск в /dev (например, /dev/xda). Последующие разделы диска (в пределах их максимального числа `minors`) получат соответствующие младшие номера — например, для /dev/xda5 будет использован номер `first_minor + 5`:

  ```
  # ls -l /dev/xda*
  brw-rw---- 1 root disk 252, 0 нояб. 12 10:44 /dev/xda
  brw-rw---- 1 root disk 252, 1 нояб. 12 10:44 /dev/xda1
  brw-rw---- 1 root disk 252, 2 нояб. 12 10:44 /dev/xda2
  brw-rw---- 1 root disk 252, 5 нояб. 12 10:46 /dev/xda5
  ```

- `disk_name` — имя диска, с которым он будет отображаться в /dev, родовое имя устройств `xd` (шаблон имени, *префикс* — драйвер может обслуживать более одного привода устройства) мы уже указывали в вызове типа:

  ```
  register_blkdev(major, MY_DEVICE_NAME);
  ```

  Теперь здесь мы можем персонифицировать имя для конкретного привода (если создается несколько устройств `xd`, как в примере) — нечто по типу:

  ```
  snprintf(disk_name, DISK_NAME_LEN - 1, "xd%c",  'a' + i);
  ```

- Это те имена (`xda`, `xdb`, `xdc`, ...), которые появятся в:

  ```
  $ cat /proc/partitions
  major minor  #blocks  name
     8        0   58615704 sda
     8        1   45056000 sda1
     8        2    3072000 sda2
     8        3   10485760 sda3
    11        0    1048575 sr0
   179        0    7774208 mmcblk0
   179        1     522081 mmcblk0p1
   179        2    1322111 mmcblk0p2
  ```

```
252        0       4096 xda
252        1       2000 xda1
252        2          1 xda2
252        5       2095 xda5
252       16       4096 xdb
252       32       4096 xdc
252       48       4096 xdd
```

- `fops` — адрес таблицы операций устройства, во многом аналогичной такой же для символьного устройства, которую мы детально обсудим вскоре.

  Кроме непосредственного заполнения полей структуры `struct gendisk`, примерно в этом месте делается занесение (в ту же структуру) полной емкости устройства, выраженной в **секторах** по 512 байт:

  ```
  inline void set_capacity(struct gendisk*, sector_t size);
  ```

Завершение регистрации

Теперь структура `struct gendisk` выделена и заполнена, но это еще не делает блочное устройство доступным для использования системой. Чтобы завершить все наши подготовительные операции, мы должны вызвать:

```
void add_disk(struct gendisk* gd);
```

Здесь `gd` и есть структура, которую мы так тщательно готовили накануне.

Вызов `add_disk()` достаточно *опасный*: как только происходит вызов, диск становится активным и его методы могут быть вызваны, независимо от нашего участия, асинхронно, в любое время. На самом же деле первые такие вызовы происходят немедленно, даже еще до того, как произойдет возврат из самого вызова `add_disk()` (это можно увидеть в системном журнале, вызываемом командой `dmesg`). Этим ранним вызовам мы обязаны попыткам ядра вычитать начальные сектора диска в поисках таблицы разделов (MBR или GPT — см. ранее). Если к моменту вызова `add_disk()` драйвер еще не полностью или некорректно инициализирован, то с большой вероятностью вызов приведет к ошибке драйвера и через некоторое, обычно короткое, время — к полному краху системы.

На этом регистрация устройства завершена, устройство создано в системе, и мы можем перейти к детальному рассмотрению теперь уже работы такого устройства.

Таблица операций устройства

Ранее мы заполнили в структуре описания диска `struct gendisk` поле адреса `fops` — таблицу операций блочного устройства. Таблица функций `struct block_device_operations` (ищите ее в <linux/blkdev.h>) по смыслу выполняет для блочного устройства ту же роль, что и таблица файловых операций символьного устройства `struct file_operations`, которая детально изучалась ранее (вид таблицы показан из ядра 3.5):

```
struct block_device_operations {
        int (*open) (struct block_device *, fmode_t);
```

```
        int (*release) (struct gendisk *, fmode_t);
        int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*direct_access) (struct block_device *, sector_t,
                              void **, unsigned long *);
        unsigned int (*check_events) (struct gendisk *disk,
                                      unsigned int clearing);
        /* ->media_changed() is DEPRECATED, use ->check_events() instead */
        int (*media_changed) (struct gendisk *);
        void (*unlock_native_capacity) (struct gendisk *);
        int (*revalidate_disk) (struct gendisk *);
        int (*getgeo)(struct block_device *, struct hd_geometry *);
        /* this callback is with swap_lock and sometimes page table lock held */
        void (*swap_slot_free_notify) (struct block_device *, unsigned long);
        struct module *owner;
};
```

Примечание

Еще до версии ядра 3.10 (3.9 и ниже, все 2.6.*x*) прототип метода `release` должен был возвращать значение `int`, но после нее (3.10 и старше) он имеет возвращаемое значение `void` (см. <linux/blkdev.h>). Поэтому мы должны здесь использовать условную компиляцию, что уже неоднократно показывалось.

Здесь все относительно интуитивно понятно, и во многом напоминает то, что мы видели для символьных устройств. Операции `open()` и `release()` — это функции, вызываемые при открытии и закрытии устройства, весьма часто они вызываются *неявно*, и их не приходится описывать. Неявный их вызов означает, что они вызываются *ядром* в нескольких случаях — например, при загрузке драйвера после вызова `add_disk()` ядро пытается прочитать в своих интересах таблицу разделов (partititon) диска (MBR или GPT). Вызовы `open()` и `release()` *всегда* следуют за *монтированием* и *размонтированием* соответственно блочного устройства командами пользователя. Это можно увидеть из кода и результатов испытаний модуля dubfl.ko (см. папку blkdev/_old_vers/3.2/dubfl в сопровождающем книгу файловом архиве):

```
# insmod dubfl.ko file=XXX
$ ls -l /dev/dbf
brw-rw---- 1 root disk 252, 1 май 20 18:57 /dev/dbf
# mount -t vfat /dev/dbf /mnt/xdb
# umount /dev/dbf
$ dmesg | tail -n2
[20961.008908] + open device /dev/dbf
[21129.476313] + close device /dev/dbf
```

В качестве образцов того, как используются некоторые простейшие функции из таблицы файловых операций, могут служить показанные реализации операций `getgeo()` (возвратить геометрию блочного устройства) и `ioctl()` из таблицы `fops` примеров (см. папку blkdev в сопровождающем книгу файловом архиве):

common.c

```
...
static struct block_device_operations mybdrv_fops = {
   .owner = THIS_MODULE,
   .ioctl = my_ioctl,
   .getgeo = my_getgeo
};
...
```

ioctl.c

```
/* #include <linux/hdreg.h>
struct hd_geometry {
   unsigned char heads;
   unsigned char sectors;
   unsigned short cylinders;
   unsigned long start;
}; */

static int my_getgeo(struct block_device *bdev, struct hd_geometry *geo) {
   unsigned long sectors = (diskmb * 1024) * 2;
   DBG(KERN_INFO "getgeo\n");
   geo->heads = 4;
   geo->sectors = 16;
   geo->cylinders = sectors / geo->heads / geo->sectors;
   geo->start = geo->sectors;
   return 0;
};

static int my_ioctl(struct block_device *bdev, fmode_t mode,
                    unsigned int cmd, unsigned long arg) {
   DBG("ioctl cmd=%d\n", cmd);
   switch(cmd) {
      case HDIO_GETGEO: {
         struct hd_geometry geo;
         LOG("ioctk HDIO_GETGEO\n");
         my_getgeo(bdev, &geo);
         if(copy_to_user((void __user *)arg, &geo, sizeof(geo)))
            return -EFAULT;
         return 0;
      }
      default:
         DBG("ioctl unknown command\n");
         return -ENOTTY;
   }
}
```

Примечание

Используемые в этом архиве макросы `ERR()`, `LOG()`, `DBG()` определены в примерах только для компактности кода, не должны смущать и скрывают под собой всего лишь `printk()`:

```
#define ERR(...) printk(KERN_ERR "! "__VA_ARGS__)
#define LOG(...) printk(KERN_INFO "+ "__VA_ARGS__)
#define DBG(...) if(debug > 0) printk(KERN_DEBUG "# "__VA_ARGS__)
```

Современное ядро никак не связано с *геометрией* блочного устройства — устройство рассматривается как линейный массив *секторов* (в наших примерах это массив секторов последовательно размещенных в памяти). Для работы операционной системы с блочным устройством не принципиально разбиение его пространства в терминах цилиндров, головок и числа секторов на дорожку. Но достаточно многие утилиты GNU для работы с дисковыми устройствами (fdisk, hdparm, ...) предполагают получение информации о геометрии диска. Чтобы не создавать неожиданностей при работе с этими утилитами, целесообразно реализовать показанные ранее операции (без них может оказаться, что вы не сможете создать разделы на поддерживаемом вашим модулем устройстве). Посмотрим, что получилось в нашей реализации:

```
# insmod block_mod_s.ko
# hdparm -g /dev/xdd
/dev/xdd:
 geometry      = 128/4/16, sectors = 8192, start = 16
```

Примечание

Убедиться в том, что для работы Linux геометрия диска не имеет значения, можно на реальном флеш-диске, записав с помощью той же утилиты `hdparm` произвольные значения T/S (число секторов в одной дорожке), H (число головок) и C (число цилиндров), но так, чтобы их произведение сохранило корректный общий размер диска, выраженный в секторах. При этом диск может выглядеть как имеющий совершенно разные геометрии, но при этом сохраняющий нормальную работоспособность. Геометрия диска, очевидно, имеет существенное значение для механических устройств, особенно для пересчета линейного номера сектора в соответствующий номер цилиндра для перемещения головки. Это может использоваться в оптимизирующем механизме доступа к очереди запросов в ядре, но разработчик модуля не имеет доступа для влияния на этот механизм.

В таблице операций `struct block_device_operations`, конечно, находит некоторое отражение тот факт, что это операции именно блочного устройства. Например, если вы разрабатываете устройство со сменными носителями, то для проверки несменяемости носителя при каждом новом открытии устройства в `open()` нужно вызвать *функцию API ядра* `check_disk_change(struct block_device*)`:

```
int open(struct block_device* dev, fmode_t mode) {
   ...
   check_disk_change(dev);
   ...
}
```

Но для проверки *фактической* смены носителя функция `check_disk_change()` осуществит обратным порядком вызов метода из вашей же таблицы операций:

```
int (*media_changed)(struct gendisk*);
```

Если носитель сменился, `media_changed()` возвращает ненулевое значение. Если будет возвращен признак смены носителя (ненулевое значение), ядро тут же вызовет еще один метод из таблицы операций:

```
int (*revalidate_disk)(struct gendisk*);
```

Этот реализованный вами метод должен сделать все необходимое, чтобы подготовить драйвер для операций **с** *новым* носителем (если устройство — это RAM-диск, как в примерах, то вызов должен, например, обнулить область памяти устройства). После осуществления вызова `revalidate_disk()` ядро тут же предпримет попытку перечитать *таблицу разделов* устройства. Так обрабатываются сменные носители в блочных устройствах.

> ***Примечание***
>
> Нужно различать устройство со сменным носителем и сменное устройство. Например, устройства со сменным носителем — это DVD/RW, Iomega ZIP и подобные им. А сменные устройства — это жесткие диски системы HotPlug, USB флеш-диски или USB внешние дисковые накопители. Реинициализация драйвера в том и в другом случае происходит совершенно разными способами.

И последние замечания относительно таблицы операций:

- обновления и изменения (временами достаточно радикальные) в `struct block_device_operations` происходят довольно часто при изменении версии ядра. Например, показанный ранее вызов `swap_slot_free_notify()` в таблице (относящейся к версии 3.5) появился только, начиная с ядра 2.6.35 (см. **http://lxr.free-electrons.com/source/include/linux/blkdev.h?v=2.6.34**);
- при изменениях уже существующие и описанные в публикациях методы объявляются устаревшими (`DEPRECATED` — они могут быть изъяты в последующем). Так объявлен (с версии 2.6.38) обсуждавшийся ранее метод `media_changed()`, а вместо него предложен метод `check_events()`, о чем и напоминает комментарий:

```
...
unsigned int (*check_events) (struct gendisk *disk,
                              unsigned int clearing);
/* ->media_changed() is DEPRECATED, use ->check_events() instead */
int (*media_changed) (struct gendisk *);
...
```

Обмен данными

Все, что было до сих пор, должно быть просто и понятно. Но при внимательном рассмотрении изумляет одна вещь: а где же сами операции чтения и записи данных устройства? А вот сами операции чтения и записи в таблице операций блочного устройства и *отсутствуют*! Они выполняются по-другому:

- получив запрос на чтение или запись ядро помещает запрос обслуживания в очередь, связанную с драйвером (которую мы инициализировали ранее);
- запросы в очереди ядро *пересортирует* так, чтобы оптимизировать выполнение последовательности запросов (например, минимизировать перемещения головок жесткого диска, объединить два или более соседних запроса в один более протяженный, ...);
- очередной (текущий крайний) запрос из очереди передается на обработку *функции драйвера*, которую мы определили для обработки запросов;
- направление (ввод или вывод) и параметры запроса (начальные сектора, число секторов и др.) определены в самом *теле* запроса.

Единичный запрос к блочному драйверу (`struct request`) в общем случае представляется набором (*вектором*) из *нескольких* операций (`struct bio`, буфер памяти) одного типа (чтение или запись). Сектора для этих операций могут быть разбросаны по диску (не последовательны) — в зависимости от характера *файловой системы*. Адреса получателей в памяти этих операций тоже могут быть разбросаны (*вектор ввода/вывода*, IOV). Вот обслуживание таких запросов и является задачей драйвера блочного устройства. Такова в целом и основная схема работы драйвера блочного устройства, хотя и очень огрубленно.

Но к этому времени мы рассмотрели все детали, необходимые для того, чтобы рассмотреть и понять код работающего примера модуля блочного устройства (в папке blkdev сопровождающего книгу файлового архива приведено несколько альтернативных примеров, которые рассматриваются для сравнения). Первый обсуждаемый пример (block_mod_s.c) реализует блочное устройство в оперативной памяти (самый наглядный и простой для понимания случай).

Этот модуль реализован в файле block_mod_s.c с некоторыми небольшими включениями, общими для всех приведенных вариантов блочных устройств, которые почти все были обсуждены ранее, и вот последнее из таких включений:

common.h

```
#include <linux/module.h>
#include <linux/vmalloc.h>
#include <linux/blkdev.h>
#include <linux/genhd.h>
#include <linux/errno.h>
#include <linux/hdreg.h>
#include <linux/version.h>

#define KERNEL_SECTOR_SIZE    512

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,35)
#define blk_fs_request(rq)      ((rq)->cmd_type == REQ_TYPE_FS)
#endif
```

```
static int diskmb = 4;
module_param_named(size, diskmb, int, 0); // размер диска в Mb,
                                          // по умолчанию - 4Mb
static int debug = 0;
module_param(debug, int, 0);              // уровень отладочных сообщений

#define ERR(...) printk(KERN_ERR "! "__VA_ARGS__)
#define LOG(...) printk(KERN_INFO "+ "__VA_ARGS__)
#define DBG(...) if(debug > 0) printk(KERN_DEBUG "# "__VA_ARGS__)
```

Здесь, в частности, описан полезный в экспериментах параметр загрузки модуля `diskmb` — это объем создаваемого блочного устройства в мегабайтах.

Код модуля block_mod_s.c предполагает реализацию *трех* альтернативных стратегий реализации обмена данными. Выбор осуществляется при загрузке модуля параметром `mode`. Значение параметра `mode` определяет, какая функция обработки будет использоваться. Поэтому (для облегчения понимания) сначала мы рассмотрим полный код модуля, но исключим из него реализации самих обработчиков обмена, а позже по порядку разберемся, как реализуются обработчики в каждой стратегии:

block_mod_s.c

```
#include "../common.h"

static int major = 0;
module_param(major, int, 0);
static int hardsect_size = KERNEL_SECTOR_SIZE;
module_param(hardsect_size, int, 0);
static int ndevices = 4;
module_param(ndevices, int, 0);
// The different "request modes" we can use:
enum {   RM_SIMPLE  = 0,  // The extra-simple request function
         RM_FULL    = 1,  // The full-blown version
         RM_NOQUEUE = 2,  // Use make_request
     };
static int mode = RM_SIMPLE;
module_param(mode, int, 0);

static int nsectors;

struct disk_dev {               // The internal representation of our device.
   int size;                    // Device size in sectors
   u8 *data;                    // The data array
   spinlock_t lock;             // For mutual exclusion */
   struct request_queue *queue; // The device request queue */
   struct gendisk *gd;          // The gendisk structure */
};
```

```
static struct disk_dev *Devices = NULL;
...

static void simple_request(struct request_queue *q) {   // Простой запрос
                                                        // с обработкой очереди
...
}

static void full_request(struct request_queue *q) {  // Запрос с обработкой
                                                     // вектора BIO
...
}

static void make_request(struct request_queue *q,
                         struct bio *bio) { // Прямое выполнение запроса
                                            // без очереди
...
}

#include "../ioctl.c"
#include "../common.c"

#define MY_DEVICE_NAME "xd"
#define DEV_MINORS       16

static void setup_device(struct disk_dev *dev, int which) { // Set up our internal device.
   memset(dev, 0, sizeof(struct disk_dev));
   dev->size = diskmb * 1024 * 1024;
   dev->data = vmalloc(dev->size);
   if(dev->data == NULL) {
      ERR("vmalloc failure.\n");
      return;
   }
   spin_lock_init(&dev->lock);
   switch(mode) { // The I/O queue mode
      case RM_NOQUEUE:
         dev->queue = blk_alloc_queue(GFP_KERNEL);
         if(dev->queue == NULL) goto out_vfree;
         blk_queue_make_request(dev->queue, make_request);
         break;
      case RM_FULL:
         dev->queue = blk_init_queue(full_request, &dev->lock);
         if(dev->queue == NULL) goto out_vfree;
         break;
      default:
         LOG("bad request mode %d, using simple\n", mode);
         /* fall into.. */
```

```
      case RM_SIMPLE:
         dev->queue = blk_init_queue(simple_request, &dev->lock);
         if(dev->queue == NULL) goto out_vfree;
         break;
   }
   blk_queue_logical_block_size(dev->queue, hardsect_size);  // Set the hardware sector size
   dev->queue->queuedata = dev;
   dev->gd = alloc_disk(DEV_MINORS);      // Число разделов при разбиении
   if(! dev->gd) {
      ERR("alloc_disk failure\n");
      goto out_vfree;
   }
   dev->gd->major = major;
   dev->gd->minors = DEV_MINORS;
   dev->gd->first_minor = which * DEV_MINORS;
   dev->gd->fops = &mybdrv_fops;
   dev->gd->queue = dev->queue;
   dev->gd->private_data = dev;
   snprintf(dev->gd->disk_name, 32, MY_DEVICE_NAME"%c", which + 'a');
   set_capacity(dev->gd, nsectors * (hardsect_size / KERNEL_SECTOR_SIZE));
   add_disk(dev->gd);
   return;
out_vfree:
   if(dev->data) vfree(dev->data);
}

static int __init blk_init(void) {
   int i;
   nsectors = diskmb * 1024 * 1024 / hardsect_size;
   major = register_blkdev(major, MY_DEVICE_NAME);
   if(major <= 0) {
      ERR("unable to get major number\n");
      return -EBUSY;
   }
   // Allocate the device array
   Devices = kmalloc(ndevices * sizeof(struct disk_dev), GFP_KERNEL);
   if(Devices == NULL) goto out_unregister;
   for(i = 0; i < ndevices; i++) // Initialize each device
      setup_device(Devices + i, i);
   return 0;
out_unregister:
   unregister_blkdev(major, MY_DEVICE_NAME);
   return -ENOMEM;
}

static void blk_exit(void) {
   int i;
```

```
   for(i = 0; i < ndevices; i++) {
      struct disk_dev *dev = Devices + i;
      if(dev->gd) {
         del_gendisk(dev->gd);
         put_disk(dev->gd);
      }
      if(dev->queue) {
         if(mode == RM_NOQUEUE)
            blk_put_queue(dev->queue);
         else
            blk_cleanup_queue(dev->queue);
      }
      if(dev->data) vfree(dev->data);
   }
   unregister_blkdev(major, MY_DEVICE_NAME);
   kfree(Devices);
}

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

MODULE_LICENSE("GPL v2");
MODULE_VERSION("1.5");
```

Здесь собрано воедино все то, что мы разрозненно обсуждали ранее. Нам осталось рассмотреть стратегии обработки и то, как это реализуется.

> ***Примечание***
>
> Для простоты начать рассмотрение можно с примеров модулей block_mod_c.c и block_mod_e.c, которые также реализуют равноценные блочные RAM-диски, но делают это только классическим, наиболее часто используемым способом — обслуживанием очереди, управляемой ядром.

Классика: очередь и обслуживание ядром

Первый способ (mode=0), как уже упоминалось ранее, — это почти классика написания модулей блочных устройств (90–95% драйверов, особенно механических дисковых устройств, используют эту стратегию). Обработка осуществляется двумя функциями:

block_mod_s.c

```
...
static int transfer(struct disk_dev *dev, unsigned long sector,
                    unsigned long nsect, char *buffer, int write) { // Собственно обмен
   unsigned long offset = sector * KERNEL_SECTOR_SIZE;
   unsigned long nbytes = nsect * KERNEL_SECTOR_SIZE;
```

```
   if((offset + nbytes) > dev->size) {
      ERR("beyond-end write (%ld %ld)\n", offset, nbytes);
      return -EIO;
   }
   if(write)
      memcpy(dev->data + offset, buffer, nbytes);
   else
      memcpy(buffer, dev->data + offset, nbytes);
   return 0;
}

static void simple_request(struct request_queue *q) {   // Простой запрос
                                                        // с обработкой очереди
   struct request *req;
   unsigned nr_sectors, sector;
   DBG("entering simple request routine\n");
   req = blk_fetch_request(q);
   while(req) {
      int ret = 0;
      struct disk_dev *dev = req->rq_disk->private_data;
      if(!blk_fs_request(req)) {          // невалидный запрос
         ERR("skip non-fs request\n");
         __blk_end_request_all(req, -EIO);
         req = blk_fetch_request(q);
         continue;
      }
      nr_sectors = blk_rq_cur_sectors(req);  // валидный запрос - обработка
      sector = blk_rq_pos(req);
      ret = transfer(dev, sector, nr_sectors, req->buffer, rq_data_dir(req));
      if(!__blk_end_request_cur(req, ret))
         req = blk_fetch_request(q);
   }
}
...
```

Еще не обработанные (или не завершенные ранее) запросы выбираются из очереди ядра (`request_queue`) вызовом:

```
struct request* req blk_fetch_request(struct request_queue*);
```

Запрос может требовать выполнения специальных (управляющих) операций, не являющихся чтением или записью и переданных «не тому» устройству — такие запросы отфильтровываются вызовом

```
bool blk_fs_request(struct request*);
```

Если же запрос признан валидным, то из него извлекаются начальный сектор для обмена (`blk_rq_pos(struct request*)`), число секторов, подлежащих обмену (`blk_rq_cur_sectors(struct request*)`), буфер (`(struct request*)->buffer` — в пространстве ядра!) и

направление требуемого обмена (`rq_data_dir(struct request*)` — запись или чтение). Далее управление передается функции драйвера `transfer()`. Эта наша функция, проверив предварительно, что данные запрошены не за пределами протяженности пространства устройства (это было бы грубой ошибкой), осуществляет копирование данных в буфер ядра или из него. Все дальнейшее «разбрасывание» полученных данных получателю запроса *делает ядро*.

Независимо от того, успешно или по ошибке завершился очередной запрос, ядру нужно сообщить о его завершении и переходе к следующему запросу из очереди. Уведомление ядра выполняют вызовы:

```
__blk_end_request_all(struct request*, int errno);
__blk_end_request_cur(struct request*, int errno);
```

Здесь в `errno` сообщается результат операции: 0 — в случае успеха, отрицательный код (как принято везде в ядре) — в случае ошибки. После этих вызовов текущий запрос считается обработанным и удаляется из очереди.

> ***Примечание***
>
> В API блочных операций практически всем вызовам «с подчеркиванием» существуют парные вызовы «без подчеркивания» — например: `__blk_end_request_all()` соответствует `blk_end_request_all()`. Разница в том, что первые (которые с __) сами захватывают блокировку монопольного использования очереди (помните, мы инициализировали ее в самом начале вместе с очередью?). А для вторых это должен обеспечить программист функции — они должны выполняться с уже захваченной блокировкой, но это предоставляет разработчику дополнительную гибкость. Однако если вы механически замените в коде форму «с подчеркиванием» на «без подчеркивания» без соответствующих изменений в коде — то это прямой путь тут же «подвесить» операционную систему в целом.

Вот, собственно, и вся «классическая» стратегия. Она описана специально очень поверхностно, потому что детальный разбор того, что происходит с очередью и запросом, — это очень увлекательное исследование, но оно не нужно для того, чтобы писать драйвер: показанный код (с минимальными вариациями) является *шаблоном* для написания обработчиков.

Очередь и обработка запроса в драйвере

Следующий вариант (`mode=1`) оказывается сложнее по смыслу происходящего, но столь же прост в реализации.

Каждая структура `struct request` (единичный *блочный* запрос в очереди) представляет собой один запрос ввода/вывода, хотя этот запрос может быть образован как результат слияния нескольких самостоятельных запросов, проделанного планирующими механизмами ядра (оптимизацией запросов). Секторы, являющиеся адресатом любого конкретного запроса, могут быть распределены по всей оперативной памяти получателя (представлять вектор ввода/вывода), хотя они всегда соответствуют на блочном устройстве набору *последовательных* секторов. Запрос (`struct request`) представлен в виде *вектора* сегментов, каждый из которых соответствует одному буферу в памяти (`struct bio`). Ядро может объединить в один запрос несколько элементов вектора, связанных со смежными секторами на диске. Структура

`struct bio` является низкоуровневым описанием элемента запроса блочного ввода/вывода (`struct request`) — такого уровня детализации нам уже достаточно, чтобы продолжить писать драйвер.

В этой стратегии функциями самого драйвера будет производится последовательная обработка всей последовательности `struct bio`, переданной в составе единичного `struct request`:

block_mod_s.c

```
...
static int xfer_bio(struct disk_dev *dev, struct bio *bio) {   // Передача одиночного BIO
   int i, ret;
   struct bio_vec *bvec;
   sector_t sector = bio->bi_sector;
   DBG("entering xfer_bio routine\n");
   bio_for_each_segment(bvec, bio, i) { // Работаем с каждым сегментом независимо
      char *buffer;
      sector_t nr_sectors;
#if LINUX_VERSION_CODE < KERNEL_VERSION(3,11,0)
      buffer = __bio_kmap_atomic(bio, i, KM_USER0);
#else
      buffer = __bio_kmap_atomic(bio, i);
#endif
      nr_sectors = bio_sectors(bio);
      ret = transfer(dev, sector, nr_sectors, buffer, bio_data_dir(bio) == WRITE);
      if(ret != 0) return ret;
      sector += nr_sectors;
#if LINUX_VERSION_CODE < KERNEL_VERSION(3,11,0)
      __bio_kunmap_atomic(bio, KM_USER0);
#else
      __bio_kunmap_atomic(bio);
#endif
   }
   return 0;
}

static int xfer_request(struct disk_dev *dev, struct request *req) {
// Передача всего запроса.
   struct bio *bio;
   int nsect = 0;
   DBG("entering xfer_request routine\n");
   __rq_for_each_bio(bio, req) {
      xfer_bio(dev, bio);
      nsect += bio->bi_size / KERNEL_SECTOR_SIZE;
   }
   return nsect;
}
```

```
static void full_request(struct request_queue *q) { // запрос с обработкой вектора BIO
   struct request *req;
   int sectors_xferred;
   DBG("entering full request routine\n");
   req = blk_fetch_request(q);
   while(req) {
      struct disk_dev *dev = req->rq_disk->private_data;
      if(!blk_fs_request(req)) {                  // невалидный запрос
         ERR("skip non-fs request\n");
         __blk_end_request_all(req, -EIO);
         req = blk_fetch_request(q);
         continue;
      }
      sectors_xferred = xfer_request(dev, req);     // валидный запрос - обработка
      if(!__blk_end_request_cur(req, 0))
         req = blk_fetch_request(q);
   }
}
```

Код обработчика `full_request()` чрезвычайно похож на предыдущий случай `simple_request()` (а как могло быть иначе? — их даже можно было бы объединить в единый код), только вместо передачи `transfer()` вызывается функция `xfer_request()`, которая, последовательно пробегая для каждой `struct bio` в запросе, вызывает для каждого элемента вектора `xfer_bio()`. Эта функция извлекает параметры и осуществляет обмен данными для единичного сегмента, используя уже знакомую нам функцию `transfer()`.

Отказ от очереди

Очереди запросов в ядре реализуют интерфейс подключения модулей, которые представляют собой различные планировщики ввода/вывода (elevator, транспортер). Работа планировщика ввода/вывода заключается в переупорядочении очереди с тем, чтобы предоставлять запросы ввода/вывода драйверу в такой последовательности, которая максимизировала бы производительность. Планировщик ввода/вывода также отвечает за объединение прилегающих запросов: когда планировщик ввода/вывода получает новый запрос, он ищет в очереди запросы, относящиеся к прилегающим секторам, и если таковые нашлись, он объединяет эти запросы (если результирующий запрос не будет нарушать некоторые условия — например, не будет слишком большим). В некоторых случаях, когда производительность не зависит от расположения секторов и последовательности обращений (RAM-диски, флеш-память, ...) можно отказаться от планировщика, предоставив для очереди свой планировщик, который будет немедленно получать запрос по его поступлении.

Это третья (`mode=2`) из рассматриваемых нами стратегий (отказ от очереди). Здесь мы совершенно по другому создаем и инициализируем очередь в ядре для устройства:

```
...
   case RM_NOQUEUE:
         dev->queue = blk_alloc_queue(GFP_KERNEL);
         if(dev->queue == NULL) goto out_vfree;
         blk_queue_make_request(dev->queue, make_request);
         break;
...
```

Мы устанавливаем для такой очереди свой обработчик всех единичных сегментов (`struct bio`), требующих обслуживания (без оптимизации, слияний и т. п.):

```
#if LINUX_VERSION_CODE <= KERNEL_VERSION(3,1,0)
static int make_request(struct request_queue *q, struct bio *bio) {
#else
static void make_request(struct request_queue *q, struct bio *bio) {
#endif
   struct disk_dev *dev = q->queuedata; // прямое выполнение запроса без очереди
   int status = xfer_bio(dev, bio);
   bio_endio(bio, status);
#if LINUX_VERSION_CODE <= KERNEL_VERSION(3,1,0)
   return 0;
#endif
}
```

Функцию `xfer_bio()` мы уже видели ранее, в предыдущем варианте.

Вот этим и исчерпываются наши варианты организации обработки потока данных в модуле блочного устройства.

Пример перманентных данных

Еще одним примером, представленным в архиве, является модуль блочного устройства dubfl.c — здесь носитель данных перенесен из оперативной памяти на реальный дисковый файл (предварительно созданный). То есть наш RAM-диск превращается в перманентный, сохраняющий свое содержимое между последовательными включениями и выключениями питания компьютера. Это позволяет производительнее экспериментировать с кодом модуля. Кроме того, код модуль демонстрирует некоторые любопытные детали работы с файловыми системами из модуля ядра... но это выходит за рамки нашего рассмотрения.

Важная отличительная особенность этой реализации заключается в том, что она позволяет продемонстрировать и изучить технику *сменных носителей*: данные из внешнего файла загружаются при монтировании (открытии) устройства и сохраняются в файл с произведенными в сеансе изменениями, когда устройство закрывается.

Некоторые важные API

Неприятностью является то, что API блочных устройств достаточно быстро изменяется — то, что было работоспособно еще в ядре, скажем, 2.6.29, — уже даже не компилируется в ядре 3.0. Некоторую (но недостаточно полную) справку по API

блочных устройств в ядре можно найти по ссылке: **http://www.hep.by/gnu/kernel/kernel-api/blkdev.html**.

Результаты тестирования

Теперь мы можем закончить рассмотрение созданных модулей блочных устройств объемным тестированием того, что получено:

```
# insmod block_mod_s.ko
# ls -l /dev/x*
brw-rw---- 1 root disk 252,  0 нояб. 13 02:01 /dev/xda
brw-rw---- 1 root disk 252, 16 нояб. 13 02:01 /dev/xdb
brw-rw---- 1 root disk 252, 32 нояб. 13 02:01 /dev/xdc
brw-rw---- 1 root disk 252, 48 нояб. 13 02:01 /dev/xdd
# hdparm -g /dev/xdd
/dev/xdd:
 geometry      = 128/4/16, sectors = 8192, start = 16
# mkfs.vfat /dev/xdd
mkfs.vfat 3.0.12 (29 Oct 2011)
```

Создадим некоторую структуру разделов с помощью fdisk (детали процесса не показаны из экономии — все делается тривиальным способом), после чего:

```
# fdisk -l /dev/xda
Диск /dev/xda: 4 МБ, 4194304 байт
4 heads, 16 sectors/track, 128 cylinders, всего 8192 секторов
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x2fb10b5b

Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/xda1              1        2000        1000   83  Linux
/dev/xda2           2001        5000        1500    5  Расширенный
/dev/xda3           5001        8191        1595+  83  Linux
/dev/xda5           2002        3500         749+  83  Linux
/dev/xda6           3502        5000         749+  83  Linux
# ls -l /dev/x*
brw-rw---- 1 root disk 252,  0 нояб. 13 02:11 /dev/xda
brw-rw---- 1 root disk 252,  1 нояб. 13 02:11 /dev/xda1
brw-rw---- 1 root disk 252,  2 нояб. 13 02:11 /dev/xda2
brw-rw---- 1 root disk 252,  3 нояб. 13 02:11 /dev/xda3
brw-rw---- 1 root disk 252,  5 нояб. 13 02:11 /dev/xda5
brw-rw---- 1 root disk 252,  6 нояб. 13 02:11 /dev/xda6
brw-rw---- 1 root disk 252, 16 нояб. 13 02:01 /dev/xdb
brw-rw---- 1 root disk 252, 32 нояб. 13 02:01 /dev/xdc
brw-rw---- 1 root disk 252, 48 нояб. 13 02:06 /dev/xdd
```

Отформатируем пару разделов на /dev/xda (диск /dev/xdd я уже отформатировал ранее в FAT32 без разметки на разделы — как дискету):

```
# mkfs.ext2 /dev/xda1
mke2fs 1.42.3 (14-May-2012)
...
# fsck /dev/xda1
fsck из util-linux 2.21.2
e2fsck 1.42.3 (14-May-2012)
/dev/xda1: clean, 11/128 files, 38/1000 blocks
# mkfs.ext2 /dev/xda5
mke2fs 1.42.3 (14-May-2012)
...
# fsck /dev/xda5
fsck из util-linux 2.21.2
e2fsck 1.42.3 (14-May-2012)
# fsck /dev/xdd
fsck из util-linux 2.21.2
dosfsck 3.0.12, 29 Oct 2011, FAT32, LFN
/dev/xdd: 0 files, 0/2036 clusters
```

Теперь мы можем смонтировать диски (из числа отформатированных) в заранее созданные каталоги:

```
# ls /mnt
dska  dskb  dskc  dskd  dske  efi  iso
# mount -tvfat /dev/xdd /mnt/dskd
# mount -text2 /dev/xda1 /mnt/dska
# mount -text2 /dev/xda5 /mnt/dskb
# mount | grep /xd
/dev/xdd on /mnt/dskd type vfat (rw,relatime,fmask=0022,dmask=0022,codepage=cp437,
iocharset=ascii,shortname=mixed,errors=remount-ro)
/dev/xda1 on /mnt/dska type ext2 (rw,relatime)
/dev/xda5 on /mnt/dskb type ext2 (rw,relatime)
# df | grep /xd
/dev/xdd              4072            0      4072            0% /mnt/dskd
/dev/xda1              979           17       912            2% /mnt/dska
/dev/xda5              731           16       678            3% /mnt/dskb
```

И проверим файловые операции копирования на этих файловых системах (я уже не один раз повторял, что нет лучших POSIX-тестов, чем стандартные GNU-утилиты):

```
# echo ++++++++++++++++++++++ > /mnt/dska/f1
# cp /mnt/dska/f1 /mnt/dskb/f2
# cp /mnt/dskb/f2 /mnt/dskd/f3
# cat /mnt/dskd/f3
++++++++++++++++++++++
# tree /mnt/dsk*
/mnt/dska
|-- f1
`-- lost+found
```

```
/mnt/dskb
|-- f2
`-- lost+found
/mnt/dskc
/mnt/dskd
`-- f3
/mnt/dske
2 directories, 3 files
```

В итоге мы должны получить дисковые устройства (рис. 4.1), с точки зрения *любых* утилит Linux ничем не отличающиеся от существующих дисковых устройств, — например, допускающие выполнение операций группового копирования:

```
$ sudo cp block_mod.ELDD_14 /mnt/xda1 -R
$ sudo cp block_mod.ELDD_14 /mnt/xda2 -R
$ sudo cp block_mod.ELDD_14 /mnt/xdb -R
$ df | grep xd
/dev/xda1             2028           18     2010           1% /mnt/xda1
/dev/xda2             2028           18     2010           1% /mnt/xda2
/dev/xdb              3963           43     3716           2% /mnt/xdb
$ tree /mnt
/mnt
├── sysimage
│   └── home
├── xda1
│   └── block_mod.ELDD_14
│       ├── block_mod_e.c
│       ├── block_mod_e.fc15.hist
│       ├── block_mod_orig.tgz
│       └── Makefile
├── xda2
│   └── block_mod.ELDD_14
│       ├── block_mod_e.c
│       ├── block_mod_e.fc15.hist
│       ├── block_mod_orig.tgz
│       └── Makefile
└── xdb
    ├── block_mod.ELDD_14
    │   ├── block_mod_e.c
    │   ├── block_mod_e.fc15.hist
    │   ├── block_mod_orig.tgz
    │   └── Makefile
    └── lost+found [error opening dir]

9 directories, 12 files
```

И еще полезный пример, подтверждающий то, что реализация операций `open()` и `release()` для блочного устройства вовсе не является критически необходимой (за

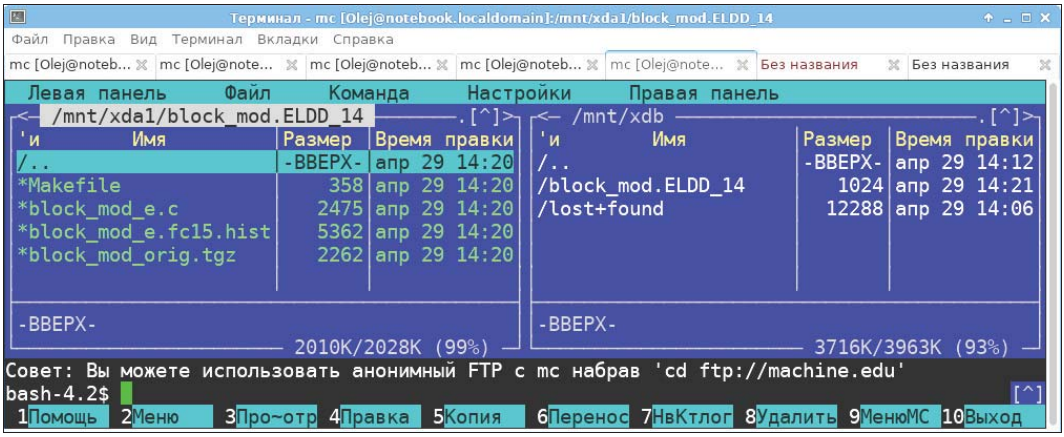


Рис. 4.1. Смонтированные блочные устройства /dev/xda и /dev/xdb

исключением случаев, когда они обеспечивают специальные цели — такие, например, как подсчет пользователей, работающих с устройством):

```
# lsmod | head -n4
Module                  Size  Used by
vfat                   17208  1
fat                    54611  1 vfat
block_mod_s            13210  4
# rmmod block_mod_s
Error: Module block_mod_s is in use
```

Число счетчика ссылок на модуль не нулевое! ...естественно, до тех пор, пока мы не размонтируем все используемые диски этого модуля:

```
# umount /mnt/dska
# umount /mnt/dskb
# umount /mnt/dskd
# lsmod | grep ^block
block_mod_s            13210  0
# rmmod block_mod_s
# ls /dev/xd*
ls: невозможно получить доступ к /dev/xd*: Нет такого файла или каталога
```

А вот что показывает *сравнительное* тестирование скорости созданных дисковых устройств, выполненное типовой GNU-утилитой, для разных выбранных стратегий обработки запросов ввода/вывода:

```
# insmod block_mod_s.ko mode=0
# hdparm -t /dev/xda
/dev/xda:
 Timing buffered disk reads:   4 MB in  0.01 seconds = 318.32 MB/sec

# insmod block_mod_s.ko mode=1
# hdparm -t /dev/xda
```

```
/dev/xda:
 Timing buffered disk reads:   4 MB in  0.03 seconds = 116.02 MB/sec

# insmod block_mod_s.ko mode=2
# hdparm -t /dev/xda
/dev/xda:
 Timing buffered disk reads:   4 MB in  0.01 seconds = 371.92 MB/sec
```

Видны весьма значительные различия в производительности... но это уже предмет для совсем другого рассмотрения.

И еще несколько примеров с перманентным сохранением данных:

```
$ time dd if=/dev/zero of=./XXX bs=512 count=10000
10000+0 записей считано
10000+0 записей написано
 скопировано 5120000 байт (5,1 MB), 0,0414243 с, 124 MB/c
real    0m0.086s
user    0m0.007s
sys     0m0.049s
$ ls -l XXX
-rw-rw-r--. 1 olej olej 5120000 апр.  26 16:18 XXX
```

Здесь мы создали «сырой» файл размером 5 Мбайт, который и будет образом носителя для нашего будущего блочного устройства.

Загружаем модуль блочного устройства, который загружает и выгружает свои данные в этот файл между сеансами своего использования:

```
$ sudo insmod dubfl.ko file=XXX
$ ls -l /dev/db*
brw-rw----. 1 root disk 252, 1 апр.  26 16:20 /dev/dbf
$ sudo hdparm -t /dev/dbf
/dev/dbf:
 Timing buffered disk reads:   4 MB in  0.01 seconds = 393.04 MB/sec
$ sudo mkfs.vfat /dev/dbf
mkfs.vfat 3.0.12 (29 Oct 2011)
$ ls /mnt
common
$ sudo mount -t vfat /dev/dbf /mnt/common/
$ df | grep dbf
/dev/dbf              4974            0     4974            0% /mnt/common
$ sudo cp -R block_mod.ELDD_14 /mnt/common
$ tree /mnt/common/
/mnt/common/
└── block_mod.ELDD_14
    ├── block_mod_e.c
    ├── block_mod_e.fc15.hist
    ├── block_mod_e.ko
    ├── block_mod_orig.tgz
    └── Makefile
 1 directory, 5 files
```

```
$ lsmod | head -n5
Module                  Size  Used by
vfat                   17209  1
fat                    54645  1 vfat
dubfl                  13074  1
tcp_lp                 12584  0
$ sudo umount /mnt/common
$ lsmod | head -n5
Module                  Size  Used by
vfat                   17209  0
fat                    54645  1 vfat
dubfl                  13074  0
tcp_lp                 12584  0
$ sudo rmmod dubfl
```

Здесь мы выгрузили модуль из памяти, но можем, для достоверности, вообще перезагрузить компьютер, а затем возобновить работу со своими данными:

```
$ sudo reboot
...
$ sudo insmod dubfl.ko file=XXX
$ sudo mount -t vfat /dev/dbf /mnt/common/
$ ls -R /mnt/common
/mnt/common:
block_mod.ELDD_14
/mnt/common/block_mod.ELDD_14:
block_mod_e.c  block_mod_e.fc15.hist  block_mod_e.ko  block_mod_orig.tgz  Makefile
$ df | grep dbf
/dev/dbf              4974          176     4798            4% /mnt/common
```

Файловая система FUSE

Задача создания драйвера блочных устройств, во-первых, и весьма сложна, и не очевидна, и меняется от версии к версии, а во-вторых, не так часто возникает перед прикладным разработчиком. Именно поэтому мы не стали ранее адаптировать рабочие примеры под следующие версии ядра, где они рабочими быть перестали... Но основные идеи, канва, там понятны, и они остаются неизменными от версии к версии.

Но есть и совершенно иной путь создания драйверов блочных устройств, ставший в последнее десятилетие массово применяемым в разнообразных проектах, и это файловая система FUSE (Filesystem in Userspace). Сложность и трудоемкость работы при использовании этой технологии снижается в разы, и она вполне подъемна для среднего начинающего, но упрямого студента...

Файловая система FUSE — активно развивающийся до сих пор проект, позволяющий нам строить свои самые разнообразные файловые системы, вообще не прибегая к программированию в ядре. Начиная с ядра 2.6.14, файловая система FUSE включена как стандартный компонент дерева исходных кодов ядра Linux. Огром-

ным преимуществом FUSE является то, что для монтирования файловых систем FUSE не требуются привилегии суперпользователя root.

Идея FUSE состоит в том, чтобы сформировать в своем *приложении* (в адресном пространстве пользователя) таблицу *файловых операций* (во многом подобно тому, как мы это делаем в драйверах устройств ядра), но несколько другого определения (<fuse.h>):

```
struct fuse_operations {
        int (*getattr) (const char *, struct stat *);
        int (*readlink) (const char *, char *, size_t);
        /* Deprecated, use readdir() instead */
        int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);
        int (*mknod) (const char *, mode_t, dev_t);
        int (*mkdir) (const char *, mode_t);
        int (*unlink) (const char *);
        int (*rmdir) (const char *);
        int (*symlink) (const char *, const char *);
        int (*rename) (const char *, const char *);
        int (*link) (const char *, const char *);
        int (*chmod) (const char *, mode_t);
        int (*chown) (const char *, uid_t, gid_t);
        int (*truncate) (const char *, off_t);
        /* Deprecated, use utimens() instead. */
        int (*utime) (const char *, struct utimbuf *);
        int (*open) (const char *, struct fuse_file_info *);
        int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);
        int (*write) (const char *, const char *, size_t, off_t, struct fuse_file_info *);
        int (*statfs) (const char *, struct statvfs *);
        int (*flush) (const char *, struct fuse_file_info *);
        int (*release) (const char *, struct fuse_file_info *);
        int (*fsync) (const char *, int, struct fuse_file_info *);
        int (*setxattr) (const char *, const char *, const char *, size_t, int);
        int (*getxattr) (const char *, const char *, char *, size_t);
        int (*listxattr) (const char *, char *, size_t);
        int (*removexattr) (const char *, const char *);
        int (*opendir) (const char *, struct fuse_file_info *);
        int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t, struct fuse_file_info *);
        int (*releasedir) (const char *, struct fuse_file_info *);
        int (*fsyncdir) (const char *, int, struct fuse_file_info *);
        void *(*init) (struct fuse_conn_info *conn);
        void (*destroy) (void *);
        int (*access) (const char *, int);
        int (*create) (const char *, mode_t, struct fuse_file_info *);
        int (*ftruncate) (const char *, off_t, struct fuse_file_info *);
        int (*fgetattr) (const char *, struct stat *, struct fuse_file_info *);
        int (*lock) (const char *, struct fuse_file_info *, int cmd, struct flock *);
        int (*utimens) (const char *, const struct timespec tv[2]);
```

```
        int (*bmap) (const char *, size_t blocksize, uint64_t *idx);
        unsigned int flag_nullpath_ok:1;
        unsigned int flag_nopath:1;
        unsigned int flag_utime_omit_ok:1;
        unsigned int flag_reserved:29;
        int (*ioctl) (const char *, int cmd, void *arg, struct fuse_file_info *,
                      unsigned int flags, void *data);
        int (*poll) (const char *, struct fuse_file_info *, struct fuse_pollhandle *ph,
                     unsigned *reventsp);
        int (*write_buf) (const char *, struct fuse_bufvec *buf, off_t off,
                          struct fuse_file_info *);
        int (*read_buf) (const char *, struct fuse_bufvec **bufp, size_t size, off_t off,
                         struct fuse_file_info *);
        int (*flock) (const char *, struct fuse_file_info *, int op);
        int (*fallocate) (const char *, int, off_t, off_t, struct fuse_file_info *);
};
```

Остроумная идея FUSE состоит в том, что, получив из *любого* пользовательского приложения запрос на выполнение файлового системного вызова к этой системе, ядро не выполняет этот запрос самостоятельно, а «перепасовывает» его на выполнение обратно в пользовательское пространство (демону FUSE) согласно этой определенной нами таблице (рис. 4.2).

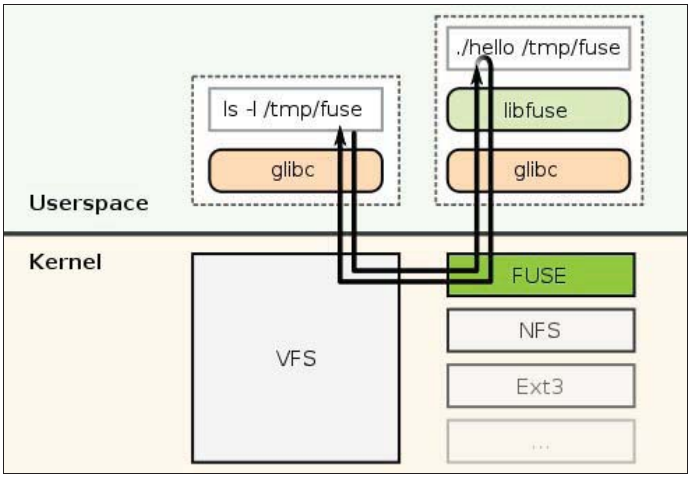


Рис. 4.2. Взаимодействие компонентов FUSE при работе пользовательской файловой системы (рисунок из документации проекта FUSE)

Не ужасайтесь общим размером показанной ранее таблицы операций — из них всех нам нужно определить в своем приложении только самый минимум конкретно необходимых операций — все остальные либо принимают разумные реализации по умолчанию, либо ошибкой сообщают о нереализованности при их исполнении:

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

```
    fuse_operations own_oper = {                // таблица операций
        .getattr = own_getattr,                 // атрибуты файла
        .mknod   = own_mknod,
        .mkdir   = own_mkdir,
        .unlink  = own_unlink,                  // удалить файл
        .rmdir   = own_rmdir,                   // удалить каталог
        .rename  = own_rename,
        .open    = own_open,
        .read    = own_read,                    // чтение открытого файла
        .write   = own_write,                   // запись в открытый файл
        .readdir = own_readdir,
    }
 ...
    return fuse_main(argc, argv, &my_oper, NULL); // передаем данные ядру ОС (модулю FUSE)
}
```

И все! Вот так и выглядит общая схема *любой* реализации пользовательской файловой системы FUSE — после запуска приложение завершается, а новая файловая система остается смонтированной к указанному в `argv[1]` пути (как будет показано вскоре).

Сама подсистема FUSE, скорее всего, уже установлена в вашем дистрибутиве по умолчанию — ввиду ее применения во многих десятках проектов, особенно в области мобильных гаджетов: цифровых фотоаппаратов, видеокамер, диктофонов...

```
$ aptitude search fuse | grep ^i
i  exfat-fuse - драйвер чтения и записи exFAT для FUSE
i  fuse - файловая система в пользовательском окружении
i  gvfs-fuse - userspace virtual filesystem - fuse server
i  ifuse - FUSE-модуль для устройств iPhone и iPod Touch
i  libfuse2 - Filesystem in Userspace (library)
```

Но вот средства разработки к ней (заголовочные файлы, библиотеки) вам придется устанавливать дополнительно, что проверяется так:

```
$ pkg-config fuse --cflags --libs
Package fuse was not found in the pkg-config search path.
Perhaps you should add the directory containing `fuse.pc'
to the PKG_CONFIG_PATH environment variable
No package 'fuse' found
```

Но этому легко помочь:

```
$ sudo apt install libfuse-dev
Чтение списков пакетов... Готово
Построение дерева зависимостей
Чтение информации о состоянии... Готово
Следующие НОВЫЕ пакеты будут установлены:
  libfuse-dev
Обновлено 0 пакетов, установлен 1 новый пакет, для удаления отмечено 0 пакетов, и 0 пакетов
не обновлено.
```

```
Необходимо скачать 105 kB архивов.
После данной операции объем занятого дискового пространства возрастет на 596 kB.
Пол:1 http://ubuntu.volia.net/ubuntu-archive focal/main amd64 libfuse-dev amd64 2.9.9-3 [105 kB]
Получено 105 kB за 0с (752 kB/s)
Выбор ранее не выбранного пакета libfuse-dev.
(Чтение базы данных ... на данный момент установлено 477514 файлов и каталогов.)
Подготовка к распаковке .../libfuse-dev_2.9.9-3_amd64.deb ...
Распаковывается libfuse-dev (2.9.9-3) ...
Настраивается пакет libfuse-dev (2.9.9-3) ...
$ pkg-config fuse --cflags --libs
-D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -lfuse -pthread
```

Реализация функциональности FUSE в ядре обеспечивается модулем, если сборщики конфигурировали его как модуль (показан дистрибутив LMDE5):

```
$ lsmod | grep fuse
fuse                  167936  5
$ grep FUSE /lib/modules/`uname -r`/build/.config
CONFIG_FUSE_FS=m
CONFIG_FUSE_DAX=y
```

Однако подсистема настолько популярна, что некоторые ее просто вкомпилируют в ядро (Mint 20.3), и модуля `FUSE` вы здесь просто не увидите:

```
$ grep FUSE /lib/modules/`uname -r`/build/.config
CONFIG_FUSE_FS=y
# CONFIG_AUFS_BR_FUSE is not set
```

А в пространстве пользователя поддержка обеспечивается библиотеками, которые мы и установили ранее:

```
$ ls -lL /usr/lib/x86_64-linux-gnu/*fuse*
-rw-r--r-- 1 root root 373720 мар  7  2020 /usr/lib/x86_64-linux-gnu/libfuse.a
-rw-r--r-- 1 root root 252184 мар  7  2020 /usr/lib/x86_64-linux-gnu/libfuse.so
-rw-r--r-- 1 root root 252184 мар  7  2020 /usr/lib/x86_64-linux-gnu/libfuse.so.2
-rw-r--r-- 1 root root 252184 мар  7  2020 /usr/lib/x86_64-linux-gnu/libfuse.so.2.9.9
```

> ***Примечание***
>
> У FUSE есть такая странная особенность, что в коде реализации приложения обязательно должна быть явно указана версия используемой ветки FUSE (без этого сборка завершится ошибкой). Например так:
>
> ```
> #define FUSE_USE_VERSION 26
> ```

Детальный разбор кода файловых систем FUSE выходит за рамки нашей тематики, но несколько более или менее работоспособных примеров (для экспериментирования) собраны в папке FUSE сопровождающего книгу файлового архива. И выглядит это в общем виде как-то так (показана ущербная, схематичная реализация, но достаточная для того, чтобы наметить способ реализации):

```
$ ./fuse ./root
$ mount | grep fuse.fuse
/home/olej/2022/own.BOOKs/BHV.kernel/examples.inWORK/FUSE/fuse.c/fuse on
```

```
/home/olej/2022/own.BOOKs/BHV.kernel/examples.inWORK/FUSE/fuse.c/root
type fuse.fuse (rw,nosuid,nodev,relatime,user_id=1000,group_id=1000)
```

Приложение завершилось, но файловая система примонтирована к указанной точке и готова к использованию (обратите внимание, что все это сделано от лица ординарного пользователя):

```
$ ls -l root
итого 0
$ mkdir root/d1
$ mkdir root/d1/d2
$ tree root
root
└── d1
    └── d2
2 directories, 0 files
$ touch root/d1/f1
touch: установка временных отметок 'root/d1/f1': Функция не реализована
$ touch root/d1/d2/f2
touch: установка временных отметок 'root/d1/d2/f2': Функция не реализована
$ tree root
root
└── d1
    ├── d2
    │   └── f2
    └── f1

2 directories, 2 files
$ ls -lR root
root:
итого 0
drw-rw-rw- 2 root root 0 янв  1  1970 d1

root/d1:
итого 0
drw-rw-rw- 2 root root 0 янв  1  1970 d2
-rw-rw-rw- 1 root root 0 янв  1  1970 f1

root/d1/d2:
итого 0
-rw-rw-rw- 1 root root 0 янв  1  1970 f2
$ umount ./root
$ tree root
root
0 directories, 0 files
```

Возникает закономерный вопрос: если собранное приложение FUSE после запуска завершается, а код выполняется в форме демона, то как отлаживать код реализующих функций в таблице операций?

```
$ ./fuse root
$ ps -A | grep fuse
  26993 ?        00:00:00 fuse
```

Для этого есть вариант запуска приложения (который не легко обнаружить по документации) с отладочной опцией `-d` — тогда ваше приложение будет выполняться без демонизации и с выводом как собственных отладочных сообщений, так и ваших собственных из кода функций:

```
$ ./fuse ./r1 -d
FUSE library version: 2.9.9
nullpath_ok: 0
nopath: 0
utime_omit_ok: 0
unique: 2, opcode: INIT (26), nodeid: 0, insize: 56, pid: 0
INIT: 7.31
flags=0x03fffffb
max_readahead=0x00020000
...
```

Такой отладочный режим будет диагностировать пошаговую реакцию на каждое ваше действие со стандартными утилитами над файловой системой: `rmmod`, `ls`, `cat`, `echo` ...

Главная же цель приведенных примеров состоит в том, чтобы показать, что для тестирования и использования такой файловой системы, созданной из пользовательского приложения, мы не используем никаких возможностей и команд сверх всего набора утилит POSIX/Linux.

Из сказанного уже понятно — и это огромное достоинство FUSE, — что разработку новых файловых систем (потому что это пользовательское пространство) можно вести не только на языке C, но на *любом* из множества языков программирования высокого уровня, представленных в Linux: C++, Python, Java, Go... И множество инструментов-интерфейсов на различных языках программирования от сторонних разработчиков не заставило себя ждать! Например, для Python вам нужно установить:

```
$ sudo pip install fusepy
Collecting fusepy
  Downloading fusepy-3.0.1.tar.gz (11 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: fusepy
  Building wheel for fusepy (setup.py) ... done
  Created wheel for fusepy: filename=fusepy-3.0.1-py3-none-any.whl size=10503
                      sha256=4149080183205ba5bcd1a46fe6700d493437338caad15491a84b6ad038037156
  Stored in directory: /root/.cache/pip/wheels/7f/41/10/
                                          f70b83a1164fdb95e7bc37bace13114a024227e56c2fee02bb
Successfully built fusepy
Installing collected packages: fusepy
Successfully installed fusepy-3.0.1
```

Интерфейс /proc

Интерфейс к файловым именам /proc (procfs) и более поздний (по времени создания) интерфейс к именам /sys (sysfs) рассматривается как канал передачи диагностической информации из модуля и управляющей — в него. Такой способ взаимодействия с модулем может полностью заменить средства вызова `ioctl()` для устройств — устаревшего и считающегося опасным (из-за отсутствия контроля типизации в `ioctl()`).

Любое изменение в состояниях подсистемы ввода/вывода: загрузка нового модуля ядра, горячее подключение/отключение реального устройства, и многие другие события будут порождать многочисленные *автоматические* изменения в иерархии файловых структур /proc и /sys и изменения содержимого их псевдофайлов. Но эти изменения происходят, как уже сказано, *автоматически* — независимо от выполнения кода пользователя (средствами sysfs, сокета `netlink` и подсистемы udev, которые уже упоминались). Нас же в этой части рассмотрения будут интересовать возможности *ручного* создания путевых имен в /proc и /sys (или древовидной иерархии имен). Эти имена должны создаваться *сверх* действий sysfs — из кода загружаемого пользовательского модуля. Такие именованные псевдофайлы и должны служить шлюзами из пользовательского пространства к значениям переменных внутри кода модуля.

В настоящее время сложилась тенденция некоторые управляющие функции переносить из /proc в /sys — отображения путевых имен модулем в эти две подсистемы по своему назначению и возможностям являются очень подобными. Содержимое имен-псевдофайлов в обоих системах является только *текстовым* отображением некоторых внутренних данных ядра. Но нужно иметь в виду и ряд различий между ними:

- файловая система /proc является общей, «родовой» принадлежностью всех систем UNIX (Free/Open/Net BSD, Solaris, QNX, MINIX 3, ...) — ее наличие и общие принципы использования оговариваются стандартом POSIX 2, а файловая система /sys представляет собой сугубо «изобретение» Linux 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 создается их много больше по числу имен, но каждое из них дает только информацию о единичном значении (в символьном представлении!), часто соответствующем одной элементарной переменной языка C: `int`, `long`, ...

Вот, например, здоровенная таблица — описание 40 ядер/процессоров *x*86_64 сервера промышленного уровня DELL PowerEdge R420:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 62
```

```
model name      : Intel(R) Xeon(R) CPU E5-2470 v2 @ 2.40GHz
stepping        : 4
microcode       : 0x42e
cpu MHz         : 1828.550
cache size      : 25600 KB
...
$ wc -l /proc/cpuinfo
1080 /proc/cpuinfo
```

А вот то же самое, но у миниатюрного одноплатного ARM «наладонника» Orange Pi One:

```
$ cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 5 (v7l)
BogoMIPS : 48.00
Features : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer  : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc07
CPU revision    : 5
...
$ wc -l /proc/cpuinfo
43 /proc/cpuinfo
```

А теперь — для сравнения — некоторые образцы информации (выбранной достаточно наугад) системы /sys:

```
$ tree /sys/module/printk
/sys/module/printk
├── parameters
│   ├── always_kmsg_dump
│   ├── console_no_auto_verbose
│   ├── console_suspend
│   ├── ignore_loglevel
│   └── time
└── uevent

1 directory, 6 files
$ cat /sys/module/printk/parameters/*
N
N
Y
N
Y
```

Значения в /sys представляются зачастую в виде довольно развесистого поддерева, в каждой терминальной ветви (листе) которого содержится единичное значение: символ, строка, число...

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 создается только для чтения, а интерфейс в /sys — для чтения и записи. Это совершенно неверно — оба интерфейса допускают и чтение, и запись.

И там и там интересующие нас значения (для чтения, а часто и для записи) находятся в терминальных узлах (листьях) дерева каждой из этих файловых систем.

Терминальные значения в */proc* и */sys*

Еще одна из особенностей, которая вообще присуща *философии* UNIX, но особенно явно прослеживается при работе с именами в /proc и *особенно* в /sys: все считываемые и записываемые значения представляются только в *символьном*, *небинарном*, формате[3]. И здесь — при преобразовании символьных строк в числовые значения в коде ядра — возникает определенное замешательство: в API ядра нет многочисленных вызовов, подобных POSIX API `atol()` и других подобных. Но при этом на помощь приходят функции, определенные в <linux/kernel.h>:

```
extern unsigned long simple_strtoul(const char *cp, char **endp, unsigned int base);
extern long simple_strtol(const char *cp, char **endp, unsigned int base);
extern unsigned long long simple_strtoull(const char *cp, char **endp, unsigned int base);
extern long long simple_strtoll(const char *cp, char **endp, unsigned int base);
```

Здесь везде:

- `cp` — указатель преобразовываемой символьной строки;
- `endp` — возвращаемый указатель на позицию, где завершилось преобразование (символ, который не мог быть преобразован), — может быть и `NULL`, если не требуется;
- `base` — система счисления (нулевое значение в этой позиции эквивалентно 10);

Возвращают все эти вызовы преобразованные целочисленные значения. Вот примеры их возможного использования:

```
long j = simple_strtol("-1000", NULL, 16);
long j = simple_strtol("-1000 значение", pchr, 0);
```

Для более сложных (но и более склонных к ошибкам) преобразований там же определены функции, подобные эквивалентам в POSIX:

```
extern int sscanf(const char *cp, const char *format, ...) ;
extern int vsscanf(const char *cp, const char *format, va_list ap);
```

Функции возвращают количество успешно преобразованных и назначенных полей. Примеры:

```
char str[ 80 ];
int d, y, n;
...
```

[3] Здесь нет реестров, ульев, пчел… и всяких других прелестей систем Windows!

```
n = sscanf("January 01 2000", "%s%d%d", str, &d, &y);
...
sscanf("24\tLewis Carroll", "%d\t%s", &d, str);
```

Параметры, как легко видеть из примеров, должны поступать в вызов `sscanf()` *по ссылке* (а контроль соответствия типов из-за переменного числа параметров отсутствует, что и служит источником ошибок и поэтому требует повышенной тщательности).

Обратные преобразования (численные значения в строку) производятся, как мы уже неоднократно видели, вызовом `sprintf()`.

Аналогичные преобразования форматов представлений, естественно, возникают не только при работе с файловыми системами /proc и /sys — та же необходимость возникает временами при работе с устройствами /dev и с сетевой подсистемой... Но при работе с именами в /proc и, как уже было отмечено, особенно в /sys эта потребность возникает *повсеместно*. Поэтому сделать акцент на форматные преобразования показалось уместным именно здесь.

Использование */proc*

Описываемая далее программная техника организации интерфейса в /proc ориентируется на API и структуры данных, используемые в ядре до версии 3.9 (включительно). Все описания для этого сосредоточены в заголовочном файле <linux/proc_fs.h>. После версии ядра 3.10 API и структуры активно изменяются (описание ключевой структуры вообще вынесено из <include> и находится в <linux/fs/proc/internal.h>). Но, поскольку даже в комментариях исходного кода записано, что это пока черновая (draft) реализация идеи, мы ее пока не станем рассматривать.

По аналогии с тем, как все операции в /dev обеспечивались через таблицу файловых операций (`struct file_operations`), все операции над именами в /proc увязываются (определения в <linux/proc_fs.h>) через специфическую для этих целей довольно обширную структуру (показаны только поля, интересные для последующего рассмотрения):

```
struct proc_dir_entry {
...
        unsigned short namelen;
        const char *name;
        mode_t mode;
...
        uid_t uid;
        gid_t gid;
        loff_t size;
...
        /*
         * NULL ->proc_fops means "PDE is going away RSN" or
         * "PDE is just created". In either case, e.g. ->read_proc won't be
         * called because it's too late or too early, respectively.
```

```
         *
         * If you're allocating ->proc_fops dynamically, save a pointer
         * somewhere.
         */
        const struct file_operations *proc_fops;
...

        read_proc_t *read_proc;
        write_proc_t *write_proc;
...
};
```

Но отличает эту структуру именно то, что она из числа очень старых образований Linux, с ней происходили многочисленные изменения и накладки — так, на нее наложились требования совместимости снизу вверх с предыдущими реализациями.

Наличие в структуре *одновременно* полей `read_proc` и `write_proc`, а параллельно с ними уже известной нам таблицы файловых операций `proc_fops` может быть подсказкой, и так оно и оказывается на самом деле — для описания операций ввода/вывода над /proc существуют два *альтернативных* (на выбор) способа их определения:

- использовать *специфический* для имен в /proc программный интерфейс в виде функций:

```
typedef int (read_proc_t)(char *page, char **start, off_t off,
                          int count, int *eof, void *data);
typedef int (write_proc_t)(struct file *file, const char __user *buffer,
                           unsigned long count, void *data);
```

- использовать *общий* механизм доступа к именам файловой системы, а именно — таблицу файловых операций `proc_fops` в составе структуры.

Теперь, когда мы кратно пробежались на качественном уровне по свойствам интерфейсов, можно перейти к примерам кода модулей, реализующих обсуждаемые механизмы (интерфейс /proc рассматривается на примерах из папки procfs/proc сопровождающего книгу файлового архива).

Специфический механизм *procfs*

Первое, что явно бросается в глаза, — это радикальной различие прототипов функций `read_proc_t` и `write_proc_t` (начиная с того, что, как будет показано далее, первая из них вообще не оперирует с адресами пользовательского пространства, а вторая делает именно это). Это связано и с тем, что функция записи типа `write_proc_t` появилась в API ядра значительно позже, чем функция чтения `read_proc_t`, и по своей семантике значительно проще последней. Пока мы сосредоточимся на функции чтения, а позже вернемся на пару слов и к записи.

Мы будем собирать целую линейку однотипных модулей, поэтому общую часть определений вынесем в отдельный файл:

common.h:

```
#define NAME_DIR  "mod_dir"
#define NAME_NODE "mod_node"
#define LEN_MSG 160

#define LOG(...) printk(KERN_INFO "! "__VA_ARGS__)
#define ERR(...) printk(KERN_ERR "! "__VA_ARGS__)
```

mod_proc.h

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/stat.h>
#include <linux/uaccess.h>
#include "common.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");

static int  __init proc_init(void);
static void __exit proc_exit(void);

module_init(proc_init);       // предварительные определения
module_exit(proc_exit);
```

Сценарий (файл Makefile) сборки многочисленных модулей этого проекта — традиционный (см. папку procfs в сопровождающем книгу файловом архиве), поэтому мы не будем на нем останавливаться.

Основную работу по созданию и уничтожению имени в /proc выполняет пара вызовов (<linux/proc_fs.h>):

```
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode,
                                         struct proc_dir_entry *parent);
void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
```

Эти операции, что будет показано далее, оперируют в равной мере как с новыми *каталогами* в /proc (создание иерархии имен), так и с конечными *терминальными* именами, являющимися источниками данных, — т. е. имя структуры `proc_dir_entry` не должно вводить в заблуждение. Обеспечивается такая универсальность вторым параметром (`mode`) вызова `create_proc_entry()`. В результате такого вызова и создается структура *описания имени*, которую мы уже видели ранее:

```
struct proc_dir_entry {
...
        read_proc_t *read_proc;
```

```
        write_proc_t *write_proc;
...
};
```

Первый пример показывает создание интерфейса к модулю в /proc, доступного только для чтения из пользовательских программ (наиболее частый на практике случай):

mod_procr.c

```
#include "mod_proc.h"
#include "proc_node_read.c"

static int __init proc_init(void) {
   int ret;
   struct proc_dir_entry *own_proc_node;
   own_proc_node = create_proc_entry(NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO, NULL);
   if(NULL == own_proc_node) {
      ret = -ENOENT;
      ERR("can't create /proc/%s", NAME_NODE);
      goto err_node;
   }
   own_proc_node->uid = own_proc_node->gid = 0;
   own_proc_node->read_proc = proc_node_read;
   LOG("/proc/%s installed", NAME_NODE);
   return 0;
err_node:
   return ret;
}

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

Здесь и далее флаги прав доступа к файлу вида `S_I*` (известные и из пользовательского API) ищите и заимствуйте из <linux/stat.h>. Напомним только, что название результирующей структуры `struct proc_dir_entry`, создаваемой при регистрации имени функцией `create_proc_entry()` (и такого же типа третий параметр вызова в этой функции), не должно вводить в заблуждение, — это не обязательно каталог, и будет ли это каталог или терминальное файловое имя, определяется значением флагов (параметр `mode`). Относительно третьего параметра вызова (`parent`) отметим, что это, как прямо следует из его имени, *ранее созданный* родительский каталог в /proc, внутри которого создается имя, а если он равен `NULL`, то имя создается непосредственно в /proc. Эта техника позволяет создавать в /proc произвольной сложности иерархии имен. Все это хорошо видно в кодах примеров.

Сама реализация функции чтения (типа `read_proc_t`) будет использована в различных модулях и вынесена в файл proc_node_read.c:

proc_node_read.c

```
#include "common.h"

// в точности списан прототип read_proc_t из <linux/proc_fs.h> :
static ssize_t proc_node_read(char *buffer, char **start, off_t off,
                              int count, int *eof, void *data) {
   static char buf_msg[LEN_MSG + 1] =
          ".........1.........2.........3.........4.........5.........6\n";
   LOG("read: %d (buffer=%p, off=%ld)", count, buffer, off);
   strcpy(buffer, buf_msg);
   LOG("return bytes: %d%s", strlen(buf_msg), *eof != 0 ? " ... EOF" : "");
   return strlen(buf_msg);
};
```

Для этой реализация оказывается *достаточно* фактически единственной операции `strcpy()` — независимо от размера запрашиваемых данных, но все это требует некоторых более детальных пояснений:

- интерфейс `read_proc_t` развивается давно и приобрел за это время довольно причудливую семантику, отражаемую множеством используемых параметров;
- в функции `read_proc_t` *не осуществляется* обмен между пространствами ядра и пользователя, она только передает данные на некоторый *промежуточный слой* ядра, который обеспечивает позиционирование данных, их дробление в соответствии с запрошенным размером и обмен данными с пространством пользователя;
- функция ориентирована на *одностраничный* обмен памятью, потому ее прототип записывается так (первый параметр):

```
ssize_t proc_node_read(char *page, char **start, off_t off,
                       int count, int *eof, void *data)
```

- но соблюдения условий: а) объем передаваемых данных укладывается *в одну страницу* (3 килобайта данных при 4-килобайтной гранулярности страниц памяти операционной системы), что и требуется в подавляющем большинстве случаев, и б) считываемые из /proc данные *не изменяются* при последовательных чтениях до достижения конца файла — вполне достаточно для такой *простейшей* реализации, как приведенная здесь (практически ничего из параметров, кроме адреса выходных данных, не использовано).

> ***Примечание***
>
> Более детальное описание (но тоже достаточно невнятное и путанное) семантики `read_proc_t`, назначения ее параметров и использования в редких случаях работы с большими объемами данных, можно найти в [2]. А для особо любознательных мы еще коротко вернемся к вариантам реализации `read_proc_t` далее...

В литературе иногда утверждается, что для /proc нет API для записи, аналогичного API для чтения, но с некоторых пор (позже, чем для чтения) в <linux/proc_fs.h> появилось и такое описание типа `write_proc_t` (аналогичного типу `read_proc_t`):

```
typedef int ()(struct file *file, const char __user *buffer,
                          unsigned long count, void *data);
```

Как легко видеть, это определение и по прототипу, и по логике (обмен с пространством пользователя) заметно отличается от `read_proc_t`;

Наконец, на последнем шаге нашей подготовительной работы мы создадим приложение (тестовое, пользовательского пространства), подобное утилите `cat`, но позволяющее параметром запуска (число) указать размер объема данных, запрашиваемых к считыванию за один раз в процессе последовательного циклического чтения (утилита `cat`, например, всегда запрашивает максимум — 32 767 байтов за одно чтение `read()`):

mcat.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include "common.h"

static int get_proc(void) {
   char dev[ 80 ];
   int df;
   sprintf(dev, "/proc/%s", NAME_NODE);
   if((df = open(dev, O_RDONLY)) < 0) {
      sprintf(dev, "/proc/%s/%s", NAME_DIR, NAME_NODE);
      if((df = open(dev, O_RDONLY)) < 0)
         printf("open device error: %m\n"), exit(EXIT_FAILURE);
   }
   return df;
}

int main(int argc, char *argv[]) {
   int df = get_proc(),
       len = (argc > 1 && atoi(argv[ 1 ]) > 0) ?
             atoi(argv[ 1 ]) : LEN_MSG;
   char msg[ LEN_MSG + 1 ] = "";
   char *p = msg;
   int res;
   do {
      if((res = read(df, p, len)) >= 0) {
         *(p += res)  = '\0';
         printf("read + %02d bytes, input buffer: %s", res, msg);
         if(*(p - 1) != '\n') printf("\n");
      }
```

```
      else printf("read device error: %m\n");
   } while (res > 0);
   close(df);
   return EXIT_SUCCESS;
};
```

Теперь все готово к испытаниям первого из полученных модулей (учтите только, что все показанное в части модулей этой главы работает с ядром 3.9 или более ранним):

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

Хорошо видно, откуда возникла цифра 3 Кбайт — 3072. Детально природу того, что происходит «под крышкой», просмотрим с помощью созданного тестового приложения, изменяя размер запрашиваемых данных:

```
$ ./mcat 10
read + 10 bytes, input buffer: .........1
read + 10 bytes, input buffer: .........1.........2
read + 10 bytes, input buffer: .........1.........2.........3
read + 10 bytes, input buffer: .........1.........2.........3.........4
read + 10 bytes, input buffer: .........1.........2.........3.........4.........5
read + 10 bytes, input buffer: .........1.........2.........3.........4.........5.........6
read + 01 bytes, input buffer: .........1.........2.........3.........4.........5.........6
read + 00 bytes, input buffer: .........1.........2.........3.........4.........5.........6
$ dmesg | tail -n50 | grep -v ^audit
! /proc/mod_node installed
! read: 10 (buffer=f32b1000, off=0)
! return bytes: 61
! read: 10 (buffer=f32b1000, off=10)
```

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

Завершение работы модуля:

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

В следующем примере выполняется регистрация имени в /proc (в точности то же самое, что и в предыдущем примере), но более простым и чаще описанным в литературе способом, — вызывая `create_proc_read_entry()` (но этот способ просто скрывает суть происходящего):

mod_procr2.c

```
#include "mod_proc.h"
#include "proc_node_read.c"

static int __init proc_init(void) {
   if(create_proc_read_entry(NAME_NODE, 0, NULL, proc_node_read, NULL) == 0) {
      ERR("can't create /proc/%s", NAME_NODE);
      return -ENOENT;
   }
   LOG("/proc/%s installed", NAME_NODE);
```

```
   return 0;
}

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

Вот и все, что нужно вашему модулю для того, чтобы начать отображать себя в /proc!

> ***Примечание (важно!)***
>
> Наличие функции `create_proc_read_entry()` — пример того, что API ядра, доступный программисту, намного шире, чем список экспортируемых имен в /boot/System.map-*, и включает ряд вызовов, которых вы не найдете в /proc/kallsyms (в том числе и `create_proc_read_entry()`). Это происходит за счет достаточно широкого использования `inline`-определений (синтаксическое расширение GCC):
>
> ```
> $ cat /proc/kallsyms | grep create_proc_
> c0522237 T create_proc_entry
> c0793101 T create_proc_profile
> $ cat /proc/kallsyms | grep create_proc_read_entry
> $
> ```
>
> А теперь посмотрим на файл определений <linux/proc_fs.h>:
>
> ```
> static inline struct proc_dir_entry *create_proc_read_entry(
>    const char *name, mode_t mode, struct proc_dir_entry *base,
>    read_proc_t *read_proc, void * data) {
> ...
> }
> ```

Возвращаемся к испытаниям второго полученного нами минимального модуля:

```
$ sudo insmod mod_procr2.ko
$ echo $?
0
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
```

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

В ядре старше 3.9 получаем:

```
$ sudo insmod mod_procr.ko
insmod: ERROR: could not insert module mod_procr.ko: Operation canceled
$ dmesg | tail -n1
[10526.036311] ! it's work only for kernel LE 3.9
```

Варианты реализации чтения

Примечание

Далее идет детализация, поэтому этот раздел можно пропустить без ущерба для остального содержания.

Уже было отмечено, что функция чтения `read_proc_t` претерпела в процессе своей эволюции множество изменений и на сегодня предполагает большое разнообразие ее использования. В подтверждение такого многообразия сошлемся на комментарий в коде *реализации* `read_proc_t` в ядре:

```
/*
 * Prototype:
 * int f(char *buffer, char **start, off_t offset,
 * int count, int *peof, void *dat)
 * Assume that the buffer is "count" bytes in size.
 * If you know you have supplied all the data you
 * have, set *peof.
 *
 * You have three ways to return data:
 * 0) Leave *start = NULL. (This is the default.)
 * Put the data of the requested offset at that
 * offset within the buffer. Return the number (n)
 * of bytes there are from the beginning of the
 * buffer up to the last byte of data. If the
 * number of supplied bytes (= n - offset) is
 * greater than zero and you didn't signal eof
 * and the reader is prepared to take more data
 * you will be called again with the requested
 * offset advanced by the number of bytes
 * absorbed. This interface is useful for files
 * no larger than the buffer.
 * 1) Set *start = an unsigned long value less than
 * the buffer address but greater than zero.
 * Put the data of the requested offset at the
 * beginning of the buffer. Return the number of
 * bytes of data placed there. If this number is
 * greater than zero and you didn't signal eof
 * and the reader is prepared to take more data
 * you will be called again with the requested
 * offset advanced by *start. This interface is
 * useful when you have a large file consisting
 * of a series of blocks which you want to count
 * and return as wholes.
 * (Hack by Paul.Russell@rustcorp.com.au)
 * 2) Set *start = an address within the buffer.
 * Put the data of the requested offset at *start.
 * Return the number of bytes of data placed there.
```

```
 * If this number is greater than zero and you
 * didn't signal eof and the reader is prepared to
 * take more data you will be called again with the
 * requested offset advanced by the number of bytes
 * absorbed.
 */
```

Даже в самом этом комментарии утверждается, что реализация выполнена уже на грани хакинга, она перегружена возможностями, и в литературе многократно отмечается ее неадекватное поведение при чтении данных, превышающих объемом 3072 байтов (пункты 1 и 2 комментария). Поэтому мы рассмотрим лишь вариант (пункт 0 в комментарии) чтения только данных, не превышающих страницу памяти. Здесь тоже могут быть различные реализации (они приведены как варианты в дополнительной папке procfs/_old_vers/2.6.42/variants сопровождающего книгу файлового архива), и далее показаны варианты их исполнения. Для каждого сравнительного варианта реализации создается свой модуль (вида mod_proc_*), который создает в /proc индивидуальное имя (вида /proc/mod_node_*):

```
$ ./insm
Module                  Size  Used by
mod_proc_3              1662  0
mod_proc_2              1638  0
mod_proc_1              1642  0
mod_proc_0              1610  0
vfat                    6740  1
/proc/mod_node_0  /proc/mod_node_1  /proc/mod_node_2  /proc/mod_node_3
```

Далее мы сравним варианты реализации — показаны будут варианты исполнения кода и кратко протокол выполнения (число вызовов и объемы копируемых данных на каждом вызове). Запрос чтения (в цикле по 20 байтов) во всех случаях одинаков и осуществляется программой `mcat` — вызов ее показан в листинге для первого варианта.

Итак, копирование полного объема на любой вызов — как обсуждалось ранее (6 операций, 6 копирований по 61 байту на каждой операции):

```
static ssize_t proc_node_read(char *buffer, char **start, off_t off,
                              int count, int *eof, void *data) {
   unsigned long len = 0;
   LOG("read: %d (buffer=%p, off=%ld, start=%p)", count, buffer, off, *start);
  // тупо копируем весь буфер, пока return не станет <= off
   memcpy(buffer, buf_msg, length);
   LOG("copy bytes: %ld", length);
   len = length;
   LOG("return bytes: %ld%s", len, *eof != 0 ? " ... EOF" : "");
   return len;
};
$ ./mcat 20 /proc/mod_node_0
...
```

```
$ dmesg | tail -n30 | grep !
! read: 20 (buffer=f30ab000, off=0, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f30ab000, off=20, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f30ab000, off=40, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f30ab000, off=60, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 19 (buffer=f30ab000, off=61, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f30ab000, off=61, start=(null))
! copy bytes: 61
! return bytes: 61
```

То же самое, но принудительно устанавливается признак конца файла (исчерпания данных), запросов данных становится на единицу меньше (5 операций, 4 копирования по 61 байту на каждой операции):

```
static ssize_t proc_node_read(char *buffer, char **start, off_t off,
                              int count, int *eof, void *data) {
   unsigned long len = 0;
   LOG("read: %d (buffer=%p, off=%ld, start=%p)", count, buffer, off, *start);
   // ... на 1 шаг раньше (return == off) установлен eof
   if(off < length) {
      memcpy(buffer, buf_msg, length);
      LOG("copy bytes: %d", length);
      len = length;
   }
   *eof = off + count >= length ? 1 : 0;
   LOG("return bytes: %ld%s", len, *eof != 0 ? " ... EOF" : "");
   return len;
};
$ dmesg | tail -n14 | grep !
! read: 20 (buffer=f67e8000, off=0, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f67e8000, off=20, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=f67e8000, off=40, start=(null))
! copy bytes: 61
! return bytes: 61
```

```
! read: 20 (buffer=f67e8000, off=60, start=(null))
! copy bytes: 61
! return bytes: 61 ... EOF
! read: 20 (buffer=f67e8000, off=61, start=(null))
! return bytes: 0 ... EOF
```

Но можно осуществлять только одно полное копирование на самом первом вызове, а при всех последующих вызовах использовать промежуточный буфер ядра (5 операций, 1 копирование по 61 байту на первой операции). Предполагается, что между вызовами `read()` данные пространства ядра остаются неизменными, но это неявно предполагается и во всех прочих вариантах:

```
static ssize_t proc_node_read(char *buffer, char **start, off_t off,
                              int count, int *eof, void *data) {
   unsigned long len = 0;
   LOG("read: %d (buffer=%p, off=%ld, start=%p)", count, buffer, off, *start);
   // 1 копирование всего буфера на 1-м запросе
   len = length;
   if(0 == off) {
      memcpy(buffer, buf_msg, length);
      LOG("copy bytes: %d", length);
   }
   *eof = off + count >= length ? 1 : 0;
   LOG("return bytes: %ld%s", len, *eof != 0 ? " ... EOF" : "");
   return len;
};
$ dmesg | tail -n11 | grep !
! read: 20 (buffer=c4775000, off=0, start=(null))
! copy bytes: 61
! return bytes: 61
! read: 20 (buffer=c4775000, off=20, start=(null))
! return bytes: 61
! read: 20 (buffer=c4775000, off=40, start=(null))
! return bytes: 61
! read: 20 (buffer=c4775000, off=60, start=(null))
! return bytes: 61 ... EOF
! read: 20 (buffer=c4775000, off=61, start=(null))
! return bytes: 61 ... EOF
```

Копирование ровно запрошенной длины при каждом вызове (5 операций, 4 копирования по 20 байтов на каждой операции):

```
static ssize_t proc_node_read(char *buffer, char **start, off_t off,
                              int count, int *eof, void *data) {
   unsigned long len = 0;
   LOG("read: %d (buffer=%p, off=%ld, start=%p)", count, buffer, off, *start);
   // копирование только count байт на каждом запросе - аккуратно меняем return!
   if(off >= length) {
      *eof = 1;
   }
```

```
   else {
      int cp;
      len = length - off;
      cp = min(count, len);
      memcpy(buffer + off, buf_msg + off, cp);
      LOG("copy bytes: %d", cp);
      len = off + cp;
      *eof = off + cp >= length ? 1 : 0;
   }
   LOG("return bytes: %ld%s", len, *eof != 0 ? " ... EOF" : "");
   return len;
};
$ dmesg | tail -n14 | grep !
! read: 20 (buffer=e5633000, off=0, start=(null))
! copy bytes: 20
! return bytes: 20
! read: 20 (buffer=e5633000, off=20, start=(null))
! copy bytes: 20
! return bytes: 40
! read: 20 (buffer=e5633000, off=40, start=(null))
! copy bytes: 20
! return bytes: 60
! read: 20 (buffer=e5633000, off=60, start=(null))
! copy bytes: 1
! return bytes: 61 ... EOF
! read: 20 (buffer=e5633000, off=61, start=(null))
! return bytes: 0 ... EOF
```

Как можно видеть, даже для таких простейших целей у реализатора имеется обширный простор для выбора. Но все варианты различаются только по показателям производительности. Учитывая, что через имена /proc обычно читаются весьма ограниченные объемы данных — главным образом диагностических, варианты становятся эквивалентными, и предпочтение, возможно, следует отдавать из соображений простоты (простое копирование).

Запись данных

С записью данных все гораздо проще. Операция записи не имеет возможности позиционирования и осуществляется по принципу «всё за раз» копированием из пространства пользователя в пространство ядра (см. ту же дополнительную папку procfs/_old_vers/2.6.42/variants сопровождающего книгу файлового архива):

proc_node_write.c

```
static int parameter = 999;

static ssize_t proc_node_write(struct file *file, const char __user *buffer,
                               unsigned long count, void *data) {
```

```
   unsigned long len = min(count, size);
   LOG("write: %ld (buffer=%p)", count, buffer);
   memset(buf_msg, NULL_CHAR, length);        // обнуление прежнего содержимого
   length = len;
   if(copy_from_user(buf_msg, buffer, len))
      return -EFAULT;
   sscanf(buf_msg, "%d", &parameter);
   LOG("parameter set to %d", parameter);
   return count;
}
```

Общий механизм файловых операций

Те же операции (чтение, запись) можно осуществлять с именами в /proc, используя традиционный механизм работы со всеми именами файловой системы. В следующем примере показан модуль, который создает имя в /proc. Оно может и читаться, и записываться — при этом используется структура указателей файловых операций в таблице операций (аналогично тому, как это делалось в драйверах интерфейса /dev):

mod_proc.c

```
#include "mod_proc.h"
#include "fops_rw.c"         // чтение-запись для /proc/mod_node

static const struct file_operations node_fops = {
   .owner = THIS_MODULE,
   .read  = node_read,
   .write  = node_write
};

static int __init proc_init(void) {
   int ret;
   struct proc_dir_entry *own_proc_node;
   own_proc_node = create_proc_entry(NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO, NULL);
   if(NULL == own_proc_node) {
      ret = -ENOENT;
      ERR("can't create /proc/%s", NAME_NODE);
      goto err_node;
   }
   own_proc_node->uid = own_proc_node->gid = 0;
   own_proc_node->proc_fops = &node_fops;
   LOG("/proc/%s installed", NAME_NODE);
   return 0;
err_node:
   return ret;
}
```

Здесь все знакомо и напоминает предыдущие примеры, за исключением заполнения таблицы файловых операций node_fops. Реализующие функции, как и ранее, вынесены в отдельный файл (они используются не один раз):

fops_rw.c

```
static char *get_rw_buf(void) {
   static char buf_msg[ LEN_MSG + 1 ] =
          ".........1.........2.........3.........4.........5\n";
   return buf_msg;
}

// чтение из /proc/mod_proc :
static ssize_t node_read(struct file *file, char *buf,
                         size_t count, loff_t *ppos) {
   char *buf_msg = get_rw_buf();
   int res;
   LOG("read: %d bytes (ppos=%lld)", count, *ppos);
   if(*ppos >= strlen(buf_msg)) {     // EOF
      *ppos = 0;
      LOG("EOF");
      return 0;
   }
   if(count > strlen(buf_msg) - *ppos)
      count = strlen(buf_msg) - *ppos;  // это копия
   res = copy_to_user((void*)buf, buf_msg + *ppos, count);
   *ppos += count;
   LOG("return %d bytes", count);
   return count;
}

// запись в /proc/mod_proc :
static ssize_t node_write(struct file *file, const char *buf,
                          size_t count, loff_t *ppos) {
   char *buf_msg = get_rw_buf();
   int res, len = count < LEN_MSG ? count : LEN_MSG;
   LOG("write: %d bytes", count);
   res = copy_from_user(buf_msg, (void*)buf, len);
   buf_msg[ len ] = '\0';
   LOG("put %d bytes", len);
   return len;
}
```

В отличие от упрощенных реализаций, с которыми мы работали с /dev, здесь показана более «зрелая» реализация операции чтения, отслеживающая позиционирование указателя чтения и допускающая произвольную длину данных в запросе read(). Обратите внимание на то, что функция чтения node_read() в этом примере *принципи-*

ально отличается от функции аналогичного назначения `proc_node_read()` в предыдущих примерах — не только своей реализацией, но и прототипом вызова: тем, что она непосредственно работает (копирует) с данными пространства пользователя, и тем, как она возвращает свои результаты.

Повторяем испытательный цикл того, что у нас получилось:

```
$ sudo insmod mod_proc.ko
$ ls -l /proc/mod_*
-rw-rw-rw- 1 root root 0 Июл  2 20:47 /proc/mod_node
$ cat /proc/mod_dir/mod_node
.........1.........2.........3.........4.........5
$ dmesg | tail -n30 | grep -v ^audit
! /proc/mod_node installed
! read: 32768 bytes (ppos=0)
! return 51 bytes
! read: 32768 bytes (ppos=51)
! EOF
$ echo new string > /proc/mod_dir/mod_node
$ cat /proc/mod_dir/mod_node
new string
$ ./mcat 3
read + 03 bytes, input buffer: new
read + 03 bytes, input buffer: new st
read + 03 bytes, input buffer: new strin
read + 02 bytes, input buffer: new string
read + 00 bytes, input buffer: new string
$ sudo rmmod mod_proc
$ cat /proc/mod_node
cat: /proc/mod_node: Нет такого файла или каталога
```

Начальное содержимое буфера модуля сделано отличающимся от предыдущих случаев как по содержанию, так и по длине (51 байт вместо 61) — чтобы визуально легко различать, какой из методов работает. Хорошо видно, как изменилась длина запрашиваемых данных утилитой `cat` (32 767). Детальный анализ происходящего:

```
$ ./mcat 20
read + 20 bytes, input buffer: .........1.........2
read + 20 bytes, input buffer: .........1.........2.........3.........4
read + 11 bytes, input buffer: .........1.........2.........3.........4.........5
read + 00 bytes, input buffer: .........1.........2.........3.........4.........5
$ dmesg | tail -n30 | grep -v ^audit
! /proc/mod_node installed
! read: 20 bytes (ppos=0)
! return 20 bytes
! read: 20 bytes (ppos=20)
! return 20 bytes
! read: 20 bytes (ppos=40)
! return 11 bytes
```

```
! read: 20 bytes (ppos=51)
! EOF
$ ./mcat 80
read + 51 bytes, input buffer: .........1.........2.........3.........4.........5
read + 00 bytes, input buffer: .........1.........2.........3.........4.........5
$ dmesg | tail -n5 | grep -v ^audit
! EOF
! read: 80 bytes (ppos=0)
! return 51 bytes
! read: 80 bytes (ppos=51)
! EOF
$ ./mcat 5
read + 05 bytes, input buffer: .....
read + 05 bytes, input buffer: .........1
read + 05 bytes, input buffer: .........1.....
read + 05 bytes, input buffer: .........1.........2
read + 05 bytes, input buffer: .........1.........2.....
read + 05 bytes, input buffer: .........1.........2.........3
read + 05 bytes, input buffer: .........1.........2.........3.....
read + 05 bytes, input buffer: .........1.........2.........3.........4
read + 05 bytes, input buffer: .........1.........2.........3.........4.....
read + 05 bytes, input buffer: .........1.........2.........3.........4.........5
read + 01 bytes, input buffer: .........1.........2.........3.........4.........5
read + 00 bytes, input buffer: .........1.........2.........3.........4.........5
```

Теперь мы получили возможность не только считывать диагностику со своего модуля, но и передавать ему управляющие воздействия, записывая в модуль новые значения. Еще раз обратите внимание на размер блока запроса на чтение (в системном журнале) и сравните с предыдущими случаями.

Ну а если нам захочется создать в /proc не отдельное имя, а собственную развитую иерархию имен? Как мы наблюдаем это, например, для любого системного каталога:

```
$ tree /proc/driver
/proc/driver
├── nvram
├── rtc
└── snd-page-alloc
0 directories, 3 files
```

Пожалуйста! Для этого придется только слегка расширить функцию инициализации предыдущего модуля (ну и привести ему в соответствие функцию выгрузки). Таким путем, по образу и подобию, вы можете создавать иерархию произвольной сложности и любой глубины вложенности:

mod_proct.c

```
#include "mod_proc.h"
#include "fops_rw.c" // чтение-запись для /proc/mod_dir/mod_node
```

```
static const struct file_operations node_fops = {
   .owner = THIS_MODULE,
   .read  = node_read,
   .write  = node_write
};

static struct proc_dir_entry *own_proc_dir;

static int __init proc_init(void) {
   int ret;
   struct proc_dir_entry *own_proc_node;
   own_proc_dir = create_proc_entry(NAME_DIR, S_IFDIR | S_IRWXUGO, NULL);
   if(NULL == own_proc_dir) {
      ret = -ENOENT;
      ERR("can't create directory /proc/%s", NAME_DIR);
      goto err_dir;
   }
   own_proc_dir->uid = own_proc_dir->gid = 0;
   own_proc_node = create_proc_entry(NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO, own_proc_dir);
   if(NULL == own_proc_node) {
      ret = -ENOENT;
      ERR("can't create node /proc/%s\n", NAME_NODE);
      goto err_node;
   }
   own_proc_node->uid = own_proc_node->gid = 0;
   own_proc_node->proc_fops = &node_fops;
   LOG("/proc/%s installed", NAME_NODE);
   return 0;
err_node:
   remove_proc_entry(NAME_DIR, NULL);
err_dir:
   return ret;
}

static void __exit proc_exit(void) {
   remove_proc_entry(NAME_NODE, own_proc_dir);
   remove_proc_entry(NAME_DIR, NULL);
}
```

Примечание

Здесь любопытно обратить внимание, с какой легкостью имя в /proc создается то как каталог, то как терминальное имя (файл) — в зависимости от выбора единственного бита во флагах создания: `S_IFDIR` или `S_IFREG`.

Эксперименты с mod_proct.ko аналогичны предыдущим, но с отличающимся путевым именем, которое создает модуль в /proc:

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

Таким образом, показано, что создание структур имен из модулей позволяет как получать диагностическую информацию из модуля, так и передавать управляющую информацию в модуль. С момента получения таких возможностей модуль становится управляемым (и это зачастую замещает необходимость в операциях `ioctl`, которые очень плохо контролируемы с точки зрения возможных ошибок).

API ядра предоставляет два альтернативных набора функций для реализации ввода/вывода. Возникает последний и закономерный вопрос: какая функция из этих двух альтернатив будет отрабатываться, если определены обе? Кто из методов обладает приоритетом? Для ответа был сделан модуль mod_2.c (он является линейной комбинацией показанного здесь и не приводится, но включен в архив примеров к тексту — см. папки proc в вариантах папки procfs/_old_vers). Вот что показывает его использование:

- определена таблица файловых операций:

```
$ sudo insmod mod_2.ko mode=1
$ cat /proc/mod_node
.........1.........2.........3.........4.........5
$ dmesg | tail -n30 | grep -v ^audit
! /proc/mod_node installed
! read: 32768 bytes
! return 51 bytes
! read: 32768 bytes
! EOF
$ sudo rmmod mod_2
```

- таблица файловых операций не определялась, производится чтение функцией чтения /proc (отличается строка вывода):

```
$ sudo insmod mod_2.ko mode=2
$ cat /proc/mod_node
.........1.........2.........3.........4.........5.........6
$ dmesg | tail -n30 | grep -v ^audit
! /proc/mod_node installed
! read: 3072 (buffer=f1629000, off=0)
! return bytes: 61  ... EOF
! read: 3072 (buffer=f1629000, off=61)
```

```
! return bytes: 0  ... EOF
$ sudo rmmod mod_2
```

- определены *одновременно* и таблица файловых операций, и функция чтения /proc, операция выполняется через таблицу файловых операций:

```
$ sudo insmod mod_2.ko mode=3
$ cat /proc/mod_node
.........1.........2.........3.........4.........5
$ dmesg | tail -n30 | grep -v ^audit
! /proc/mod_node installed
! read: 32768 bytes
! return 51 bytes
! read: 32768 bytes
! EOF
```

Интерфейс /sys

Одно из главных «приобретений» ядра, начинающееся от версий 2.6, — это появление в Linux единой унифицированной модели представления устройств. Главные составляющие, сделавшие возможным ее 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;
- задействуется механизм для выполнения общих, часто встречающихся функций — таких как счетчики использования;
- появляется 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 (т. е. в обеспечение последнего пункта приведенного списка). Но позже модель оказалась гораздо плодотворнее и используется заметно шире.

Уже описывалось ранее, как при загрузке (выгрузке) любого модуля, горячем подключении (отключении) любого физического устройства и других значительных событиях ядром рассылаются асинхронные уведомления через широковещатель-

ный дейтаграммный сокет протокола netlink. Эти широковещательные уведомления могут быть получены *любым* приложением *пользовательского* пространства, которое может осуществлять какую-либо реакцию в файловой системе на такие динамические события. Ранее было показано, как увидеть содержимое таких уведомлений для их анализа и отладки реакций на них. Одну из таких подсистем (а их может быть одновременно много), осуществляющую реакции на асинхронные уведомления ядра, и реализует демон udevd, который отображает эти изменения в файловой системе sysfs:

```
$ ps -A | grep udev
    735 ?        00:00:02 systemd-udevd
```

Реакция udev на уведомления о событиях ядра определяется *правилами* (текстовыми файлами .rules) — вот их примеры:

```
$ ls -w80 /lib/udev/rules.d/
39-usbmuxd.rules                    73-special-net-names.rules
40-usb-media-players.rules          75-net-description.rules
40-usb_modeswitch.rules             75-probe_mtd.rules
40-vm-hotadd.rules                  77-mm-broadmobi-port-types.rules
50-firmware.rules                   77-mm-cinterion-port-types.rules
...
70-touchpad.rules                   95-upower-hid.rules
70-u2f.rules                        95-upower-wup.rules
70-uaccess.rules                    96-e2scrub.rules
71-power-switch-proliant.rules      97-dmraid.rules
71-seat.rules                       97-hid2hci.rules
71-u-d-c-gpu-detection.rules        99-systemd.rules
73-seat-late.rules
$ ls -w100 /etc/ufw/*.rules
/etc/ufw/after6.rules  /etc/ufw/before6.rules  /etc/ufw/user6.rules
/etc/ufw/after.rules   /etc/ufw/before.rules   /etc/ufw/user.rules
```

В файлах описываются действия при возникновении конкретных событий: создание имен или иерархии имен в /dev или /sys, запуск каких-то утилит, приложений... Эти действия формулируются в специфическом, хорошо описанном синтаксисе, например:

```
$ cat 70-mouse.rules
# do not edit this file, it will be overwritten on update

ACTION=="remove", GOTO="mouse_end"
KERNEL!="event*", GOTO="mouse_end"
ENV{ID_INPUT_MOUSE}=="", GOTO="mouse_end"

# mouse:<subsystem>:v<vid>p<pid>:name:<name>:*
KERNELS=="input*", ENV{ID_BUS}=="usb", \
        IMPORT{builtin}="hwdb 'mouse:$env{ID_BUS}:v$attr{id/vendor}p$attr{id/
                                                        product}:name:$attr{name}:'", \
        GOTO="mouse_end"
```

```
KERNELS=="input*", ENV{ID_BUS}=="bluetooth", \
        IMPORT{builtin}="hwdb
'mouse:$env{ID_BUS}:v$attr{id/vendor}p$attr{id/product}:name:$attr{name}:'", \
        GOTO="mouse_end"
DRIVERS=="psmouse", SUBSYSTEMS=="serio", \
        IMPORT{builtin}="hwdb 'mouse:ps2::name:$attr{device/name}:'", \
        GOTO="mouse_end"

LABEL="mouse_end"
```

В этом *автоматизме* выполняемых реакций и состоит главное предназначение /sys. Тем не менее в ряде проприетарных проектов возникает потребность в *ручном* создании (иерархии) имен для диагностики состояний или управления и контроля подобно тому, как это делается в /proc.

Создание и использование имен в */sys*

Сама по себе эта система весьма сложная и объемная, и о ней одной можно и нужно писать отдельную книгу. Но в контексте нашего рассмотрения нас интересует, кроме упомянутого автоматизма, возможность создания в файловой системе /sys интерфейса из модуля к файловым именам. Эта возможность весьма напоминает то, как модуль создает файловые имена в подсистеме /proc.

Базовым понятием модели представления устройств являются объекты типа `struct kobject` (определяется в файле <linux/kobject.h>). Тип `struct kobject` по смыслу аналогичен абстрактному базовому классу `Object` в объектно-ориентированных языках программирования — таких как C# и Java. Этот тип определяет общую функциональность — такую как счетчик ссылок, имя, указатель на родительский объект, — что позволяет создавать объектную иерархию.

Зачастую объекты `struct kobject` сами по себе не создаются и не используются, они встраиваются в другие структуры данных, после чего те приобретают свойства, присущие `struct kobject`, — например, такие как встраиваемость в иерархию объектов. Вот как это выражается в определении уже известной нам структуры представления символьного устройства:

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

Во внешнем представлении в каталоге /sys в интересующем нас смысле каждому объекту `struct kobject` соответствует *каталог*, что видно и из самого определения структуры (показано для ядра 5.4):

```
struct kobject {
        const char              *name;
        struct list_head        entry;
```

```
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct kernfs_node      *sd; /* sysfs directory entry */
...
};
```

Но это вовсе не означает, что каждый инициализированный объект `struct kobject` автоматически экспортируется в файловую систему /sys. Для того чтобы объект сделать видимым в /sys, необходимо вызвать:

```
int kobject_add(struct kobject *kobj, struct kobject *parent,
                const char *fmt, ...);
```

Но это не придется нам делать явно в приведенных далее примерах по той простой причине, что используемые для регистрации имен в /sys высокоуровневые вызовы API (`class_create()`) делают это за нас.

Таким образом, объекты `struct kobject` естественным образом отображаются в *каталоги* пространства имен /sys, которые увязываются в иерархии. Файловая система sysfs — это дерево каталогов без файлов. А как создать файлы в этих каталогах, в содержимое которых отображаются данные ядра? Каждый объект `struct kobject` (каталог) содержит (через свой компонент `struct kobj_type`) массив (указателей) структур `struct attribute`:

```
struct kobj_type {
        void (*release)(struct kobject *kobj);
        const struct sysfs_ops *sysfs_ops;
        struct attribute **default_attrs;       /* use default_groups instead */
        const struct attribute_group **default_groups;
        const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
        const void *(*namespace)(struct kobject *kobj);
        void (*get_ownership)(struct kobject *kobj, kuid_t *uid, kgid_t *gid);
};
```

А вот каждая такая структура `struct attribute` (определена в <linux/sysfs.h>) и является определением одного *файлового имени*, содержащегося в рассматриваемом каталоге:

```
struct attribute {
        const char              *name; /* имя атрибута-файла */
        umode_t                 mode;  /* права доступа к файлу */
...
};
typedef unsigned short          umode_t;
```

Показанный там же (`struct kobj_type`) *массив структур* таблиц операций (`struct sysfs_ops` — определение также в <linux/sysfs.h>) содержит два поля: определения функций `show()` и `store()` — соответственно чтения и записи символьного поля данных ядра, отображаемых этим файлом:

```
struct sysfs_ops {
   ssize_t (*show)(struct kobject*, struct attribute*, char *buf);
   ssize_t (*store)(struct kobject*, struct attribute*, const char *buf, size_t count);
...
}
```

И сами эти функции и их использование показаны в приведенных далее примерах. Их прототипы весьма часто модифицируются от версии к версии ядра (у нас показано для 3.10), поэтому следует обязательно справиться о них в их текущих определениях — это сэкономит много времени.

> ***Пояснение***
>
> Показанное поле `sysfs_ops` в составе `struct kobj_type` — это именно массив структур, хотя это и не очевидно из описаний. Таким образом, с каждым атрибутом (файловым именем) может быть связан свой набор функций чтения и записи этого файлового имени извне.

Этих сведений о sysfs нам должно быть достаточно для создания интерфейса модуля в пространстве имен /sys, но перед тем, как переходить к примеру, остановимся и приведем краткое уточнение по части аналогий и различий /proc и /sys, выступающих в качестве интерфейсов для отображения модулем подконтрольных ему данных ядра. Различия систем /proc и /sys складываются главным образом на основе негласных соглашений и устоявшихся традиций:

- информация терминальных имен /proc — комплексная и обычно содержит большие объемы текстовой информации: иногда это таблицы, и даже с заголовками, проясняющими смысл столбцов таблицы;
- информацию терминальных имен /sys (атрибутов) рекомендуется оформлять в виде: а) простых; б) символьных значений; в) представляющих величины, 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 со сформированным заголовком таблицы, разъясняющим смысл колонок, а во втором — представление целочисленных значений.

А теперь мы готовы перейти к рассмотрению возможного вида модуля (см. папку sysfs в сопровождающем книгу файловом архиве), читающего из атрибута-имени, созданного им в /sys, и пишущего в него (большая часть происходящего в этом модуле, за исключением регистрации имен в /sys, нам уже известна):

xxx.c

```
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/parport.h>
#include <asm/uaccess.h>
#include <linux/pci.h>
#include <linux/version.h>

#define LEN_MSG 160
static char buf_msg[ LEN_MSG + 1 ] = "Hello from module!\n";

/* <linux/device.h>
LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
struct class_attribute {
   struct attribute attr;
   ssize_t (*show)(struct class *class, struct class_attribute *attr, char *buf);
   ssize_t (*store)(struct class *class, struct class_attribute *attr,
                    const char *buf, size_t count);
};
LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,32)
struct class_attribute {
   struct attribute attr;
   ssize_t (*show)(struct class *class, char *buf);
   ssize_t (*store)(struct class *class, const char *buf, size_t count);
};
*/

/* sysfs show() method. Calls the show() method corresponding to the individual sysfs file */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t x_show(struct class *class, struct class_attribute *attr, char *buf) {
#else
static ssize_t x_show(struct class *class, char *buf) {
#endif
   strcpy(buf, buf_msg);
   printk("read %d\n", strlen(buf));
   return strlen(buf);
}

/* sysfs store() method. Calls the store() method corresponding to the individual sysfs file */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t x_store(struct class *class, struct class_attribute *attr,
                       const char *buf, size_t count) {
#else
static ssize_t x_store(struct class *class, const char *buf, size_t count) {
#endif
   printk("write %d\n" , count);
```

```
   strncpy(buf_msg, buf, count);
   buf_msg[ count ] = '\0';
   return count;
}

/* <linux/device.h>
#define CLASS_ATTR(_name, _mode, _show, _store) \
struct class_attribute class_attr_##_name = __ATTR(_name, _mode, _show, _store) */
CLASS_ATTR(xxx, 0666, &x_show, &x_store);

static struct class *x_class;

int __init x_init(void) {
   int res;
   x_class = class_create(THIS_MODULE, "x-class");
   if(IS_ERR(x_class)) printk("bad class create\n");
   res = class_create_file(x_class, &class_attr_xxx);
/* <linux/device.h>
extern int __must_check class_create_file(struct class *class, const struct class_attribute
*attr); */
   printk("'xxx' module initialized\n");
   return 0;
}

void x_cleanup(void) {
/* <linux/device.h>
extern void class_remove_file(struct class *class, const struct class_attribute *attr); */
   class_remove_file(x_class, &class_attr_xxx);
   class_destroy(x_class);
   return;
}

module_init(x_init);
module_exit(x_cleanup);
MODULE_LICENSE("GPL");
```

В коде показанного модуля:

- создается класс `struct class` (соответствующая ему структура `struct kobject`), отображаемый вызовом `class_create()` в создаваемый *каталог* с именем `x-class`;
- создается атрибутная запись `struct class_attribute` с именем *переменной* `class_attr_xxx`. Эта переменная создается макросом `CLASS_ATTR()`, поэтому имя переменной образуется (текстовой конкатенацией) из параметра макровызова. Такое может казаться необычным без привычки, но все, что делается вокруг **/sys**, — выполняется в подобной технике (этот и ряд подобных макросов);
- тот же макровызов определяет и имена (адреса) функций обработчиков `show()` и `store()` для этого атрибута;

- созданная атрибутная запись увязывается с ранее созданным классом вызовом `class_create_file()`. Именно на этом этапе будет создано *файловое* имя в каталоге `x-class`. Само имя, под которым будет представляться файл, определилось на предыдущем шаге в вызове `CLASS_ATTR()`;
- функция-обработчик `show()` будет вызываться при выполнении запросов на чтение файлового имени /sys/class/x-class/xxx, а `store()` — соответственно на запись.

Обычной практикой является запись целой последовательности макровызовов `CLASS_ATTR()`, определяющих несколько атрибутных записей, которые позже последовательностью вызовов `class_create_file()` увяжутся с единым классом (создается целая группа файлов в одном каталоге).

При таком групповом создании **имена** обрабатывающих функций (третий и четвертый параметры макровызовов `CLASS_ATTR()`) также формируются как конкатенация параметров вызова. Другой практикой уточнения, к какому атрибуту относится запрос, является динамический анализ параметра `attr`, полученного функциями `show()` и `store()` (только в относительно поздних версиях ядра — после 2.6.32).

Особенностями кода, работающего с подсистемой /sys, является высокая волатильность всего API, доступного для использования в коде: прототипы функций, структуры данных, макроопределения и все прочее. Это подчеркивается тем, что комментарии относительно различий версий включены в текст приведенного примера, чтобы определить наиболее «версиеопасные направления» (определения взяты из хедер-файлов).

Теперь мы готовы рассмотреть работу кода:

```
$ sudo insmod xxx.ko
$ lsmod | head -n2
Module                  Size  Used by
xxx                     1047  0
$ ls -lR /sys/class/x-class
/sys/class/x-class:
итого 0
-rw-rw-rw- 1 root root 4096 Янв 27 23:34 xxx
$ tree /sys/class/x-class
/sys/class/x-class
└── xxx
0 directories, 1 file
$ ls -l /sys/module/xxx/
итого 0
drwxr-xr-x 2 root root    0 Янв 27 23:57 holders
-r--r--r-- 1 root root 4096 Янв 27 23:57 initstate
drwxr-xr-x 2 root root    0 Янв 27 23:57 notes
-r--r--r-- 1 root root 4096 Янв 27 23:57 refcnt
drwxr-xr-x 2 root root    0 Янв 27 23:57 sections
-r--r--r-- 1 root root 4096 Янв 27 23:57 srcversion
$ dmesg | tail -n18 | grep -v ^audit
'xxx' module initialized
```

```
$ cat /sys/class/x-class/xxx
Hello from module!
$ dmesg | tail -n15 | grep -v ^audit
read 19
```

К этому месту мы убеждаемся (по форме вывода), что операция чтения со стороны пользователя действительно выполняется функцией `show()` и названия функций `show()` и `store()` отражают направления передачи данных именно со стороны внешнего наблюдателя (из пространства пользователя). Для разработчика кода модуля они носят в точности противоположный смысл. Смотрим дальнейшие операции:

```
$ echo это новое содержимое > /sys/class/x-class/xxx
$ cat /sys/class/x-class/xxx
это новое содержимое
$ dmesg | tail -n10 | grep -v ^audit
write 39
read 39
$ sudo rmmod xxx
```

В тех случаях, а это, как правило, когда стремятся создать целую группу файловых имен, связанных с модулем, поведение которых обычно сходно и различается лишь деталями (например, привязкой к *различным* переменным внутри модуля), чаще всего используют параметризуемые макроопределения для определения функций `show()` и `store()`. Читать и отлаживать такое практически невозможно, но это работает и массово используется. Для конкретизации сказанного перепишем предыдущий модуль так, чтобы он создавал три независимые точки входа (показаны только отличия от xxx.c):

xxm.c

```
...
#define LEN_MSG 160

// определения функций обработчиков
#define IOFUNCS(name)                                                    \
static char buf_##name[ LEN_MSG + 1 ] = "не инициализировано "#name"\n";  \
static ssize_t SHOW_##name(struct class *class, struct class_attribute *attr,  \
                           char *buf) {                                  \
   strcpy(buf, buf_##name);                                              \
   printk("read %d\n", strlen(buf));                                    \
   return strlen(buf);                                                   \
}                                                                         \
static ssize_t STORE_##name(struct class *class, struct class_attribute *attr, \
                            const char *buf, size_t count);              \
   printk("write %d\n", count);                                          \
   strncpy(buf_##name, buf, count);                                      \
   buf_##name[ count ] = '\0';                                           \
   return count;                                                         \
}
```

```
IOFUNCS(data1);
IOFUNCS(data2);
IOFUNCS(data3);

// определение атрибутных записей
#define OWN_CLASS_ATTR(name) \
struct class_attribute class_attr_##name = \
__ATTR(name, 0666, &SHOW_##name, &STORE_##name)
static OWN_CLASS_ATTR(data1); // создается class_attr_data1
static OWN_CLASS_ATTR(data2); // создается class_attr_data2
static OWN_CLASS_ATTR(data3); // создается class_attr_data3
...

int __init x_init(void) {
   ...
   res = class_create_file(x_class, &class_attr_data1);
   res = class_create_file(x_class, &class_attr_data2);
   res = class_create_file(x_class, &class_attr_data3);
   ...
}

void x_cleanup(void) {
   class_remove_file(x_class, &class_attr_data1);
   class_remove_file(x_class, &class_attr_data2);
   class_remove_file(x_class, &class_attr_data3);
   ...
}
```

В подобных конструкциях могут создаваться весьма обширные коллекции файловых имен — в обсуждаемом примере их три:

```
$ sudo insmod xxm.ko
$ tree /sys/class/x-class
/sys/class/x-class
├── data1
├── data2
└── data3
0 directories, 3 files
$ cat /sys/class/x-class/data1
не инициализировано data1
$ cat /sys/class/x-class/data2
не инициализировано data2
$ cat /sys/class/x-class/data3
не инициализировано data3
$ echo строка 1 > /sys/class/x-class/data1
$ echo строка 2 > /sys/class/x-class/data2
$ echo строка 3 > /sys/class/x-class/data3
```

```
$ cat /sys/class/x-class/data1
строка 1
$ cat /sys/class/x-class/data2
строка 2
$ cat /sys/class/x-class/data3
строка 3
```

Часто задаваемый вопрос: а можно ли в /sys создать *иерархию* имен произвольной глубины и структуризации так, как это возможно сделать в /proc? В первом приближении ответ: *нет*. Потому что /sys — это каталог управления и отображения *подсистемы ввода/вывода* устройств со своей структурой, а /sys/class, в котором мы создаем имена для связи с модулем, — это каталог *классов* устройств.

Но если это окажется уж особо необходимым, мы можем уже внутри собственного класса устройств создать фиктивное устройство (каталог) по типу следующего (в продолжение предыдущих кодов):

```
ssize_t device_show_ulong(struct device *dev, struct device_attribute *attr, char *buf) {
... };
ssize_t device_store_ulong(struct device *dev, struct device_attribute *attr,
                           const char *buf, size_t count) {
...};
// #define DEVICE_ATTR(_name, _mode, _show, _store) \
//         struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
static OWN_DEVICE_ATTR(data, 0666, &device_show_ulong, &device_store_ulong);
   ...
   x_class = class_create(THIS_MODULE, "x-class");
   dev_t dev = MKDEV(10, 1); // произвольно, для примера
   struct device *pdev = device_create(x_class, NULL, dev, NULL, "zzz");
   int res = device_create_file(pdev, &dev_attr_data);
   ...
```

Все названные вызовы описаны в <linux/device.h> и, как легко заметить, все они подобны обсуждавшимся ранее с заменой префиксов `class_*` на `device_*`. При этом будет создано фиктивное устройство:

```
$ ls -l /dev/zzz
crw------- 1 root root 10, 1 май 16 20:18 /dev/zzz
```

А в /sys/class будет создан подкаталог (ссылка) x_class/zzz с характерной структурой устройства, но с дополнительным терминальным именем data. Но, конечно, подобная схема очень искусственна.

Ошибки обменных операций

А что, если по какой-то причине *созданное* путевое имя в /sys не может быть считано, когда пользователю нужно сообщить об ошибке операции ввода/вывода? Для этого в собственных функциях чтения/записи возвращаем *отрицательный* код ошибки (показаны только отличия от примера xxx.c):

xxe.c

```
...
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t x_show(struct class *class, struct class_attribute *attr, char *buf) {
#else
static ssize_t x_show(struct class *class, char *buf) {
#endif
   ...
   return -EIO;
}

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,32)
static ssize_t x_store(struct class *class, struct class_attribute *attr,
                       const char *buf, size_t count) {
#else
static ssize_t x_store(struct class *class, const char *buf, size_t count) {
#endif
   ...
   return -EIO;
}
...
```

Уточненные символьные коды ошибок черпаем из заголовочных файлов errno-base.h и errno.h каталога <uapi/asm-generic> (и еще нескольких с более специфичными кодами):

```
#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Argument list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
...
```

А вот как выглядит выполнение с ошибками с точки зрения пользователя:

```
$ sudo insmod xxe.ko
$ lsmod | head -n3
Module                  Size  Used by
xxe                    12546  0
fuse                   80271  3
$ pwd
/sys/class/x-class
$ cat xxx
cat: xxx: Ошибка ввода/вывода
```

```
$ echo 1 > xxx
bash: echo: ошибка записи: Ошибка ввода/вывода
```

На этом мы и остановимся в рассмотрении подсистемы /sys. Потому как сейчас функции /sys в Linux расширились настолько, что об одной этой файловой подсистеме можно и нужно писать отдельную книгу: все устройства в системе (сознательно стараниями автора модуля для него или даже помимо его воли) находят отображения в /sys, а сопутствующая ей подсистема пользовательского пространства udev динамически управляет правилами создания имен и полномочиями доступа к ним. Но это уже абсолютно другая история. Мы же на кратко описанном примере рассмотрели совершенно частную задачу: как из собственного модуля создать интерфейс к именам в /sys для создания диагностических или управляющих интерфейсов этого модуля.

Сетевые интерфейсы и протоколы

Сетевая подсистема гораздо более разветвленная, нежели интерфейс 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.

Существует еще один дополнительный мотив, согласно которому сетевая подсистема должна быть для программиста-разработчика особенно близка и интересна: в связи со взрывным расширением спектра протоколов и устройств коммуникаций разработка драйверов специфических (проприетарных) коммуникационных устройств попадает в задачи гораздо чаще, чем для любых других видов устройств, и предоставляет расширенные возможности для творчества...

Сетевая подсистема Linux ориентирована в большей степени на обслуживание протоколов Ethernet на канальном уровне и TCP/IP на уровне транспортном, но эта модель расширяется с равным успехом и на другие типы протоколов, покрывая, таким образом, всю сферу возможностей. На сегодня сетевая подсистема Linux обеспечивает поддержку широчайшего спектра протоколов и устройств:

- проводные соединения Ethernet (LAN);
- беспроводные соединения Wi-Fi (LAN) и Bluetooth (PAN);
- разнообразные проводные устройства «последней мили» (WAN): E1/T1/J1, различные модификации DSL, ...
- беспроводные модемы (WAN), относящиеся к разнообразным протоколам, сетям и модификациям: GSM, GPRS, EDGE, CDMA, EV-DO (EVDO), WiMAX, LTE, ...

Весь этот спектр (и еще некоторые классы менее употребляемых технологий) поддерживается *единой* сетевой подсистемой.

Сеть TCP/IP, как известно, очень условно согласуется (или вовсе не согласуется) с семиуровневой моделью OSI взаимодействия открытых систем (она и разработана раньше модели OSI, и, естественно, что они не соответствуют друг другу). В Linux сложилась такая терминология разделения на подуровни (см. [24]):

- все, что относится к поддержке оборудования и к канальному уровню, — описывается как сетевые *интерфейсы* и обозначается L2. Преимущественно это Ethernet, но и другие сетевые протоколы канального уровня (Token Ring, ArcNet, ...) тоже;
- протоколы сетевого уровня OSI (IP/IPv4/IPv6 , IPX, RIP, OSPF, ARP, ...) — как *сетевой* уровень стека протоколов, или уровень L3;
- все, что выше (ICMP, UDP, TCP, SCTP, ...), — как протоколы *транспортного* уровня, или уровень L4;
- все же то, что принадлежит к лежащим еще выше уровням (сеансовый, представительский, прикладной) модели OSI (например: SSH, SIP, RTP, ...), — никоим образом не проявляется в ядре и относится уже только к области клиентских и серверных утилит пространства пользователя.

Такая сложившаяся числовая нумерация сетевых слоев (layers: L2, L3, L4) соответствует *аналогиям* (не более) из модели OSI (обратите внимание, что в принятой в Linux терминологии нет слоя L1 — это физический уровень передачи данных, который не попадает в круг интересов разработчиков системы). Точно так же в круг интересов разработчиков *ядра* Linux не попадают все сетевые слои, которые лежат выше L4, — это уже протоколы и приложения пользовательского уровня (абстрак-

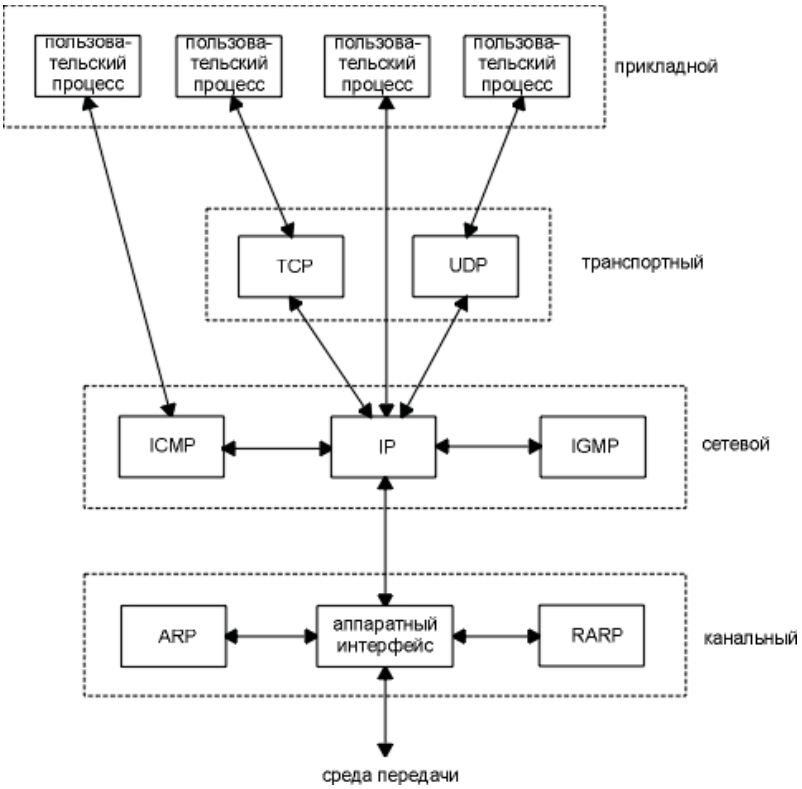


Рис. 4.3. Сетевой стек Linux (заимствовано из легендарной книги У. Р. Стивенса «TCP/IP Illustrated»)

ция сетевых сокетов), но именно там создаются и потребляются сетевые пакеты (рис. 4.3).

Сетевые инструменты

Но прежде, чем заняться созданием модулей сетевых интерфейсов, нам придется коротко восстановить в памяти те инструменты, с помощью которых мы будем наблюдать сетевые состояния в системе и управлять ими.

Сетевые интерфейсы

В отличие от всех прочих *устройств* в системе (символьных или блочных), которым соответствуют имена устройств в каталоге /dev, сетевые устройства создают сетевые *интерфейсы*, которые не отображаются как именованные устройства, но каждый из которых имеет набор своих характеристических параметров (MAC-адрес, IP-адрес, маска сети, шлюз, ...). Интерфейсы могут быть физическими (отображающими реальные аппаратные сетевые устройства, например `eth0` — адаптер Ethernet), или логическими, виртуальными (отражающими некоторые моделируемые понятия, например `tap0` — туннельный интерфейс).

Имена сетевых интерфейсов, как мы увидим вскоре, могут быть *произвольными* и определяются *модулем* ядра, реализующим интерфейс для устройств такого типа. В системе не может быть интерфейсов с совпадающими именами, поэтому поддерживающий модуль будет как-либо модифицировать имена интерфейсов однотипных устройств в соответствии с принятой схемой именований.

> ***Примечание***
>
> Существует предрассудок, что имена проводных устройств Ethernet — это, например, `eth0`, `eth1`, `eth2`, ..., а интерфейсы Wi-Fi — это соответственно `wlan0`, `wlan1`, `wlan2` и т. д. Это не так. Когда-то это соответствовало действительности, но в современном Linux имена интерфейсов могут быть произвольными. Более того, на одной и той же аппаратной конфигурации при установке различных версий даже одного и того же дистрибутива имена интерфейсов и схема именования могут различаться.

Посмотреть текущие существующие сетевые интерфейсы можно, например, так:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp2s14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group
                                                                        default qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wlp8s0: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN mode DORMANT group default
                                                                            qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
```

Этим же интерфейсам соответствуют подкаталоги с соответствующими именами в /proc, и каждый такой подкаталог содержит псевдофайлы-параметры (по диагностике или управлению) соответствующего интерфейса:

```
$ ls /proc/sys/net/ipv4/conf
all  default  enp2s14  lo  wlp8s0
$ ls -w80 /proc/sys/net/ipv4/conf/enp2s14/
accept_local          disable_xfrm                        proxy_arp_pvlan
accept_redirects      force_igmp_version                  route_localnet
accept_source_route   forwarding                          rp_filter
arp_accept            igmpv2_unsolicited_report_interval  secure_redirects
arp_announce          igmpv3_unsolicited_report_interval  send_redirects
arp_filter            log_martians                        shared_media
arp_ignore            mc_forwarding                       src_valid_mark
arp_notify            medium_id                           tag
bootp_relay           promote_secondaries
disable_policy        proxy_arp
```

Не менее важной, чем набор сетевых интерфейсов, характеристикой сетевой подсистемы является таблица роутинга ядра, которая полностью и однозначно определяет направления (между интерфейсами) распространения трафика, — например:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: em1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT
                                                                    group default qlen 1000
    link/ether a0:1d:48:f4:93:5c brd ff:ff:ff:ff:ff:ff
3: wlo1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DORMANT
                                                                    group default qlen 1000
    link/ether 34:23:87:d6:85:0d brd ff:ff:ff:ff:ff:ff
4: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN
                                                                mode DEFAULT group default
    link/ppp
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    1024   0        0 em1
80.255.73.34    0.0.0.0         255.255.255.255 UH    0      0        0 ppp0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 em1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 wlo1
```

Несоответствие таблицы роутинга состояниям сетевых интерфейсов (что весьма часто случается при экспериментах и отладке сетевых модулей ядра) — наиболее вероятная причина отличия поведения сети от ожидаемого (картина восстанавливается соответствующими командами `route`, добавляющими направления в таблицу или удаляющими их из нее). Самое краткое и *исчерпывающее* описание алгоритма работы TCP/IP-сети (из известных автору) дал У. Р. Стивенс:

- IP-пакеты (создающиеся на хосте или приходящие на него снаружи), если они не предназначены этому хосту, ретранслируются в соответствии с одной из строк таблицы роутинга на основе IP-адреса *получателя*;

- если ни одна строка таблицы не соответствует адресу получателя (подсеть или хост), то пакеты ретранслируются в интерфейс, обозначенный как интерфейс по умолчанию, который *всегда* присутствует в таблице роутинга (интерфейс с `Destination`, равным `0.0.0.0`, в только что показанном примере);
- пакет, пришедший с некоторого интерфейса, *никогда* не ретранслируется в этот же интерфейс.

По этому алгоритму всегда можно разобрать картину происходящего в системе с любой самой сложной конфигурацией интерфейсов.

Одному аппаратному сетевому устройству (физическому интерфейсу) может соответствовать один (наиболее частый случай) или несколько различных сетевых интерфейсов. Новые (логические) сетевые интерфейсы могут *надстраиваться* (соответствующими модулями ядра) над уже существующими (физическими или виртуальными). По тексту далее мы станем называть такие надстроенные интерфейсы *виртуальными* — построенными над другими, ранее существующими. Задача надстройки виртуальных сетевых интерфейсов зачастую и является самой распространенной задачей, стоящей перед практическим разработчиком. Простейшим примером множественности логических сетевых интерфейсов могут служить *сетевые алиасы*, когда существующему интерфейсу дополнительно присваиваются адрес и маска, делающие его представленным еще в одной подсети:

```
$ ifconfig
enp2s14: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.5  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::215:60ff:fec4:ee02  prefixlen 64  scopeid 0x20<link>
        ether 00:15:60:c4:ee:02  txqueuelen 1000  (Ethernet)
        RX packets 16943  bytes 22978143 (21.9 MiB)
        RX errors 0  dropped 1  overruns 0  frame 0
        TX packets 10437  bytes 851740 (831.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 16
...
$ route -n
Kernel IP routing table
Destination   Gateway       Genmask       Flags Metric Ref  Use Iface
0.0.0.0       192.168.1.1   0.0.0.0       UG    1024   0      0 enp2s14
192.168.1.0   0.0.0.0       255.255.255.0 U     0      0      0 enp2s14
```

К этому моменту у нас присутствует единственный Ethernet-интерфейс `enp2s14` в LAN `192.168.1.0/24` (он же интерфейс по умолчанию). Мы можем создать ему алиасный интерфейс:

```
$ sudo ifconfig enp2s14:1 192.168.5.5
$ ifconfig
enp2s14: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.5  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::215:60ff:fec4:ee02  prefixlen 64  scopeid 0x20<link>
        ether 00:15:60:c4:ee:02  txqueuelen 1000  (Ethernet)
```

```
        RX packets 17105  bytes 22992037 (21.9 MiB)
        RX errors 0  dropped 1  overruns 0  frame 0
        TX packets 10544  bytes 864868 (844.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 16

enp2s14:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.5.5  netmask 255.255.255.0  broadcast 192.168.5.255
        ether 00:15:60:c4:ee:02  txqueuelen 1000  (Ethernet)
        device interrupt 16
...
```

Теперь в системе создан (таким простым способом) дополнительный *алиасный* сетевой интерфейс (синоним) в подсеть `192.168.5.0/24`, и трафик с хостами этой подсети будет направляться через интерфейс `enp2s14:1` (хотя физически будет все так же проходить через `enp2s14`):

```
$ ip addr
...
2: enp2s14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.5/24 brd 192.168.1.255 scope global enp2s14
       valid_lft forever preferred_lft forever
    inet 192.168.5.5/24 brd 192.168.5.255 scope global enp2s14:1
       valid_lft forever preferred_lft forever
    inet6 fe80::215:60ff:fec4:ee02/64 scope link
       valid_lft forever preferred_lft forever
...
$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         192.168.1.1     0.0.0.0         UG    1024   0        0 enp2s14
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 enp2s14
192.168.5.0     0.0.0.0         255.255.255.0   U     0      0        0 enp2s14
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    1024   0        0 enp2s14
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 enp2s14
192.168.5.0     0.0.0.0         255.255.255.0   U     0      0        0 enp2s14
$ ping -c2 192.168.5.5
PING 192.168.5.5 (192.168.5.5) 56(84) bytes of data.
64 bytes from 192.168.5.5: icmp_seq=1 ttl=64 time=0.526 ms
64 bytes from 192.168.5.5: icmp_seq=2 ttl=64 time=0.323 ms
--- 192.168.5.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.323/0.424/0.526/0.103 ms
```

Техника создания и работы с сетевыми алиасами является в высшей степени полезной при отработке кодов модулей ядра, обсуждаемых далее.

Создание *виртуальных* сетевых интерфейсов, обладающих некоторыми дополнительными качествами относительно базовых интерфейсов, над которыми они надстроены (например, шифрование трафика по определенному алгоритму), и является главным предметом следующего далее обзора реализующих модулей ядра.

Инструменты наблюдения

При рассмотрении ранее драйверов блочных устройств должно было броситься в глаза то обстоятельство, что эти драйверы отличаются от драйверов других классов устройств (символьных, например) тем, что они потребовали для своего испытания, тестирования и отладки привлечения совершенно особого и широкого спектра специфического программного инструментария: fdisk, gparted, gdisk, `mkfs.*`, `mount` и других. Еще более явно это просматривается при работе с сетевыми интерфейсами и протоколами — успех здесь определяется тем, каким арсеналом инструментов мы обладаем и как гибко можем его использовать. Объясняется это тем, что здесь мы работаем с *совершенно иным* инструментарием, неприменимым к устройствам, и наоборот — инструментарий для работы с устройствами неприменим к сетевой сфере.

И поскольку представление сетевых интерфейсов принципиально отличается от представления устройств, то при отработке модулей ядра, поддерживающих сетевые средства, задействуется совершенно особое множество *команд-утилит*. Их мы используем для контроля, диагностики и управления сетевыми интерфейсами. Набор сетевых утилит, применяемых в сетевой разработке, огромен! Но далее мы только назовем некоторые из них — те, без которых такая работа просто невозможна...

Простейшими инструментами *диагностики* в нашей отработке примеров будет посылка «пульсов» — тестирующих ICMP-пакетов `ping` (был показан ранее) и `traceroute` (задержка прохождения промежуточных хостов на трассе маршрута):

```
$ traceroute 80.255.64.23
traceroute to 80.255.64.23 (80.255.64.23), 30 hops max, 60 byte packets
 1  192.168.1.1 (192.168.1.1)  1.052 ms  1.447 ms  1.952 ms
 2  * * *
 3  10.50.21.14 (10.50.21.14)  32.584 ms  34.609 ms  34.828 ms
 4  umc-10G-gw.ix.net.ua (195.35.65.50)  37.521 ms  38.751 ms  39.052 ms
 5  * * *
```

Команды `ping` и `traceroute` имеют множество опций (показываемых по `--help` и описанных в `man`). Например — из числа самых необходимых при отработке сетевых модулей — возможность направить поток ICMP только в указанный сетевой интерфейс из числа нескольких наличествующих:

```
$ ping -I wlo1 192.168.1.1
PING 192.168.1.1 (192.168.1.1) from 192.168.1.21 wlo1: 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=2.12 ms
```

```
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=1.99 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=64 time=2.00 ms
...
```

Самым известным (и одним из самых старых) инструментом *управления* сетевыми интерфейсами является утилита `ifconfig`:

```
$ ifconfig
...
cipsec0   Link encap:Ethernet  HWaddr 00:0B:FC:F8:01:8F
          inet addr:192.168.27.101  Mask:255.255.255.0
          inet6 addr: fe80::20b:fcff:fef8:18f/64 Scope:Link
          UP RUNNING NOARP  MTU:1356  Metric:1
          RX packets:4 errors:0 dropped:3 overruns:0 frame:0
          TX packets:18 errors:0 dropped:5 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:538 (538.0 b)  TX bytes:1670 (1.6 KiB)
...
wlan0     Link encap:Ethernet  HWaddr 00:13:02:69:70:9B
          inet addr:192.168.1.21  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::213:2ff:fe69:709b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10863 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11768 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3274108 (3.1 MiB)  TX bytes:1727121 (1.6 MiB)
```

Здесь показаны два сетевых интерфейса: физическая беспроводная сеть Wi-Fi (`wlan0`) и виртуальный интерфейс `cipsec0` (виртуальная частная сеть, VPN), созданный программными средствами (Cisco Systems VPN Client) от Cisco Systems, работающий через один и тот же (`wlan0` в показанном случае) физический канал (что подтверждает сказанное ранее о возможности наличия нескольких сетевых интерфейсов над одним физическом каналом). Для управления создаваемым сетевым интерфейсом (например, операции `up` или `down`) — в отличие от диагностики — утилита `ifconfig` потребует прав root.

> ***Примечание***
>
> Интересно, что если тот же VPN-канал к тому же удаленному серверу-хосту создать «родными» Linux средствами OpenVPN (вместо Cisco Systems VPN Client), то мы получим совершенно другой (и даже, если нужно, еще один дополнительный параллельный) сетевой интерфейс, — различия в интерфейсах обусловлены модулями ядра, которые их создавали:
>
> ```
> $ ifconfig tun0
> tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
>           inet addr:192.168.27.112  P-t-P:192.168.27.112  Mask:255.255.255.0
>           UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1412  Metric:1
>           RX packets:13 errors:0 dropped:0 overruns:0 frame:0
>           TX packets:13 errors:0 dropped:0 overruns:0 carrier:0
>           collisions:0 txqueuelen:500
>           RX bytes:1905 (1.8 KiB)  TX bytes:883 (883.0 b)
> ```

Более новым и не менее известным[4], но более развитым инструментом *управления* является утилита `ip` (в некоторых дистрибутивах может потребоваться ее отдельная установка с пакетом, известным под именем `iproute2`) — вот результаты ее выполнения для той же конфигурации:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
4: vboxnet0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether 0a:00:27:00:00:00 brd ff:ff:ff:ff:ff:ff
5: pan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
    link/ether ae:4c:18:a0:26:1b brd ff:ff:ff:ff:ff:ff
6: cipsec0: <NOARP,UP,LOWER_UP> mtu 1356 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:0b:fc:f8:01:8f brd ff:ff:ff:ff:ff:ff
$ ip addr show dev cipsec0
6: cipsec0: <NOARP,UP,LOWER_UP> mtu 1356 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:0b:fc:f8:01:8f brd ff:ff:ff:ff:ff:ff
    inet 192.168.27.101/24 brd 192.168.27.255 scope global cipsec0
    inet6 fe80::20b:fcff:fef8:18f/64 scope link
       valid_lft forever preferred_lft forever
```

Параметры утилиты допускают «быстрые» сокращения: `addr` — `a`, `show` — `s`, `link` — `l` и так далее. Важно, чтобы сокращаемый параметр стоял в допустимой позиции команды.

Утилита `ip` имеет очень разветвленный синтаксис (и возможности) — настолько разветвленный, что запомнить его нельзя... но, к счастью, имеет и такую же разветвленную (*древовидную*, многоуровневую) систему помощи:

```
$ ip help
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
       ip [ -force ] -batch filename
where  OBJECT := { link | addr | addrlabel | route | rule | neigh | ntable |
                   tunnel | maddr | mroute | monitor | xfrm }
       OPTIONS := { -V[ersion] | -s[tatistics] | -d[etails] | -r[esolve] |
                    -f[amily] { inet | inet6 | ipx | dnet | link } |
                    -o[neline] | -t[imestamp] | -b[atch] [filename] }
$ ip addr help
Usage: ip addr {add|change|replace} IFADDR dev STRING [ LIFETIME ]
                                                      [ CONFFLAG-LIST]
```

[4] Утилита `ifconfig` — очень давний универсальный инструмент управления сетью, наследие, присутствующее во всех POSIX операционных системах (Solaris, *BSD, QNX, MINIX, ...), а утилита `ip` появилась намного позже, и является, главным образом «изобретением Linux», хотя со временем начала с успехом использоваться и в других системах.

```
       ip addr del IFADDR dev STRING
       ip addr {show|flush} [ dev STRING ] [ scope SCOPE-ID ]
                            [ to PREFIX ] [ FLAG-LIST ] [ label PATTERN ]
...
```

Утилитами работы с сетевыми интерфейсами (`ifconfig` и `ip`) мы будем широко пользоваться при отработке модулей ядра, создающих такие интерфейсы.

Еще одним инструментом *управления* работой сетевой системы, не пересекающимся с отмеченными ранее, является утилита `route` — обеспечивающая управление таблицей роутинга ядра (и диагностики содержимого). Простейший пример использования утилиты `route` для изменений в таблице роутинга будет показан далее, при рассмотрении переименования сетевого интерфейса (хотя и все операции с роутингом вы можете выполнять все той же утилитой: `ip route ...`).

Один из самых полезных *специальных* инструментов при отработке и конфигурировании сетевого кода — это утилита `ethtool` (возможно, в вашем дистрибутиве ее придется установить стандартным способом из репозитория). Вот насколько расширенную статистику работы интерфейса дает эта утилита:

```
$ ip l
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
                                                                                qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group
                                                                        default qlen 1000
    link/ether 02:81:5f:d5:e7:17 brd ff:ff:ff:ff:ff:ff
$ sudo ethtool -S eth0
NIC statistics:
     tx_underflow: 0
     tx_carrier: 0
     tx_losscarrier: 0
     vlan_tag: 0
     tx_deferred: 0
     tx_vlan: 0
     tx_jabber: 0
     tx_frame_flushed: 0
     tx_payload_error: 0
     tx_ip_header_error: 0
     rx_desc: 0
     sa_filter_fail: 0
     overflow_error: 0
     ipc_csum_error: 0
     rx_collision: 0
     rx_crc_errors: 0
     dribbling_bit: 0
     rx_length: 0
     rx_mii: 0
     rx_multicast: 0
```

```
rx_gmac_overflow: 0
rx_watchdog: 0
da_rx_filter_fail: 0
sa_rx_filter_fail: 0
rx_missed_cntr: 0
rx_overflow_cntr: 0
rx_vlan: 0
rx_split_hdr_pkt_n: 0
tx_undeflow_irq: 0
tx_process_stopped_irq: 11857
tx_jabber_irq: 0
rx_overflow_irq: 0
rx_buf_unav_irq: 0
rx_process_stopped_irq: 0
rx_watchdog_irq: 0
tx_early_irq: 11856
fatal_bus_error_irq: 0
rx_early_irq: 0
threshold: 1
tx_pkt_n: 12009
rx_pkt_n: 55142
normal_irq_n: 0
rx_normal_irq_n: 55125
napi_poll: 67566
tx_normal_irq_n: 485
tx_clean: 12441
tx_set_ic_bit: 485
irq_receive_pmt_irq_n: 0
mmc_tx_irq_n: 0
mmc_rx_irq_n: 0
mmc_rx_csum_offload_irq_n: 0
irq_tx_path_in_lpi_mode_n: 0
irq_tx_path_exit_lpi_mode_n: 0
irq_rx_path_in_lpi_mode_n: 0
irq_rx_path_exit_lpi_mode_n: 0
phy_eee_wakeup_error_n: 0
ip_hdr_err: 0
ip_payload_err: 0
ip_csum_bypassed: 0
ipv4_pkt_rcvd: 0
ipv6_pkt_rcvd: 0
no_ptp_rx_msg_type_ext: 0
ptp_rx_msg_type_sync: 0
ptp_rx_msg_type_follow_up: 0
ptp_rx_msg_type_delay_req: 0
ptp_rx_msg_type_delay_resp: 0
ptp_rx_msg_type_pdelay_req: 0
```

```
ptp_rx_msg_type_pdelay_resp: 0
ptp_rx_msg_type_pdelay_follow_up: 0
ptp_rx_msg_type_announce: 0
ptp_rx_msg_type_management: 0
ptp_rx_msg_pkt_reserved_type: 0
ptp_frame_type: 0
ptp_ver: 0
timestamp_dropped: 0
av_pkt_rcvd: 0
av_tagged_pkt_rcvd: 0
vlan_tag_priority_val: 0
l3_filter_match: 0
l4_filter_match: 0
l3_l4_filter_no_match: 0
irq_pcs_ane_n: 0
irq_pcs_link_n: 0
irq_rgmii_n: 0
mtl_tx_status_fifo_full: 0
mtl_tx_fifo_not_empty: 0
mmtl_fifo_ctrl: 0
mtl_tx_fifo_read_ctrl_write: 0
mtl_tx_fifo_read_ctrl_wait: 0
mtl_tx_fifo_read_ctrl_read: 0
mtl_tx_fifo_read_ctrl_idle: 0
mac_tx_in_pause: 0
mac_tx_frame_ctrl_xfer: 0
mac_tx_frame_ctrl_idle: 0
mac_tx_frame_ctrl_wait: 0
mac_tx_frame_ctrl_pause: 0
mac_gmii_tx_proto_engine: 0
mtl_rx_fifo_fill_level_full: 0
mtl_rx_fifo_fill_above_thresh: 0
mtl_rx_fifo_fill_below_thresh: 0
mtl_rx_fifo_fill_level_empty: 0
mtl_rx_fifo_read_ctrl_flush: 0
mtl_rx_fifo_read_ctrl_read_data: 0
mtl_rx_fifo_read_ctrl_status: 0
mtl_rx_fifo_read_ctrl_idle: 0
mtl_rx_fifo_ctrl_active: 0
mac_rx_frame_ctrl_fifo: 0
mac_gmii_rx_proto_engine: 0
tx_tso_frames: 0
tx_tso_nfrags: 0
mtl_est_cgce: 0
mtl_est_hlbs: 0
mtl_est_hlbf: 0
mtl_est_btre: 0
```

```
     mtl_est_btrlm: 0
     q0_tx_pkt_n: 12009
     q0_tx_irq_n: 0
     q0_rx_pkt_n: 55142
     q0_rx_irq_n: 0
```

Но она предоставляет не только самую обширную диагностику, но и позволяет конфигурировать параметры интерфейсов самым тонким образом *из командной строки*. Вот ее справка, которая позволяет оценить обширность возможностей:

```
$ sudo ethtool --help | wc -l
165
```

К сожалению, не все сетевые интерфейсы, даже встроенные изначально в инсталляционные дистрибутивы, имеют (в настройках) возможность тонкой статистики и настройки:

```
$ inxi -S
System:    Host: raspberrypi Kernel: 5.15.32-v7+ armv7l bits: 32 Console: tty 0 Distro:
Raspbian GNU/Linux 11 (bullseye)
$ ip l
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT
group default qlen 1000
    link/ether b8:27:eb:7e:2c:a8 brd ff:ff:ff:ff:ff:ff
$ sudo ethtool -S eth0
no stats available
```

Еще один инструмент, который непременно понадобится при тщательном тестировании созданного модуля-драйвера, — это утилита разрешения сетевых имен в IP-адреса и наоборот:

```
$ nslookup yandex.ru  192.168.1.1
Server:               192.168.1.1
Address:              192.168.1.1#53
Non-authoritative answer:
Name:                 yandex.ru
Address: 213.180.204.11
Name:                 yandex.ru
Address: 93.158.134.11
Name:                 yandex.ru
Address: 213.180.193.11
```

Вторым параметром команды (необязательным) здесь является IP-адрес DNS-сервера, через который требуется выполнить разрешение имен (при его отсутствии будет использована последовательность DNS, конфигурированных в системе по умолчанию).

Для *анализа трафика* разрабатываемого сетевого интерфейса вам, безусловно, потребуется что-либо из числа известных утилит *сетевых сниферов* — таких как

tcpdump (**http://www.tcpdump.org/**), или ее GUI-эквивалент Wireshark (**http://www.wireshark.org/**). Посмотрим, как для одного из сетевых интерфейсов (p7p1):

```
$ ip addr show dev p7p1
3: p7p1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:9e:02:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global p7p1
    inet6 fe80::a00:27ff:fe9e:202/64 scope link
       valid_lft forever preferred_lft forever
```

выглядит результат tcpdump — далее приведен полученный в tcpdump протокол (показано только начало) выполнения операции ping на этот интерфейс с внешнего хоста LAN:

```
$ sudo tcpdump -i p7p1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on p7p1, link-type EN10MB (Ethernet), capture size 65535 bytes
08:57:53.070217 ARP, Request who-has 192.168.56.101 tell 192.168.56.1, length 46
08:57:53.070271 ARP, Reply 192.168.56.101 is-at 08:00:27:9e:02:02 (oui Unknown), length 28
08:57:53.070330 IP 192.168.56.1 > 192.168.56.101: ICMP echo request, id 2478, seq 1, length 64
08:57:53.070373 IP 192.168.56.101 > 192.168.56.1: ICMP echo reply, id 2478, seq 1, length 64
08:57:54.071415 IP 192.168.56.1 > 192.168.56.101: ICMP echo request, id 2478, seq 2, length 64
08:57:54.071464 IP 192.168.56.101 > 192.168.56.1: ICMP echo reply, id 2478, seq 2, length 64
...
```

Здесь мы видим работу ARP-механизма разрешения IP-адресов в локальной сети (начало протокола) и прием и передачу IP-пакетов (тип протокола ICMP).

> ***Примечание***
>
> В примере специально показано имя (p7p1) проводного (Ethernet) сетевого интерфейса в том виде, который он может иметь в некоторых (например, в Fedora) дистрибутивах Linux (вместо eth0, eth1, ...). Такое (новое) обозначение увязывает сетевой интерфейс с адресом реализующего его реального сетевого адаптера на шине PCI.

Инструменты интегрального тестирования

Ранее отмечалось, что для тестирования драйверов устройств оптимальными тестерами являются стандартные утилиты POSIX/GNU — такие как echo, cat, cp и другие. Для тестирования и отладки сетевых модулей ядра тоже хорошо бы предварительно определиться с набором тестовых инструментов (утилит), которые также были бы относительно стандартизованными (или широко используемыми) и позволяли бы проводить тестирование наиболее быстро и с наименьшими затратами.

Удачный вариант тестеров представляют собой утилиты передачи файлов по протоколу SSH: sftp и scp — обе утилиты копируют указанный (по URL) *сетевой файл*. Разница между ними состоит в том, что sftp требует указания только источника и копирует его в текущий каталог, а для scp указываются и источник, и приемник (и каждый из них может быть сетевым URL) — таким образом допускается выполнение копирования и из третьего, стороннего узла):

```
$ sftp olej@192.168.1.9:/home/olej/YYY
olej@192.168.1.9's password:
Connected to 192.168.1.9.
Fetching /home/olej/YYY to YYY
/home/olej/YYY                              100% 98MB 10.9MB/s 00:09

$ scp olej@192.168.1.137:/boot/initramfs-3.6.11-5.fc17.i686.img img1
olej@192.168.1.137's password:
initramfs-3.6.11-5.fc17.i686.img            100%   18MB  17.6MB/s   00:01
```

Еще одним эффективным инструментом тестирования и отладки может стать утилита `nc` (network cat). У этой утилиты есть множество возможностей (описанных в `man`) — в частности, она позволяет как передавать данные в сеть (клиент), так и принимать эти данные из сети (сервер):

```
$ echo 0123456789 > dg.txt
$ cat dg.txt | nc -l 12345

$ nc 192.168.56.1 12345 > file.txt
$ cat file.txt
0123456789
```

Здесь в первой группе команд `nc` запускается как *клиент*, во второй — как *сервер*, параметр `12345` — это согласованный номер порта TCP, через который происходит передача, а `192.168.56.1` — IP-адрес узла, *с которого* осуществляет прием сервер. По умолчанию используется протокол передачи TCP, но его можно изменить на UDP, используя для `nc` опцию `-u`.

В отношении сетевых протоколов транспортного уровня L4 (TCP, UDP, а также менее известных транспортных протоколов SCTP и DCCP) напомним следующее:

- количество портов ограничено с учетом 16-битной адресации (2^{16} = 65 536), начало — 0);
- все порты разделены на три диапазона:
  - *общеизвестные* (или *системные*): 0–1023;
  - *зарегистрированные* (или *пользовательские*): 1024–49 151;
  - *динамические* (или *частные*, *приватные*): 49 152–65 535.

Динамическими порты (от 49 152 до 65 535) являются в том смысле, что они могут быть использованы любым процессом и с любой целью. Часто программа, работающая на зарегистрированном порту (от 1024 до 49 151), порождает другие процессы, которые затем используют эти динамические порты.

Самая свежая информация о регистрации номеров портов может быть найдена здесь: **http://www.iana.org/numbers.htm#P**.

Структуры данных сетевого стека

Сетевая реализация построена так, чтобы не зависеть от конкретики протоколов. Весь сетевой обмен — все протоколы, не только TCP/IP или даже Ethernet, — обслуживаются единым стеком сетевых протоколов. Основной структурой данных, описывающей *сетевой интерфейс* (устройство), является `struct net_device` (к ней мы вернемся позже, описывая устройство).

А вот *основная* структура обмениваемых *данных* (между сетевыми уровнями), на движении экземпляров данных которой между сетевыми уровнями построена работа всей подсистемы, — это *буферы сокетов* (определения в <linux/skbuff.h>). Буфер сокетов состоит из 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 ее представить:

```
typedef unsigned char *sk_buff_data_t;
struct sk_buff {
   struct sk_buff *next; /* These two members must be first. */
   struct sk_buff *prev;
...
   sk_buff_data_t  transport_header;
   sk_buff_data_t  network_header;
   sk_buff_data_t  mac_header;
...
   unsigned char *head,
                 *data;
...
};
```

Структура вложенности заголовков сетевых уровней в точности соответствует структуре инкапсуляции сетевых протоколов внутри друг друга — это позволяет обрабатывающему слою получать доступ к информации, относящейся только к нужному ему слою.

Экземпляры данных типа `struct sk_buff`:

- возникают при поступлении очередного сетевого пакета (здесь нужно принимать во внимание возможность сегментации пакетов) из внешней физической среды распространения данных. Об этом событии извещает прерывание (IRQ), генерируемое сетевым адаптером. При этом создается (чаще извлекается из пула использованных) очередной экземпляр буфера сокета, затем он заполняется данными из поступившего пакета и передается *вверх* от сетевого слоя к слою до приложения *прикладного уровня*, которое и является получателем пакета. После чего экземпляр данных буфера сокета уничтожается (утилизируется, помещается в пул использованных);
- возникают в среде приложения *прикладного уровня*, которое является *отправителем* пакета данных. Пакет отправляемых данных помещается в созданный буфер сокета, который начинает перемещаться вниз от сетевого слоя к слою до

достижения канального уровня L2. На этом уровне осуществляется физическая передача данных пакета через сетевой адаптер в среду распространения. В случае успешного завершения передачи (что подтверждается прерыванием, генерируемым сетевым адаптером, часто по той же линии IRQ, что и при приеме пакета) буфер сокета уничтожается (утилизируется). При отсутствии подтверждения отправки (IRQ) от адаптера обычно делается несколько повторных попыток прежде, чем принять решение об ошибке канала.

Прохождение экземпляра данных буфера сокета сквозь стек сетевых протоколов будет детально проанализировано далее.

Драйверы: сетевой интерфейс

Задача сетевого интерфейса — быть тем местом, в котором:

- создаются экземпляры структуры `struct sk_buff` по каждому принятому из интерфейса пакету (здесь нужно принимать во внимание возможность сегментации IP-пакетов), после чего созданный экземпляр структуры продвигается по стеку протоколов вверх — до получателя пользовательского пространства, где он и уничтожается;
- исходящие экземпляры структуры `struct sk_buff`, порожденные где-то на верхних уровнях протоколов пользовательского пространства, должны отправляться (чаще всего каким-то аппаратным механизмом), а сами экземпляры структуры после этого — уничтожаться.

Более детально эти вопросы мы рассмотрим чуть позже — при обсуждении прохождения пакетов сквозь стек сетевых протоколов. А пока наша задача — *создание* той концевой точки (интерфейса), где эти последовательности действий начинаются и завершаются.

Создание сетевых интерфейсов

Рассмотрим пример простого создания и регистрации в системе нового сетевого интерфейса (многие примеры этого раздела заимствованы из [6] с небольшими модификациями и содержатся в папке network/net сопровождающего книгу файлового архива):

network.c

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

static struct net_device *dev;

static int my_open(struct net_device *dev) {
   printk(KERN_INFO "! Hit: my_open(%s)\n", dev->name);
   /* start up the transmission queue */
```

```
   netif_start_queue(dev);
   return 0;
}

static int my_close(struct net_device *dev) {
   printk(KERN_INFO "! Hit: my_close(%s)\n", dev->name);
   /* shutdown the transmission queue */
   netif_stop_queue(dev);
   return 0;
}

/* Note this method is only needed on some; without it
   module will fail upon removal or use. At any rate there is a memory
   leak whenever you try to send a packet through in any case*/
static int stub_start_xmit(struct sk_buff *skb, struct net_device *dev) {
   dev_kfree_skb(skb);
   return 0;
}

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

static void my_setup(struct net_device *dev) {
   /* Fill in the MAC address with a phoney */
   for(int j = 0; j < ETH_ALEN; ++j)
      dev->dev_addr[j] = (char)j;
   ether_setup(dev);
   dev->netdev_ops = &ndo;
}

static int __init my_init(void) {
   printk(KERN_INFO "! Loading stub network module:....");
#if (LINUX_VERSION_CODE < KERNEL_VERSION(3, 17, 0))
   dev = alloc_netdev(0, "fict%d", my_setup);
#else
   dev = alloc_netdev(0, "fict%d", NET_NAME_ENUM, my_setup);
#endif
   if(register_netdev(dev)) {
      printk(KERN_INFO "! Failed to register\n");
      free_netdev(dev);
      return -1;
   }
   printk(KERN_INFO "! Succeeded in loading %s!\n", dev_name(&dev->dev));
   return 0;
}
```

```
static void __exit my_exit(void) {
   printk(KERN_INFO "! Unloading stub network module\n");
   unregister_netdev(dev);
   free_netdev(dev);
}

module_init(my_init);
module_exit(my_exit);

MODULE_AUTHOR("Bill Shubert");
MODULE_AUTHOR("Jerry Cooperstein");
MODULE_AUTHOR("Tatsuo Kawasaki");
MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_DESCRIPTION("LDD:1.0 s_24/lab1_network.c");
MODULE_LICENSE("GPL v2");
```

Здесь нужно обратить внимание на вызов `alloc_netdev()`, который в качестве параметра получает шаблон (`%d`) имени нового интерфейса, — мы задаем префикс имени интерфейса (`fict`), а система сама присваивает первый свободный номер интерфейса с таким префиксом. Обратите также внимание, как в цикле заполнился фиктивным значением `00:01:02:03:04:05` MAC-адрес интерфейса, что мы увидим вскоре в диагностике.

Тут же воспользуемся сделанным модулем и посмотрим, что там у нас с сетевыми интерфейсами:

```
$ sudo insmod network.ko
$ ip link
...
5: fict0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode DEFAULT
                                                                    group default qlen 1000
    link/ether 00:01:02:03:04:05 brd ff:ff:ff:ff:ff:ff
$ ip a s dev fict0
5: fict0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group
                                                                        default qlen 1000
    link/ether 00:01:02:03:04:05 brd ff:ff:ff:ff:ff:ff
$ sudo rmmod network
$ ip a s dev fict0
Device "fict0" does not exist.
```

Мы только что создали новый сетевой интерфейс. Он еще не подготовлен для осуществления сетевого трафика, но и не нарушает работоспособность сетевой подсистемы и не препятствует работе других (существовавших до того) сетевых интерфейсов.

Новая схема, и детальнее о ее создании

Описанная ранее укрупненная схема использовалась в неизменном виде на протяжении как минимум первых десяти лет существования сетевой подсистемы Linux,

и в таком виде представлена во многих источниках в литературе и обсуждениях. Но, начиная с ядра 3.17, прототип *макроса* создания интерфейса меняется (<linux/netdevice.h>):

- было:

```
#define alloc_netdev(sizeof_priv, name, setup)
```

- стало:

```
#define alloc_netdev(sizeof_priv, name, name_assign_type, setup)
```

Как легко видеть, теперь вместо трех параметров — четыре, третий из которых — константа, определяющая порядок нумерации создаваемых интерфейсов и описанная в том же файле определений:

```
/* interface name assignment types (sysfs name_assign_type attribute) */
#define NET_NAME_UNKNOWN     0 /* unknown origin (not exposed to userspace) */
#define NET_NAME_ENUM        1 /* enumerated by kernel */
#define NET_NAME_PREDICTABLE 2 /* predictably named by the kernel */
#define NET_NAME_USER        3 /* provided by user-space */
#define NET_NAME_RENAMED     4 /* renamed by user-space */
```

Похоже, что пока (на уровне ядра 3.17) эти различные схемы еще не дифференцируются, но начата активная работа по модернизации схемы. Следствием же является, что все сетевые драйверы, особенно проприетарные или от производителей, *должны быть переписаны* с учетом новой схемы.

Для иллюстрации и изучения этой схемы приведем модифицированный вариант предыдущего модуля (показаны только существенно различающиеся части):

mulnet.c

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

static int num = 1;              // число создаваемых интерфейсов
module_param(num, int, 0);
static char* title;              // префикс имени интерфейсов
module_param(title, charp, 0);
static int digit = 1;            // числовые суфиксы (по умолчанию)
module_param(digit, int, 0);
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(3, 17, 0))
static int mode = 1;             // режим нумерации интерфейсов
module_param(mode, int, 0);
#endif

static struct net_device *adev[] = { NULL, NULL, NULL, NULL };

...
```

```
static int ipos;

static void __init my_setup(struct net_device *dev) {
   /* Fill in the MAC address with a phoney */
   int j;
   for(j = 0; j < ETH_ALEN; ++j)
      dev->dev_addr[j] = (char)(j + ipos);
   ether_setup(dev);
   dev->netdev_ops = &ndo;
}

static int __init my_init(void) {
   char prefix[20];
   if(num > sizeof(adev) / sizeof(adev[0])) {
      printk(KERN_INFO "! link number error");
      return -EINVAL;
   }
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(3, 17, 0))
   if(mode < 0 || mode > NET_NAME_RENAMED) {
      printk(KERN_INFO "! unknown name assign mode");
      return -EINVAL;
   }
#endif
   printk(KERN_INFO "! loading network module for %d links", num);
   sprintf(prefix, "%s%s", (NULL == title ? "fict" : title), "%d");
   for(ipos = 0; ipos < num; ipos++) {
      if(!digit)
         sprintf(prefix, "%s%c", (NULL == title ? "fict" : title), 'a' + ipos);
#if (LINUX_VERSION_CODE < KERNEL_VERSION(3, 17, 0))
      adev[ipos] = alloc_netdev(0, prefix, my_setup);
#else
      adev[ipos] = alloc_netdev(0, prefix, NET_NAME_UNKNOWN + mode, my_setup);
#endif
      if(register_netdev(adev[ipos])) {
         int j;
         printk(KERN_INFO "! failed to register");
         for(j = ipos; j >= 0; j--) {
            if(j != ipos) unregister_netdev(adev[ipos]);
            free_netdev(adev[ipos]);
         }
         return -ELNRNG;
      }
   }
   printk(KERN_INFO "! succeeded in loading %d devices!", num);
   return 0;
}
```

```
static void __exit my_exit(void) {
   int i;
   printk(KERN_INFO "! unloading network module");
   for(i = 0; i < num; i++) {
      unregister_netdev(adev[i]);
      free_netdev(adev[i]);
   }
}
...
```

В дополнение здесь показано создание нескольких (трех) однотипных сетевых интерфейсов (параметр `num=...`) и возможность «ручного» присвоения (параметр `digit=0`) им произвольных имен (без формата `%d`):

```
$ sudo insmod mulnet.ko num=3 title=zz mode=2
$ sudo cat /proc/modules | grep mulnet
mulnet 16384 0 - Live 0xffffffffc0f4c000 (OE)
$ ip link
...
6: zz0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 00:01:02:03:04:05 brd ff:ff:ff:ff:ff:ff
7: zz1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 01:02:03:04:05:06 brd ff:ff:ff:ff:ff:ff
8: zz2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 02:03:04:05:06:07 brd ff:ff:ff:ff:ff:ff
$ sudo rmmod mulnet
```

А вот как выглядят интерфейсы, если они создаются без числовых суффиксов, в режиме ручного формирования имен:

```
$ sudo insmod mulnet.ko num=3 digit=0
$ ip link
...
9: ficta: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 00:01:02:03:04:05 brd ff:ff:ff:ff:ff:ff
10: fictb: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode
DEFAULT group default qlen 1000
    link/ether 01:02:03:04:05:06 brd ff:ff:ff:ff:ff:ff
11: fictc: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode
DEFAULT group default qlen 1000
    link/ether 02:03:04:05:06:07 brd ff:ff:ff:ff:ff:ff
```

Если в вашей Linux-системе для управления сетевой подсистемой используется аплет Network Manager, то непосредственно после запусков показанных модулей созданные три интерфейса сразу же в нем отобразятся (рис. 4.4).

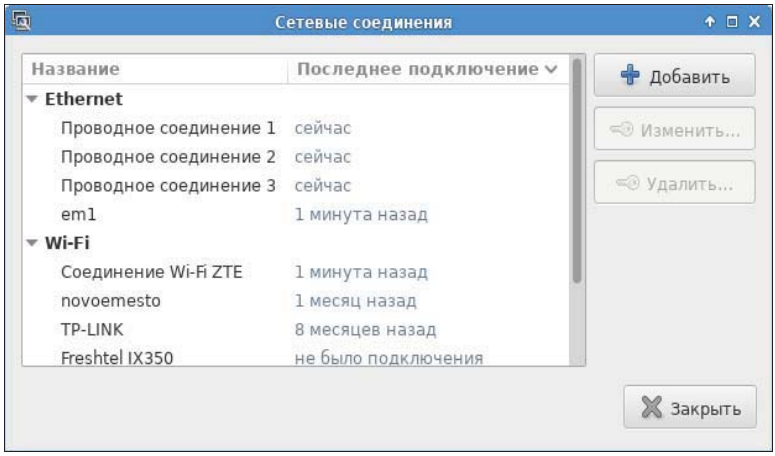


Рис. 4.4. Новые созданные сетевые интерфейсы (три экземпляра)

А после создания сетевого интерфейса (когда мы придадим ему определенную функциональность) вы сразу можете начинать его конфигурирование (в принципе достаточно хлопотное и рутинное), пользуясь GUI-инструментом (рис. 4.5).

Обращаем внимание, что при последовательных создании и уничтожении сетевых интерфейсов их *нумерация* (в утилите `ip`) остается сквозной возрастающей (продолжается от последнего созданного).

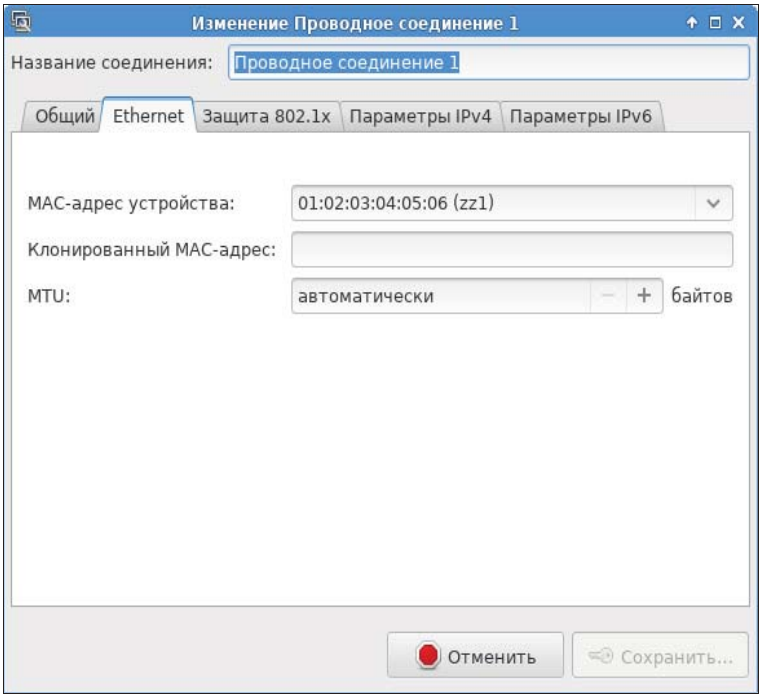


Рис. 4.5. Свойства созданного сетевого интерфейса

Операции сетевого интерфейса

Чтобы «придать жизнь» созданному сетевому интерфейсу, нужно реализовать для него набор обменных операций, чему и будет посвящена оставшаяся часть этого раздела. Вся связь сетевого интерфейса с выполняемыми на нем *операциями* осуществляется через таблицу функций <linux/netdevice.h> операций сетевого интерфейса ndo (**n**et **d**evice **o**perations):

```
struct net_device_ops {
        int                     (*ndo_init)(struct net_device *dev);
        void                    (*ndo_uninit)(struct net_device *dev);
        int                     (*ndo_open)(struct net_device *dev);
        int                     (*ndo_stop)(struct net_device *dev);
        netdev_tx_t             (*ndo_start_xmit) (struct sk_buff *skb,
                                                   struct net_device *dev);
        ...
        struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
        void                    (*ndo_get_stats64)(struct net_device *dev,
                                 struct rtnl_link_stats64 *storage);
        ...
}
```

В ядре 3.09, например, в struct net_device_ops определено 39 операций (и около 50 операций в ядре 3.14, и около 76 — в ядре 5.4), но реально разрабатываемые модули реализуют только некоторую малую часть из них.

Характерно, что в таблице операций интерфейса присутствует операция *передачи* сокетного буфера ndo_start_xmit в физическую среду, но вовсе нет операции *приема* пакетов (сокетных буферов). Это совершенно естественно, как мы увидим вскоре: принятые пакеты (например, в обработчике аппаратного прерывания IRQ) тут же передаются в *очередь* (ядра) принимаемых пакетов и далее уже обрабатываются сетевым стеком. А вот выполнять операцию ndo_start_xmit обязательно, хотя бы как минимум для вызова API ядра dev_kfree_skb(), который утилизирует (уничтожает) сокетный буфер после успешной (да и безуспешной тоже) операции передачи пакета. Если этого не делать, в системе возникнет слабо выраженная утечка памяти (с каждым пакетом), которая в конечном итоге рано или поздно приведет к краху системы.

Теперь мы можем начать установку в системе созданного нами ранее (пока еще фиктивного) сетевого устройства:

```
$ sudo insmod network.ko
$ dmesg | tail -n3
[34539.455022] ! Loading stub network module:....
[34539.455398] ! Succeeded in loading fict0!
[34539.479321] ! Hit: my_open(fict0)
$ ip link show dev fict0
12: fict0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode
DEFAULT group default qlen 1000
    link/ether 00:01:02:03:04:05 brd ff:ff:ff:ff:ff:ff
```

```
$ sudo ifconfig fict0 192.168.1.222 up
$ ping -c4 192.168.1.222
PING 192.168.1.222 (192.168.1.222) 56(84) bytes of data.
64 bytes from 192.168.1.222: icmp_seq=1 ttl=64 time=0.030 ms
64 bytes from 192.168.1.222: icmp_seq=2 ttl=64 time=0.056 ms
64 bytes from 192.168.1.222: icmp_seq=3 ttl=64 time=0.058 ms
64 bytes from 192.168.1.222: icmp_seq=4 ttl=64 time=0.072 ms

--- 192.168.1.222 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3065ms
rtt min/avg/max/mdev = 0.030/0.054/0.072/0.015 ms
$ route -n
Таблица маршутизации ядра протокола IP
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0         192.168.1.3     0.0.0.0         UG    100    0        0 eno1
0.0.0.0         192.168.1.6     0.0.0.0         UG    101    0        0 eno2
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 eno1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 fict0
192.168.1.0     0.0.0.0         255.255.255.0   U     100    0        0 eno1
192.168.1.0     0.0.0.0         255.255.255.0   U     101    0        0 eno2
```

Показанное пока вовсе не означает какую-то реальную работоспособность созданного интерфейса — создаваемый поток сокетных буферов замыкается в петлю внутри самого сетевого стека:

```
$ ifconfig fict0
fict0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.222  netmask 255.255.255.0  broadcast 192.168.1.255
        ether 00:01:02:03:04:05  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Обратите внимание, как совершенно *произвольное значение* (выбранное нами) заполняется в структуре `net_device` и устанавливается в качестве MAC-адреса (аппаратного) созданного интерфейса (в функции `my_setup()`).

Как уже отмечалось ранее, основу структуры описания сетевого интерфейса составляет структура `struct net_device`, описанная в <linux/netdevice.h>. При работе с сетевыми интерфейсами ее стоит изучить весьма тщательно. Это очень крупная структура, содержащая не только описание аппаратных средств, но и конфигурационные параметры сетевого интерфейса по отношению к вышележащим протоколам (приведенный далее пример взят из ядра 3.09, но только потому, что там поля прокомментированы. А в ядре 5.4 она вообще огромна и содержит множество параметров, которые нужно изучать непосредственно разработчикам оборудования):

```
struct net_device {
   char  name[IFNAMSIZ] ;
...
```

```
   unsigned long  mem_end;   /* shared mem end     */
   unsigned long  mem_start; /* shared mem start   */
   unsigned long  base_addr; /* device I/O address */
   unsigned int   irq;       /* device IRQ number  */
...
   unsigned       mtu;       /* interface MTU value     */
   unsigned short type;      /* interface hardware type */
...
   struct net_device_stats stats;
   struct list_head dev_list;
...
   /* Interface address info. */
   unsigned char  perm_addr[MAX_ADDR_LEN];   /* permanent hw address    */
   unsigned char  addr_len;                  /* hardware address length */
...
}
```

Здесь поле `type`, например, определяет тип аппаратного адаптера с точки зрения ARP-механизма разрешения MAC-адресов (<linux/if_arp.h>):

```
...
#define ARPHRD_ETHER       1    /* Ethernet 10Mbps           */
...
#define ARPHRD_IEEE802     6    /* IEEE 802.2 Ethernet/TR/TB */
#define ARPHRD_ARCNET      7    /* ARCnet                    */
...
#define ARPHRD_IEEE1394   24    /* IEEE 1394 IPv4 - RFC 2734 */
...
#define ARPHRD_IEEE80211 801    /* IEEE 802.11               */
```

Здесь же заносятся и такие совершенно аппаратные характеристики интерфейса (реализующего его физического адаптера), как, например, адрес базовой области ввода/вывода (`base_addr`), используемая линия аппаратного прерывания (`irq`), максимальная длина пакета для данного интерфейса (`mtu`)...

Детальный разбор огромного числа полей `struct net_device` (этой и любой другой сопутствующей) или их возможных значений не имеет смысла, хотя бы потому, что эта структура радикально изменяется даже от подверсии к подверсии ядра, — такой разбор должен проводиться «по месту» на основе изучения названных ранее заголовочных файлов.

Со структурой сетевого интерфейса обычно создается и связывается (кодом модуля) *приватная структура данных*, в которой пользователь может размещать произвольные собственные данные любой сложности, ассоциированные с интерфейсом. Это обычная практика ядра Linux, а не только сетевой подсистемы. Указатель такой приватной структуры помещается в структуру сетевого интерфейса. Это особо актуально, если предполагается, что драйвер может создавать несколько сетевых интерфейсов (например, несколько идентичных сетевых адаптеров). Доступ к приватной структуре данных должен определяться *исключительно* специально опреде-

ленной для того функцией `netdev_priv()`. Далее показан возможный вид функции — это определение взято из ядра 5.4, но никто не даст гарантий, что в другом ядре (раньшем или позднем) оно радикально не отличается:

```
/**
 * Get network device private data
 */
static inline void *netdev_priv(const struct net_device *dev)
{
        return (char *)dev + ALIGN(sizeof(struct net_device), NETDEV_ALIGN);
}
```

> ***Примечание***
>
> Как легко видеть из определения, приватная структура данных дописывается непосредственно в хвост `struct net_device` — это обычная практика создания структур переменного размера, принятая в языке C, начиная со стандарта C89 (и в C99). Но именно из-за этого, учитывая эффекты выравнивания данных (и их возможного изменения в будущем), нежелательно адресоваться к приватным данным непосредственно, а следует использовать только `netdev_priv()`.

При начальном размещении интерфейса размер определенной пользователем приватной структуры передается первым параметром функции размещения — например, так:

```
child = alloc_netdev(sizeof(struct priv), "fict%d", &setup);
```

После успешного выполнения размещения интерфейса приватная структура также будет размещена («в хвост» структуре `struct net_device`) и станет доступна по вызову `netdev_priv()`.

Все структуры `struct net_device`, описывающие доступные сетевые интерфейсы в системе, увязаны в единый связный список.

> ***Примечание***
>
> В ядре Linux все и любые списочные связные структуры строятся на основе API кольцевых двухсвязных списков — структур данных `struct list_head` (поле `dev_list` в `struct net_device`). Техника связных списков в ядре Linux будет подробно рассмотрена позже, когда речь пойдет про API ядра.

Следующий пример визуализирует содержимое списка сетевых интерфейсов:

devices.c

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

static int __init my_init(void) {
   struct net_device *dev;
   printk(KERN_INFO "Hello: module loaded at 0x%px\n", my_init);
   dev = first_net_device(&init_net);
```

```
   printk(KERN_INFO "Hello: dev_base address=0x%px\n", dev);
   while (dev) {
      char smac[ETH_ALEN*3];
      struct rtnl_link_stats64 stat = {};
      for(int j = 0; j < ETH_ALEN; j++)
         sprintf(smac + j * 3, "%02x%c",
                 dev->dev_addr[j], ETH_ALEN - 1 != j ? ':' : '\0' );
      if (dev->netdev_ops->ndo_get_stats64 != NULL)
         dev->netdev_ops->ndo_get_stats64(dev, &stat);
      printk(KERN_INFO "name=%-6s irq=%-3d MAC=%s "
                       "rx_bytes=%-8llu tx_bytes=%-8llu \n",
                       dev->name, dev->irq, smac,
                       stat.rx_bytes, stat.tx_bytes);
      dev = next_net_device(dev);
    }
    return -1;
}

module_init(my_init);

MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_LICENSE("GPL v2");
```

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

```
$ sudo insmod mulnet.ko num=4 title=abc
$ ip l
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group
default qlen 1000
    link/ether 90:b1:1c:54:3a:46 brd ff:ff:ff:ff:ff:ff
3: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group
default qlen 1000
    link/ether 90:b1:1c:54:3a:47 brd ff:ff:ff:ff:ff:ff
4: team1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT
group default qlen 1000
    link/ether ee:8b:90:64:9b:dd brd ff:ff:ff:ff:ff:ff
5: abc0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 00:01:02:03:04:05 brd ff:ff:ff:ff:ff:ff
6: abc1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 01:02:03:04:05:06 brd ff:ff:ff:ff:ff:ff
7: abc2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 02:03:04:05:06:07 brd ff:ff:ff:ff:ff:ff
```

```
8: abc3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode DEFAULT
group default qlen 1000
    link/ether 03:04:05:06:07:08 brd ff:ff:ff:ff:ff:ff
$ sudo insmod devices.ko
insmod: ERROR: could not insert module devices.ko: Operation not permitted
$ dmesg | tail -n 10
[18147.771220] Hello: module loaded at 0xffffffffc0d24000
[18147.771222] Hello: dev_base address=0xffff95f48b20e000
[18147.771228] name=lo     irq=0   MAC=00:00:00:00:00:00 rx_bytes=9602226  tx_bytes=9602226
[18147.771232] name=eno1   irq=16  MAC=90:b1:1c:54:3a:46 rx_bytes=33949503 tx_bytes=11483504
[18147.771235] name=eno2   irq=17  MAC=90:b1:1c:54:3a:47 rx_bytes=22621614 tx_bytes=7072507
[18147.771240] name=team1  irq=0   MAC=ee:8b:90:64:9b:dd rx_bytes=0        tx_bytes=0
[18147.771242] name=abc0   irq=0   MAC=00:01:02:03:04:05 rx_bytes=0        tx_bytes=0
[18147.771244] name=abc1   irq=0   MAC=01:02:03:04:05:06 rx_bytes=0        tx_bytes=0
[18147.771246] name=abc2   irq=0   MAC=02:03:04:05:06:07 rx_bytes=0        tx_bytes=0
[18147.771248] name=abc3   irq=0   MAC=03:04:05:06:07:08 rx_bytes=0        tx_bytes=0
```

Переименование сетевого интерфейса

В приведенном только что примере было показано, как имя сетевого интерфейса задается модулем, создающим этот интерфейс. Но имя сетевого интерфейса (созданного модулем или присутствующего в системе) — это не совершенно константное значение, которое должно оставаться неизменным во все время работы интерфейса в системе. Сравните:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN mode DEFAULT qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
$ ip link show wlan0
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN mode DEFAULT qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
$ sudo ip link set dev wlan0 name wln2
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wln2: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN mode DEFAULT qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
```

Здесь имя `wlan0` исходного интерфейса Wi-Fi было изменено командой `ip` на новое имя `wln2`. И далее с этим новым именем интерфейса будут работать все сетевые утилиты и протоколы.

Это не имеет прямого отношения к программированию модулей ядра, но оказывается чрезвычайно *мощным* инструментом при тестировании модулей, при экспериментах или при конфигурировании программных пакетов, включающих модули ядра сетевых устройств, которые создают свои новые сетевые интерфейсы. Но для использования этого инструмента нужно учесть несколько факторов.

Пусть мы имеем первоначально систему с двумя интерфейсами (`p2p1` и `p7p1`):

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 p2p1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 p2p1
192.168.56.1    0.0.0.0         255.255.255.255 UH    0      0        0 p7p1
```

Изменить имя можно только для остановленного интерфейса — в противном случае интерфейс будет «занят» для операции (в первоначальном примере `wlan0` и был как раз в остановленном состоянии):

```
$ sudo ip link set dev p7p1 name crypt0
RTNETLINK answers: Device or resource busy
```

Так что интерфейс нужно остановить для выполнения переименования, после чего снова поднять:

```
$ sudo ifconfig p7p1 down
$ sudo ip link set dev p7p1 name crypt0
$ sudo ifconfig crypt0 192.168.56.4
$ ip address show dev crypt0
3: crypt0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:08:9a:bd brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.3/32 brd 192.168.56.3 scope global crypt0
    inet6 fe80::a00:27ff:fe08:9abd/64 scope link
       valid_lft forever preferred_lft forever
```

Но после остановки и перезапуска интерфейса разрушается соответствующая ему запись в таблице роутинга (такое произошло бы, если бы мы и не переименовывали интерфейс):

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 p2p1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 p2p1
$ sudo route add -net 192.168.56.0 netmask 255.255.255.0 dev crypt0
$ route -n
Kernel IP routing table
```

При работающем интерфейсе `crypt0` с присвоенным ему IP создается впечатление его неработоспособности, поэтому для восстановления работы нужно добавить запись об интерфейсе в таблицу роутинга:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 p2p1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 p2p1
192.168.56.0    0.0.0.0         255.255.255.0   U     0      0        0 crypt0
```

После чего можно убедиться в полной работоспособности интерфейса с новым именем:

```
$ ping 192.168.56.1
PING 192.168.56.1 (192.168.56.1) 56(84) bytes of data.
64 bytes from 192.168.56.1: icmp_req=1 ttl=64 time=0.422 ms
64 bytes from 192.168.56.1: icmp_req=2 ttl=64 time=0.239 ms
^C
--- 192.168.56.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.239/0.330/0.422/0.093 ms
```

Путь пакета сквозь стек протоколов

Теперь у нас достаточно деталей, чтобы проследить путь пакетов (буферов сокетов) сквозь сетевой стек, увидеть, как буферы сокетов возникают в системе и когда они ее покидают, а также ответить на вопрос, почему вышележащие протокольные уровни (будут рассмотрены чуть ниже) никогда не порождают и не уничтожают буферы сокетов, а только обрабатывают (или модифицируют) содержащуюся в них информацию (работают как фильтры). Итак, последовательность связей мы можем разложить в следующем порядке

Прием: традиционный подход

Традиционный подход состоит в том, что каждый приходящий сетевой пакет порождает аппаратное прерывание по линии IRQ адаптера, что и служит сигналом на прием очередного сетевого пакета и создание буфера сокета для его сохранения и обработки принятых данных. Порядок действий модуля сетевого интерфейса при этом следующий:

1. Читая конфигурационную область PCI адаптера 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... (все это будет рассмотрено позже, когда речь пойдет про аппаратные шины).

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 приему нового пакета из сети (то же прерывание может происходить и при завершении отправки пакета в сеть — здесь нужен анализ причины) создается (или запрашивается из пула используемых) новый экземпляр буфера сокетов:

```
static irqreturn_t my_interrupt(int irq, void *dev_id) {
   ...
   struct sk_buff *skb = kmalloc(sizeof(struct sk_buff), ...);
   // заполнение данных *skb чтением из портов сетевого адаптера
   netif_rx(skb);
   return IRQ_HANDLED;
}
```

Все эти действия выполняются не в самом обработчике верхней половины прерываний от сетевого адаптера, а в обработчике отложенного прерывания `NET_RX_SOFTIRQ` для этой линии. Последним действием станет передача заполненного сокетного буфера вызову `netif_rx()` (или `netif_receive_skb()`), который и запустит процесс движения его (буфера) вверх по структуре сетевого стека (отметит отложенное программное прерывание `NET_RX_SOFTIRQ` для исполнения).

Прием: высокоскоростной интерфейс

Особенность природы сетевых интерфейсов состоит в том, что их активность носит взрывной характер: после весьма продолжительных периодов молчания возникают интервалы пиковой активности, когда сетевые пакеты (сегментированные на IP-пакеты объемы передаваемых данных) следуют сплошной плотной чередой. После такого пика активности могут снова наступать значительные промежутки полного отсутствия активности или вялой активности на интерфейсе (обмен ARP-пакетами для обновления информации разрешения локальных адресов и подобные виды активности). Современные сетевые карты Ethernet используют скорости обмена до 10 Гбит/с, но уже даже при значительно более низких интенсивностях традиционный подход становится нецелесообразным — в периоды высокой плотности поступления пакетов:

- новые приходящие пакеты создают вложенные запросы IRQ нескольких уровней при еще не обслуженном приеме текущего IRQ;
- асинхронное обслуживание каждого IRQ в плотном потоке создает слишком большие накладные расходы.

Поэтому был добавлен набор API для обработки таких плотных потоков пакетов, поступающих с высокоскоростных интерфейсов, который и получил название NAPI (New API[5]). Идея состоит в том, чтобы прием пакетов осуществлять не методом аппаратного прерывания, а методом *программного опроса* (polling) — точнее, комбинацией этих двух возможностей:

[5] Естественно, до какого времени он будет «новым», неизвестно, — очевидно, до появления еще более нового.

1. При поступлении *первого* пакета «пачки» инициируется прерывание IRQ адаптера (все начинается как в традиционном методе).
2. В обработчике прерывания *запрещается* поступление дальнейших запросов прерывания с этой линии IRQ по *приему* пакетов.

   IRQ с этой же линии по *отправке* пакетов могут продолжать поступать — таким образом, этот запрет является не программным запретом линии IRQ со стороны процессора, а записью управляющей информации в *аппаратные регистры* сетевого адаптера. Адаптер должен предусматривать такое раздельное управление поступлением прерываний по приему и передаче, но для современных высокоскоростных адаптеров это обычно соблюдается.
3. После прекращения прерываний по приему обработчик переходит в режим циклического считывания и обработки принятых из сети пакетов, сетевой адаптер при этом накапливает поступающие пакеты во внутреннем кольцевом буфере приема, а считывание производится либо до полного исчерпания кольцевого буфера, либо до определенного порогового числа считанных пакетов (10, 20, ...), называемого *бюджетом функции поллинга*.

   Естественно, это считывание и обработка пакетов происходят не в собственно обработчике прерывания (верхней половине), а в его отсроченной части.
4. По каждому принятому в опросе пакету генерируется сокетный буфер для продвижения его по стеку сетевых протоколов вверх.
5. После *завершения цикла* программного опроса по его результатам устанавливается состояние завершения `NAPI_STATE_DISABLE` (если не осталось больше не считанных пакетов в кольцевом буфере адаптера) или `NAPI_STATE_SCHED` (это говорит, что устройство адаптера должно продолжать опрашиваться, когда ядро в следующий раз перейдет к циклу опросов в отложенном обработчике прерываний).
6. Если результатом является `NAPI_STATE_DISABLE`, то после завершения цикла программного опроса восстанавливается разрешение генерации прерываний по линии IRQ приема пакетов (записью в порты сетевого адаптера).

В реализующем коде модуля это укрупненно должно выглядеть подобно следующему (при условии, что линия IRQ связана с аппаратным адаптером, как это описано для традиционного метода):

1. Реализатор обязан предварительно создать и зарегистрировать специфичную для модуля функцию опроса (poll-функцию), используя вызов (см. <netdevice.h>):

```
static inline void netif_napi_add(struct net_device *dev,
                                  struct napi_struct *napi,
                                  int (*poll)(struct napi_struct *, int),
                                  int weight);
```

   где:

   - `dev` — это рассмотренная ранее структура зарегистрированного сетевого интерфейса;

- poll — регистрируемая функция программного опроса, о которой речь пойдет далее;
- weight — относительный вес, приоритет, который придает разработчик этому интерфейсу. Для адаптеров 10 и 100 Мб здесь часто указано значение 16, а для адаптеров 10 и 100 Гб — значение 64;
- napi — дополнительный параметр, указатель на специальную структуру, которая будет передаваться в каждый вызов функции poll, и где будет по результату выполнения этой функции заполняться поле state значениями NAPI_STATE_DISABLE или NAPI_STATE_SCHED. Вид этой структуры должен быть таким (согласно <netdevice.h>):

```
struct napi_struct {
   struct list_head poll_list;
   unsigned long state;
   int weight;
   int (*poll)(struct napi_struct *, int);
};
```

Зарегистрированная функция программного опроса (полностью зависимая от задачи и реализуемая в коде модуля) имеет подобный вид:

```
static int my_card_poll(struct napi_struct *napi, int budget) {
   int work_done; // число реально обработанных в цикле опроса
                  // сетевых пакетов
   work_done = my_card_input(budget, ...); // реализационно специфический
                                           // прием пакетов
   if(work_done < budget) {
      netif_rx_complete(netdev, napi);
      my_card_enable_irq(...);             // разрешить IRQ приема
   }
   return work_done;
}
```

Здесь пользовательская функция my_card_input() в цикле пытается аппаратно сосчитать budget сетевых пакетов, для каждого считанного сетевого пакета создает сокетный буфер и вызывает netif_receive_skb(), после чего этот буфер начинает движение по стеку протоколов вверх. Если кольцевой буфер сетевого адаптера исчерпался ранее budget пакетов (нет более наличных пакетов), то адаптеру разрешается возбуждать прерывания по приему, а ядро вызовом netif_rx_complete() уведомляется, что отменяется отложенное программное прерывание NET_RX_SOFTIRQ для дальнейшего вызова функции опроса. Если же удалось сосчитать budget пакетов (в буфере адаптера, видимо, есть еще не обработанные пакеты), то опрос продолжится при следующем цикле обработки отложенного программного прерывания NET_RX_SOFTIRQ.

2. Обработчик аппаратного прерывания линии IRQ сетевого адаптера (активирующийся при приходе *первого* сетевого пакета «пачки» активности) должен выполнять примерно следующее:

```
static irqreturn_t my_interrupt(int irq, void *dev_id) {
   struct net_device *netdev = dev_id;
   if(likely(netif_rx_schedule_prep(netdev, ...))) {
      my_card_disable_irq(...);            // запретить IRQ приема
      __netif_rx_schedule(netdev, ...);
   }
   return IRQ_HANDLED;
}
```

Здесь ядро должно быть уведомлено, что новая порция сетевых пакетов готова для обработки. Для этого:

- вызов `netif_rx_schedule_prep()` подготавливает устройство для помещения в список для программного опроса, устанавливая состояние в `NAPI_STATE_SCHED`;
- если предыдущий вызов успешен (а противное возникает только если NAPI уже активен), то вызовом `__netif_rx_schedule()` устройство помещается в список для программного опроса в цикле обработки отложенного программного прерывания `NET_RX_SOFTIRQ`.

Вот, собственно, и всё относительно новой модели приема сетевых пакетов. Здесь нужно иметь в виду, что бюджет, разово устанавливаемый в функции опроса (локальный бюджет), не должен быть чрезмерно большим. По крайней мере:

- опрос не должен потреблять более одного системного тика (глобальная переменная `jiffies`) — иначе это будет искажать диспетчеризацию потоков ядра;
- бюджет не должен быть больше глобально установленного ограничения:

```
$ cat /proc/sys/net/core/netdev_budget
300
```

- после каждого цикла опроса число обработанных пакетов (возвращаемых функцией опроса) вычитается из этого глобального бюджета, и если остаток меньше нуля, то обработчик программного прерывания `NET_RX_SOFTIRQ` останавливается.

Передача пакетов

Описанными действиями инициируется создание и движение сокетного буфера вверх по стеку. Движение же вниз (при отправке в сеть) обеспечивается по другой цепочке:

1. При инициализации сетевого интерфейса (это момент, который уже был назван ранее — см. п. 2 списка в *разд. «Прием: традиционный подход»*), создается таблица операций сетевого интерфейса, одно из полей которой — `ndo_start_xmit` — определяет функцию передачи пакета в сеть:

```
struct net_device_ops ndo = {
   .ndo_open = my_open,
   .ndo_stop = my_close,
   .ndo_start_xmit = stub_start_xmit,
};
```

2. При вызове `stub_start_xmit()` должна обеспечить аппаратную передачу полученного сокета в сеть, после чего уничтожает (возвращает в пул) буфер сокета:

```
static int stub_start_xmit(struct sk_buff *skb, struct net_device *dev) {
   // ... аппартное обслуживание передачи
   dev_kfree_skb(skb);
   return 0;
}
```

   Реально чаще уничтожение отправляемого буфера будет происходить не при инициализации операции, а при ее (успешном) завершении (аппаратном), что отслеживается *по той же линии IRQ*, что и прием пакетов из сети.

Часто задаваемый вопрос: а где же в этом процессе место (код), где реально создается содержательная информация, *помещаемая* в сокетный буфер, или где *потребляется* информация из принимаемых сокетных буферов? Ответ: не ищите такого места в пределах сетевого стека ядра — любая *информация* для отправки в сеть или потребляемая из сети возникает в поле зрения только на прикладных уровнях, в приложениях пространства пользователя — таких, например, как `ping`, `ssh`, `telnet` и великом множестве других. Интерфейс из этого прикладного уровня в стек протоколов ядра обеспечивается известным POSIX API сокетов прикладного уровня.

Статистика интерфейса

Процессы, происходящие на сетевом интерфейсе, сложно явно наблюдать (в сравнении, скажем, с интерфейсами /dev или /proc). Поэтому очень важной характеристикой интерфейса становится накопленная статистика происходящих на нем процессов. Для накопления статистики работы сетевого интерфейса описана специальная структура (достаточно большая, она определена там же — в <linux/netdevice.h>, а здесь показано только начало структуры) :

```
struct net_device_stats {
   unsigned long  rx_packets;     /* total packets received    */
   unsigned long  tx_packets;     /* total packets transmitted */
   unsigned long  rx_bytes;       /* total bytes received      */
   unsigned long  tx_bytes;       /* total bytes transmitted   */
   unsigned long  rx_errors;      /* bad packets received      */
   unsigned long  tx_errors;      /* packet transmit problems  */
...
}
```

Поля такой структуры должны заполняться самим *кодом модуля* статистическими данными проходящих пакетов (например, инкрементируя `tx_packets` при передаче пакета).

Это старая схема создания статистики. В новой схеме (в новых драйверах) статистика должна накапливаться в *новой* подобной структуре (определена в include/uapi/linux/if_link.h):

```
struct rtnl_link_stats64 {
        __u64   rx_packets;            /* total packets received       */
```

```
        __u64   tx_packets;             /* total packets transmitted    */
        __u64   rx_bytes;               /* total bytes received         */
        __u64   tx_bytes;               /* total bytes transmitted      */
        __u64   rx_errors;              /* bad packets received         */
...
```

Возвращается таблица (та или иная) накопленной статистики (в том числе и для передачи в пространство пользователя — см. результат команды `ifconfig`) функцией `ndo_get_stats` или `ndo_get_stats64` в *таблице операций* интерфейса `struct net_device_ops` — ранее эти поля были специально показаны:

```
        ...
        struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
        void                     (*ndo_get_stats64)(struct net_device *dev,
                                  struct rtnl_link_stats64 *storage);
        ...
```

Как легко видеть, способы возврата результата в двух этих вариантах существенно различаются: старый возвращал указатель на существующую структуру, а новый получает указатель на размещенную в памяти структуру, которую он заполняет значениями.

Модуль должен реализовать такие собственные функции (одну из них или обе) и поместить их в таблицу операций `struct net_device_ops`. Это делается, если вы хотите получать статистику сетевого интерфейса пользователя вызовом `ifconfig` или через интерфейс файловой системы /proc, как это ожидаемо и происходит для всех сетевых интерфейсов:

```
$ ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr 00:13:02:69:70:9B
          inet addr:192.168.1.22  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::213:2ff:fe69:709b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8658 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9070 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4240425 (4.0 MiB)  TX bytes:1318733 (1.2 MiB)
```

Начнем со старого метода... Где обычно размещается структура `net_device_stats`, которую мы предполагаем возвращать пользователю? Часто встречается несколько вариантов:

1. Если модуль обслуживает только один конкретный сетевой интерфейс, то структура может размещаться на глобальном уровне кода модуля:

```
static struct net_device_stats stats = {};
...
static struct net_device_stats *my_get_stats(struct net_device *dev) {
   return &stats;
}
...
```

```
static struct net_device_ops ndo = {
   ...
   .ndo_get_stats = my_get_stats,
};
```

2. Часто структура статистики размещается в качестве составной части (или просто вместо) структуры *приватных данных* (о ней речь шла ранее), которую разработчик связывает с сетевым интерфейсом:

```
static struct net_device *my_dev = NULL;
struct my_private {
   struct net_device_stats stats;
   ...
};
...
static struct net_device_stats *my_get_stats(struct net_device *dev) {
   struct my_private *priv = (my_private*)netdev_priv(dev);
   return &priv->stats;
}
...
static struct net_device_ops ndo = {
   ...
   .ndo_get_stats = my_get_stats,
}
...
void my_setup(struct net_device *dev) {
   memset(netdev_priv(dev), 0, sizeof(struct my_private));
   dev->netdev_ops = &ndo;
}

int __init my_init(void) {  // инициализация
    my_dev = alloc_netdev(sizeof(struct my_private), "my_if%d", my_setup);
}
```

3. Наконец, может использоваться структура, включенная (имплементированная) непосредственно *в состав* определения интерфейса `struct net_device`:

```
...
static struct net_device_stats *my_get_stats(struct net_device *dev) {
   return &dev->stats;
}
```

Все эти три варианта использования приведены (для сравнения: файлы virt.c, virt1.c и virt2.c) в папке network/virt сопровождающего книгу файлового архива.

Для новой схемы накопления статистики (`ndo_get_stats64`) также могут использоваться схемы 1 или 2, но при возврате результата эта структура должна копироваться по указателю вызова.

Внимание!

Очень важно учитывать, что в таблице операций сетевого интерфейса (`struct net_device_ops`) одно из полей (да и оба тоже) указателей функций (`ndo_get_stats` и `ndo_get_stats64`) часто бывает нулевым[6], поэтому, запрашивая статистику у драйвера, прежде разыменования указателя (вызова функции метода), проверяйте его на `NULL` — в противном случае грозит крах системы!

Виртуальный сетевой интерфейс

В предыдущих примерах мы создавали сетевые интерфейсы, но они *не осуществляли* реально с физической средой ни передачи, ни приема. Для выполнения такого уровня проработки нужно было бы иметь реальное коммуникационное оборудование на шине PCI и рассматривать операции с ним. Это не всегда доступно, но, главное, это сильно загрузило бы рассмотрение массой частных деталей конкретного оборудования (а мы стараемся уйти от сложных гипотетических моделей).

Однако мы можем создать интерфейс, который будет *перехватывать* трафик сетевого ввода/вывода с другого, реально существующего в системе интерфейса, и обеспечивать обработку этих потоков (приведенные в папке network/virt сопровождающего книгу файлового архива файлы virt.c, virt1.c, virt2.c различаются между собой только способом сбора статистики, и вы можете экспериментировать с ними в равной степени):

virt.c

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/moduleparam.h>
#include <net/arp.h>

#define ERR(...) printk(KERN_ERR "! "__VA_ARGS__)
#define LOG(...) printk(KERN_INFO "! "__VA_ARGS__)

static char* link = "eth0";
module_param(link, charp, 0);

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

static struct net_device *child = NULL;

struct priv {
   struct net_device_stats stats;
```

[6] В новых существующих драйверах `ndo_get_stats` зачастую бывает `NULL`.

```
   struct net_device *parent;
};

static rx_handler_result_t handle_frame(struct sk_buff **pskb) {
   struct sk_buff *skb = *pskb;
   if(child) {
      struct priv *priv = netdev_priv(child);
      priv->stats.rx_packets++;
      priv->stats.rx_bytes += skb->len;
      LOG("rx: injecting frame from %s to %s", skb->dev->name, child->name);
      skb->dev = child;
      /* netif_receive_skb(skb);
      return RX_HANDLER_CONSUMED; */
      return RX_HANDLER_ANOTHER;
   }
   return RX_HANDLER_PASS;
}

static int open(struct net_device *dev) {
   netif_start_queue(dev);
   LOG("%s: device opened", dev->name);
   return 0;
}

static int stop(struct net_device *dev) {
   netif_stop_queue(dev);
   LOG("%s: device closed", dev->name);
   return 0;
}

static netdev_tx_t start_xmit(struct sk_buff *skb, struct net_device *dev) {
   struct priv *priv = netdev_priv(dev);
   priv->stats.tx_packets++;
   priv->stats.tx_bytes += skb->len;
   if(priv->parent) {
      skb->dev = priv->parent;
      skb->priority = 1;
      dev_queue_xmit(skb);
      LOG("tx: injecting frame from %s to %s", dev->name, skb->dev->name);
      return 0;
   }
   return NETDEV_TX_OK;
}

static struct net_device_stats *get_stats(struct net_device *dev) {
   return &((struct priv*)netdev_priv(dev))->stats;
}
```

```
static struct net_device_ops crypto_net_device_ops = {
   .ndo_open = open,
   .ndo_stop = stop,
   .ndo_get_stats = get_stats,
   .ndo_start_xmit = start_xmit,
};

// #define MAX_ADDR_LEN    32  <netdev.h>
// #define ETH_ALEN        6   /* Octets in one ethernet addr   */ <if_ether.h>

static void setup(struct net_device *dev) {
   int j;
   ether_setup(dev);
   memset(netdev_priv(dev), 0, sizeof(struct priv));
   dev->netdev_ops = &crypto_net_device_ops;
   for(j = 0; j < ETH_ALEN; ++j) // fill in the MAC address with a phoney
      dev->dev_addr[j]= (char)j;
}

int __init init(void) {
   int err = 0;
   struct priv *priv;
   char ifstr[40];
   sprintf(ifstr, "%s%s", ifname, "%d");
#if (LINUX_VERSION_CODE < KERNEL_VERSION(3, 17, 0))
   child = alloc_netdev(sizeof(struct priv), ifstr, setup);
#else
   child = alloc_netdev(sizeof(struct priv), ifstr, NET_NAME_UNKNOWN, setup);
#endif
   if(child == NULL) {
      ERR("%s: allocate error", THIS_MODULE->name); return -ENOMEM;
   }
   priv = netdev_priv(child);
   priv->parent = __dev_get_by_name(&init_net, link); // parent interface
   if(!priv->parent) {
      ERR("%s: no such net: %s", THIS_MODULE->name, link);
      err = -ENODEV; goto err;
   }
   if(priv->parent->type != ARPHRD_ETHER && priv->parent->type != ARPHRD_LOOPBACK) {
      ERR("%s: illegal net type", THIS_MODULE->name);
      err = -EINVAL; goto err;
   }
   /* also, and clone its IP, MAC and other information */
   memcpy(child->dev_addr, priv->parent->dev_addr, ETH_ALEN);
   memcpy(child->broadcast, priv->parent->broadcast, ETH_ALEN);
```

```
   if((err = dev_alloc_name(child, child->name))) {
      ERR("%s: allocate name, error %i", THIS_MODULE->name, err);
      err = -EIO; goto err;
   }
   register_netdev(child);
   rtnl_lock();
   netdev_rx_handler_register(priv->parent, &handle_frame, NULL);
   rtnl_unlock();
   LOG("module %s loaded", THIS_MODULE->name);
   LOG("%s: create link %s", THIS_MODULE->name, child->name);
   LOG("%s: registered rx handler for %s", THIS_MODULE->name, priv->parent->name);
   return 0;
err:
   free_netdev(child);
   return err;
}

void __exit virt_exit(void) {
   struct priv *priv = netdev_priv(child);
   if(priv->parent) {
      rtnl_lock();
      netdev_rx_handler_unregister(priv->parent);
      rtnl_unlock();
      LOG("unregister rx handler for %s\n", priv->parent->name);
   }
   unregister_netdev(child);
   free_netdev(child);
   LOG("module %s unloaded", THIS_MODULE->name);
}

module_init(init);
module_exit(virt_exit);

MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_AUTHOR("Nikita Dorokhin");
MODULE_LICENSE("GPL v2");
MODULE_VERSION("3.2");
```

Перехват *входящего* трафика родительского интерфейса здесь осуществляется установкой нового обработчика (функция `handle_frame()`) входящих пакетов для созданного интерфейса вызовом `netdev_rx_handler_unregister()`, который появился в API ядра, начиная с 2.6.36 (ранее это приходилось делать другими способами). При передаче каждого *исходящего* сокетного буфера в сеть (функция `start_xmit()`) мы просто подменяем в структуре сокетного буфера интерфейс, через который физически должна производиться отправка. Работа с таким интерфейсом выглядит примерно следующим образом:

- на *любой* существующий и работоспособный существующий сетевой интерфейс:

```
$ ip a s
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
    link/ether 90:b1:1c:54:3a:46 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.14/24 brd 192.168.1.255 scope global noprefixroute eno1
       valid_lft forever preferred_lft forever
3: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
    link/ether 90:b1:1c:54:3a:47 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.13/24 brd 192.168.1.255 scope global noprefixroute eno2
       valid_lft forever preferred_lft forever
4: team1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group
default qlen 1000
    link/ether ee:8b:90:64:9b:dd brd ff:ff:ff:ff:ff:ff
```

- устанавливаем новый виртуальный интерфейс и конфигурируем его — «навешиваем» виртуальный интерфейс на физический:

```
$ sudo insmod virt.ko link=eno2
$ ip l
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group
default qlen 1000
    link/ether 90:b1:1c:54:3a:46 brd ff:ff:ff:ff:ff:ff
3: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group
default qlen 1000
    link/ether 90:b1:1c:54:3a:47 brd ff:ff:ff:ff:ff:ff
4: team1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode
DEFAULT group default qlen 1000
    link/ether ee:8b:90:64:9b:dd brd ff:ff:ff:ff:ff:ff
10: virt0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN mode
DEFAULT group default qlen 1000
    link/ether 90:b1:1c:54:3a:47 brd ff:ff:ff:ff:ff:ff
```

- конфигурируем новый интерфейс и запускаем:

```
$ sudo ifconfig virt0 192.168.1.240 up
$ sudo ifconfig virt0
virt0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.240  netmask 255.255.255.0  broadcast 192.168.1.255
        ether 90:b1:1c:54:3a:47  txqueuelen 1000  (Ethernet)
```

```
        RX packets 56  bytes 8513 (8.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 35  bytes 4011 (4.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- проверяем прозрачность интерфейса `virt0`:

```
$ ping -I virt0 -c3 192.168.1.3
PING 192.168.1.3 (192.168.1.3) from 192.168.1.240 virt0: 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=4.92 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=4.85 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=4.25 ms

--- 192.168.1.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 4.250/4.671/4.917/0.299 ms
```

- ... и то, что родительский интерфейс стал недоступным (неработоспособным):

```
$ ping -I eno2 -c3 192.168.1.3
PING 192.168.1.3 (192.168.1.3) from 192.168.1.13 eno2: 56(84) bytes of data.

--- 192.168.1.3 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2032ms
```

- выполняем соединение SSH через новый интерфейс в LANL:

```
$ ssh -B virt0 olej@192.168.1.142
olej@192.168.1.142's password:
Linux raspberrypi 5.15.32-v7+ #1538 SMP Thu Mar 31 19:38:48 BST 2022 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Jun 19 21:12:40 2022 from 192.168.1.200
olej@raspberrypi:~ $ inxi -CSxxx
System:    Host: raspberrypi Kernel: 5.15.32-v7+ armv7l bits: 32 compiler: gcc v: 10.2.1
Console: tty 2 DM: LightDM 1.26.0
           Distro: Raspbian GNU/Linux 11 (bullseye)
CPU:       Info: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32 type: MCP arch: v7l
                                                                                  rev: 5
           features: Use -f option to see features bogomips: 0
           Speed: 700 MHz min/max: 600/1000 MHz Core speeds (MHz): 1: 700 2: 700 3: 700 4: 700
olej@raspberrypi:~ $ uname -a
Linux raspberrypi 5.15.32-v7+ #1538 SMP Thu Mar 31 19:38:48 BST 2022 armv7l GNU/Linux
olej@raspberrypi:~ $ exit
выход
Connection to 192.168.1.142 closed.
```

Мы вошли через новый интерфейс в ARM-компьютер... (и вышли в заключение);

- и проверяем статистику трафика (RX, TX):

```
$ sudo ifconfig virt0
virt0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.240  netmask 255.255.255.0  broadcast 192.168.1.255
        ether 90:b1:1c:54:3a:47  txqueuelen 1000  (Ethernet)
        RX packets 411  bytes 42993 (42.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 328  bytes 34030 (34.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- выгружаем модуль и убеждаемся, что работа родительского интерфейса возобновилась:

```
$ sudo rmmod virt
$ ping -I eno2 -c3 192.168.1.3
PING 192.168.1.3 (192.168.1.3) from 192.168.1.13 eno2: 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=9.88 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=3.57 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=3.53 ms

--- 192.168.1.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 3.530/5.661/9.883/2.984 ms
```

> ***Примечание***
>
> Чтобы не увеличивать сложность обсуждения, здесь показан упрощенный пример модуля, который полностью перехватывает трафик родительского интерфейса и при этом замещает родительский интерфейс (для этого нужно раздельно обрабатывать пакеты IP и пакеты разрешения адресов ARP). В папке network/virt-full сопровождающего книгу файлового архива приведен пример, который корректно анализирует адреса получателей и сохраняет независимую работу родительского интерфейса. Но чтобы не загромождать обсуждение деталями, те, кто интересуются, могут поэкспериментировать с этим работающим полным виртуальным интерфейсом самостоятельно.

Виртуальный сетевой интерфейс (созданный тем или иным способом) — это очень мощный инструмент периода создания и отладки сетевых модулей, поэтому он заслуживает отдельного рассмотрения.

Протокол сетевого уровня

На этом уровне (L3) обеспечивается обработка таких протоколов, как: IP/IPv4/IPv6, IPX, ICMP, RIP, OSPF, ARP, или добавление оригинальных пользовательских протоколов. Для установки обработчиков сетевого уровня предоставляется API сетевого уровня (<linux/netdevice.h>):

```
struct packet_type {
   __be16  type; /* This is really htons(ether_type). */
```

```
   struct net_device *dev; /* NULL is wildcarded here    */
   int (*func) (struct sk_buff *, struct net_device *, struct packet_type *, struct net_device *);
...
   struct list_head list;
};
extern void dev_add_pack(struct packet_type *pt);
extern void dev_remove_pack(struct packet_type *pt);
```

Фактически в протокольных модулях — как здесь, так и далее на транспортном уровне — мы должны *добавить фильтр*, через который проходят буферы сокетов *из входящего потока* интерфейса (исходящий поток реализуется проще, как показано в примерах, приведенных ранее). Функция `dev_add_pack()` добавляет еще один новый обработчик для пакетов заданного типа, реализуемый функцией `func()`. При этом функция *добавляет, но не замещает* существующий обработчик (в том числе и обработчик по умолчанию сетевой системы Linux). На обработку в функцию отбираются (попадают) те буферы сокетов, которые удовлетворяют критериям, заложенным в структуре `struct packet_type` (по типу протокола `type` сетевого интерфейса `dev`).

Примеры добавления собственных обработчиков сетевых протоколов содержатся в папке network/netproto сопровождающего книгу файлового архива. Вот так может быть добавлен обработчик нового протокола сетевого уровня:

net_proto.c

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

#define ERR(...) printk(KERN_ERR "! "__VA_ARGS__)
#define LOG(...) printk(KERN_INFO "! "__VA_ARGS__)

int test_pack_rcv(struct sk_buff *skb, struct net_device *dev,
                  struct packet_type *pt, struct net_device *odev) {
   LOG("packet received with length: %u\n", skb->len);
   kfree_skb(skb);
   return skb->len;
};

#define TEST_PROTO_ID 0x1234
static struct packet_type test_proto = {
   __constant_htons(ETH_P_ALL),  // may be: __constant_htons(TEST_PROTO_ID),
   NULL,
   test_pack_rcv,
   (void*)1,
   NULL
};
```

```
static int __init my_init(void) {
   dev_add_pack(&test_proto);
   LOG("module loaded\n");
   return 0;
}

static void __exit my_exit(void) {
   dev_remove_pack(&test_proto);
   LOG(KERN_INFO "module unloaded\n");
}

module_init(my_init);
module_exit(my_exit);

MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_LICENSE("GPL v2");
```

> ***Примечание***
>
> Самая большая сложность с подобными примерами — это то, какими средствами вы будете их тестировать для нестандартного протокола, когда операционная система, возможно, не знает такого сетевого протокола и не имеет утилит обмена в таком протоколе...

Выполнение такого примера:

```
$ sudo insmod net_proto.ko
$ dmesg | tail -n6
module loaded
packet received with length: 74
packet received with length: 60
packet received with length: 66
packet received with length: 241
packet received with length: 52
$ sudo rmmod net_proto
```

В этом примере обработчик протокола перехватывает (фильтрует) **все** пакеты (константа `ETH_P_ALL`) на всех сетевых интерфейсах. В случае собственного протокола здесь должна бы быть константа `TEST_PROTO_ID` (но для такого случая нам нечем оттестировать модуль). Очень большое число идентификаторов протоколов (Ethernet Protocol ID's) можно найти в <linux/if_ether.h> — вот 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 мы находим описание заголовка Ethernet-пакета, который помогает в заполнении структуры `struct packet_type`:

```
struct ethhdr {
   unsigned char h_dest[ETH_ALEN];   /* destination eth addr */
   unsigned char h_source[ETH_ALEN]; /* source ether addr    */
   __be16       h_proto;             /* packet type ID field */
} __attribute__((packed));
```

Этот пример порождает ряд вопросов:

- можно ли установить несколько обработчиков потока пакетов (для одного или различающихся типов протоколов `type`);
- в каком порядке будут вызываться функции-обработчики при поступлении пакета;
- как специфицировать один отдельный сетевой интерфейс, к которому должен применяться фильтр;
- какую роль играет вызов `kfree_skb()` (в функции-фильтре обработки протокола `test_pack_rcv()`) и какой сокетный буфер (или его копия) при этом освобождается?

Для уяснения этих вопросов нам необходимо собрать на базе предыдущего другой пример (показаны только различающиеся фрагменты кода):

net_proto2.c

```
...
static int debug = 0;
module_param(debug, int, 0);

static char* link = NULL;
module_param(link, charp, 0);

int test_pack_rcv_1(struct sk_buff *skb, struct net_device *dev,
                    struct packet_type *pt, struct net_device *odev) {
   int s = atomic_read(&skb->users);
   kfree_skb(skb);
   if(debug > 0)
      LOG("function #1 - %p => users: %d->%d\n", skb, s, atomic_read(&skb->users));
   return skb->len;
};

int test_pack_rcv_2(struct sk_buff *skb, struct net_device *dev,
                    struct packet_type *pt, struct net_device *odev) {
   int s = atomic_read(&skb->users);
```

```
   kfree_skb(skb);
   if(debug > 0)
      LOG("function #2 - %p => users: %d->%d\n", skb, s, atomic_read(&skb->users));
   return skb->len;
};

static struct packet_type
test_proto1 = {
   __constant_htons(ETH_P_IP),
   NULL,
   test_pack_rcv_1,
   (void*)1,
   NULL
},
test_proto2 = {
   __constant_htons(ETH_P_IP),
   NULL,
   test_pack_rcv_2,
   (void*)1,
   NULL
};

static int __init my_init(void) {
   if(link != NULL) {
      struct net_device *dev = __dev_get_by_name(&init_net, link);
      if(NULL == dev) {
         ERR("%s: illegal link", link);
         return -EINVAL;
      }
      test_proto1.dev = test_proto2.dev = dev;
   }
   dev_add_pack(&test_proto1);
   dev_add_pack(&test_proto2);
   if(NULL == test_proto1.dev) LOG("module %s loaded for all links\n", THIS_MODULE->name);
   else LOG("module %s loaded for link %s\n", THIS_MODULE->name, link);
   return 0;
}

static void __exit my_exit(void) {
   if(test_proto2.dev != NULL) dev_put(test_proto2.dev);
   if(test_proto1.dev != NULL) dev_put(test_proto1.dev);
   dev_remove_pack(&test_proto2);
   dev_remove_pack(&test_proto1);
   LOG("module %s unloaded\n", THIS_MODULE->name);
}
```

Здесь, собственно, выполняется абсолютно то же, что и в предыдущем варианте, но мы устанавливаем одновременно *два* новых дополнительных обработчика для

пакетов IPv4 (ETH_P_IP), причем устанавливаем именно в последовательности: test_pack_rcv_1(), а затем: test_pack_rcv_2(), и смотрим, что из этого получается... Загружаем модуль:

```
$ sudo insmod net_proto2.ko link=p7p1
$ dmesg | tail -n1
[  403.339591] ! module net_proto2 loaded for link p7p1
```

И далее (конфигурировав интерфейс на адрес 192.168.56.3) с другого хоста выполняем:

```
$ ping -c3 192.168.56.3
PING 192.168.56.3 (192.168.56.3) 56(84) bytes of data.
64 bytes from 192.168.56.3: icmp_req=1 ttl=64 time=0.668 ms
64 bytes from 192.168.56.3: icmp_req=2 ttl=64 time=0.402 ms
64 bytes from 192.168.56.3: icmp_req=3 ttl=64 time=0.330 ms
--- 192.168.56.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.330/0.466/0.668/0.147 ms
```

В системном журнале мы найдем:

```
$ dmesg | tail -n7
[  403.339591] ! module net_proto2 loaded for link p7p1
[  420.305824] ! function #2 - eb6873c0 => users: 2->1
[  420.305829] ! function #1 - eb6873c0 => users: 2->1
[  421.306302] ! function #2 - eb687c00 => users: 2->1
[  421.306308] ! function #1 - eb687c00 => users: 2->1
[  422.306289] ! function #2 - eb687180 => users: 2->1
[  422.306294] ! function #1 - eb687180 => users: 2->1
```

Отсюда видно, что обработчики срабатывают в порядке, обратном их установке (установленный позже срабатывает раньше), но срабатывают *все*. Они получают в качестве аргумента адрес одной и той же копии буфера сокета. Относительно kfree_skb() мы должны обратиться к исходному коду реализации ядра (файл net/core/skbuff.c):

```
void kfree_skb(struct sk_buff *skb) {
   if (unlikely(!skb))
      return;
   if (likely(atomic_read(&skb->users) == 1))
      smp_rmb();
   else if (likely(!atomic_dec_and_test(&skb->users)))
      return;
   trace_kfree_skb(skb, __builtin_return_address(0));
   __kfree_skb(skb);
}
```

Вызов kfree_skb() будет реально освобождать буфер сокета только в случае skb->users == 1, при всех остальных значениях он будет только декрементировать skb->users (счетчик использования). Для уточнения происходящего мы можем за-

комментировать вызовы `kfree_skb()` в двух приведенных ранее обработчиках и наблюдать при этом:

```
$ dmesg | tail -n7
[11373.754524] ! module net_proto2 loaded for link p7p1
[11398.930057] ! function #2 - ed3dfc00 => users: 2->2
[11398.930061] ! function #1 - ed3dfc00 => users: 3->3
[11399.929838] ! function #2 - ed3dfb40 => users: 2->2
[11399.929843] ! function #1 - ed3dfb40 => users: 3->3
[11400.929522] ! function #2 - ed3df480 => users: 2->2
[11400.929527] ! function #1 - ed3df480 => users: 3->3
```

Если функция-обработчик не декрементирует `skb->users` по завершении вызовом `kfree_skb()`, то после окончательного вызова обработки по умолчанию буфер сокета не будет уничтожен и в системе будет наблюдаться *утечка памяти*. Она незначительная, но неумолимая, и в конце концов приведет к краху системы. Проверку на отсутствие утечки памяти (по этой причине или по любой иной) предлагается проверить приемом по сетевому каналу значительного объема данных (несколько гигабайт) с фиксацией состояния памяти командой `free`. Для этого можно с успехом использовать утилиту `nc` (network cat):

1. На тестируемом узле запустить скрипт ./client:

```
LIMIT=1000000
for ((a=1; a <= LIMIT ; a++))
do
   rm -a z.txt
   nc 192.168.56.1 12345 > z.txt
   sleep 1
done
```

2. На стороннем узле сети запустить скрипт ./server:

```
dd if=/dev/zero of=z.txt bs=1M count=1
LIMIT=1000000
for ((a=1; a <= LIMIT ; a++))
do
   cat z.txt | nc -l 12345
done
```

   Здесь в скриптах: `192.168.56.1` — IP-адрес узла, где работает `server`, `12345` — номер TCP порта, через который `nc` предписывается передавать данные. При желании можно использовать и UDP, добавив к `nc` опцию `-u`.

3. И подождать некоторое время — порядка 10–20 минут:

```
$ free
             total       used       free     shared    buffers     cached
Mem:        768084     260636     507448          0      23392     129108
-/+ buffers/cache:     108136     659948
Swap:      1540092          0    1540092
```

```
$ ifconfig p7p1
p7p1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.56.3  netmask 255.255.255.255  broadcast 192.168.56.3
        inet6 fe80::a00:27ff:fe08:9abd  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:08:9a:bd  txqueuelen 1000  (Ethernet)
        RX packets 2017560  bytes 3048762624 (2.8 GiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 398156  bytes 26283474 (25.0 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Еще раз о виртуальном интерфейсе...

Ранее рассматривалось создание виртуального сетевого интерфейса (использующего трафик реального физического, родительского) средствами `netdev_rx_handler_register()`. Но такой способ не свободен от ряда недостатков: он появился хронологически достаточно поздно — в API ядра, начиная с версии 2.6.36, и не со всяким ядром может быть использован. Кроме того, он в некоторой степени громоздкий и путанный. Но это же можно сделать и другим способом, не зависящим от версий используемого ядра, задействовав протокольные механизмы, но уже *сетевого* уровня (L3). Кроме того, это хороший реалистичный пример использования протокольных фильтров. Примеры подобной реализации расположены в папках network/.../virt-proto сопровождающего книгу файлового архива.

> ***Примечание***
>
> В указанном архиве представлены два варианта модуля: упрощенный модуль virtl.ko (lite-вариант), интерфейс (`virt0`) которого замещает родительский сетевой интерфейс, и полный вариант virt.ko, который анализирует сетевые фреймы (и ARP- и IP4-протоколов) и затрагивает только трафик, к его интерфейсу относящийся. Разница состоит в том, что на время загрузки упрощенного модуля работа родительского интерфейса прекращается, а при загрузке полного варианта оба интерфейса работают независимо. Но код полного модуля гораздо более громоздкий, а для понимания принципов он ничего не добавляет. Далее детально рассмотрен упрощенный вариант, не скрывающий принципы, и только позже мы в пару слов коснемся полного варианта: код его и протокол испытаний приведены в архиве, поэтому его детализация не вызывает сложностей.

virtl.c :

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/inetdevice.h>
#include <linux/moduleparam.h>
#include <net/arp.h>
#include <linux/ip.h>
```

```
#define ERR(...) printk(KERN_ERR "! "__VA_ARGS__)
#define LOG(...) printk(KERN_INFO "! "__VA_ARGS__)
#define DBG(...) if(debug != 0) printk(KERN_INFO "! "__VA_ARGS__)

static char* link = "eth0";
module_param(link, charp, 0);

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

static int debug = 0;
module_param(debug, int, 0);

static struct net_device *child = NULL;
static struct net_device_stats stats;  // статическая таблица
                                       // статистики интерфейса
static u32 child_ip;

struct priv {
   struct net_device *parent;
};

static char* strIP(u32 addr) {      // диагностика IP в точечной нотации
   static char saddr[ MAX_ADDR_LEN ];
   sprintf(saddr, "%d.%d.%d.%d",
            (addr) & 0xFF, (addr >> 8) & 0xFF,
            (addr >> 16) & 0xFF, (addr >> 24) & 0xFF
          );
   return saddr;
}

static int open(struct net_device *dev) {
   struct in_device *in_dev = dev->ip_ptr;
   struct in_ifaddr *ifa = in_dev->ifa_list;      /* IP ifaddr chain */
   LOG("%s: device opened", dev->name);
   child_ip = ifa->ifa_address;
   netif_start_queue(dev);
   if(debug != 0) {
      char sdebg[ 40 ] = "";
      sprintf(sdebg, "%s:", strIP(ifa->ifa_address));
      strcat(sdebg, strIP(ifa->ifa_mask));
      DBG("%s: %s", dev->name, sdebg);
   }
   return 0;
}

static int stop(struct net_device *dev) {
   LOG("%s: device closed", dev->name);
```

```
   netif_stop_queue(dev);
   return 0;
}

static struct net_device_stats *get_stats(struct net_device *dev) {
   return &stats;
}

// передача фрейма
static netdev_tx_t start_xmit(struct sk_buff *skb, struct net_device *dev) {
   struct priv *priv = netdev_priv(dev);
   stats.tx_packets++;
   stats.tx_bytes += skb->len;
   skb->dev = priv->parent;   // передача в родительский (физический) интерфейс
   skb->priority = 1;
   dev_queue_xmit(skb);
   DBG("tx: injecting frame from %s to %s with length: %u",
        dev->name, skb->dev->name, skb->len);
   return 0;
}

static struct net_device_ops net_device_ops = {
   .ndo_open = open,
   .ndo_stop = stop,
   .ndo_get_stats = get_stats,
   .ndo_start_xmit = start_xmit,
};

// прием фрейма
int pack_parent(struct sk_buff *skb, struct net_device *dev,
                struct packet_type *pt, struct net_device *odev) {
   skb->dev = child;          // передача фрейма в виртуальный интерфейс
   stats.rx_packets++;
   stats.rx_bytes += skb->len;
   DBG("tx: injecting frame from %s to %s with length: %u",
        dev->name, skb->dev->name, skb->len);
   kfree_skb(skb);
   return skb->len;
};

static struct packet_type proto_parent = {
   __constant_htons(ETH_P_ALL), // перехватывать все пакеты: ETH_P_ARP & ETH_P_IP
   NULL,
   pack_parent,
   (void*)1,
   NULL
};
```

```
int __init init(void) {
   void setup(struct net_device *dev) { // вложенная функция (GCC)
      int j;
      ether_setup(dev);
      memset(netdev_priv(dev), 0, sizeof(struct priv));
      dev->netdev_ops = &net_device_ops;
      for(j = 0; j < ETH_ALEN; ++j)     // заполнить MAC фиктивным адресом
         dev->dev_addr[ j ] = (char)j;
   }
   int err = 0;
   struct priv *priv;
   char ifstr[ 40 ];
   sprintf(ifstr, "%s%s", ifname, "%d");
#if (LINUX_VERSION_CODE < KERNEL_VERSION(3, 17, 0))
   child = alloc_netdev(sizeof(struct priv), ifstr, setup);
#else
   child = alloc_netdev(sizeof(struct priv), ifstr, NET_NAME_UNKNOWN, setup);
#endif
   if(child == NULL) {
      ERR("%s: allocate error", THIS_MODULE->name); return -ENOMEM;
   }
   priv = netdev_priv(child);
   priv->parent = __dev_get_by_name(&init_net, link); // родительский интерфейс
   if(!priv->parent) {
      ERR("%s: no such net: %s", THIS_MODULE->name, link);
      err = -ENODEV; goto err;
   }
   if(priv->parent->type != ARPHRD_ETHER && priv->parent->type != ARPHRD_LOOPBACK) {
      ERR("%s: illegal net type", THIS_MODULE->name);
      err = -EINVAL; goto err;
   }
   memcpy(child->dev_addr, priv->parent->dev_addr, ETH_ALEN);
   memcpy(child->broadcast, priv->parent->broadcast, ETH_ALEN);
   if((err = dev_alloc_name(child, child->name))) {
      ERR("%s: allocate name, error %i", THIS_MODULE->name, err);
      err = -EIO; goto err;
   }
   register_netdev(child);         // зарегистрировать новый интерфейс
   proto_parent.dev = priv->parent;
   dev_add_pack(&proto_parent);    // установить обработчик фреймов для родителя
   LOG("module %s loaded", THIS_MODULE->name);
   LOG("%s: create link %s", THIS_MODULE->name, child->name);
   return 0;
err:
   free_netdev(child);
   return err;
}
```

```
void __exit exit(void) {
   dev_remove_pack(&proto_parent); // удалить обработчик фреймов
   unregister_netdev(child);
   free_netdev(child);
   LOG("module %s unloaded", THIS_MODULE->name);
   LOG("============================================");
}

module_init(init);
module_exit(exit);

MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_LICENSE("GPL v2");
MODULE_VERSION("3.6");
```

От рассмотренных ранее примеров код отличается только тем, что после регистрации нового сетевого интерфейса (`virt0`) он выполняет вызов `dev_add_pack()`, предварительно установив в структуре `packet_type` поле `dev` на указатель родительского интерфейса, — только с этого интерфейса входящий трафик будет перехватываться определенной в структуре функцией `pack_parent()`. Эта функция фиксирует статистику интерфейса и, самое главное, *подменяет* в сокетном буфере указатель родительского интерфейса на виртуальный. Обратная подмена (виртуального на физический) происходит в функции отправки фрейма `start_xmit()`, но это не отличается от того, что мы видели ранее. Вот как это работает:

1. На *тестируемом* компьютере загружаем модуль и конфигурируем его:

```
$ ip address
...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:52:b9:e0 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.21/24 brd 192.168.1.255 scope global eth0
    inet6 fe80::a00:27ff:fe52:b9e0/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:0f:13:6d brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.102/24 brd 192.168.56.255 scope global eth1
    inet6 fe80::a00:27ff:fe0f:136d/64 scope link
       valid_lft forever preferred_lft forever
$ sudo insmod ./virt.ko link=eth1 debug=1
$ sudo ifconfig virt0 192.168.50.19
$ sudo ifconfig virt0
virt0     Link encap:Ethernet  HWaddr 08:00:27:0f:13:6d
          inet addr:192.168.50.19  Bcast:192.168.50.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe0f:136d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:46 errors:0 dropped:0 overruns:0 carrier:0
```

```
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:8373 (8.1 KiB)
```

Показана статистика с нулевым числом принятых байтов на интерфейсе.

2. На *тестирующем* компьютере создаем алиасный IP для тестируемой подсети (для *другой* от основной подсети: `192.168.50.0/24`) и можем осуществлять трафик на созданный интерфейс:

```
$ sudo ifconfig vboxnet0:1 192.168.50.1
$ ping 192.168.50.19
PING 192.168.50.19 (192.168.50.19) 56(84) bytes of data.
64 bytes from 192.168.50.19: icmp_req=1 ttl=64 time=0.627 ms
64 bytes from 192.168.50.19: icmp_req=2 ttl=64 time=0.305 ms
64 bytes from 192.168.50.19: icmp_req=3 ttl=64 time=0.326 ms
^C
--- 192.168.50.19 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.305/0.419/0.627/0.148 ms
```

3. На этом же (тестирующем) компьютере (ответной стороне) очень содержательно наблюдать трафик (в отдельном терминале), фиксируемый `tcpdump`:

```
$ sudo tcpdump -i vboxnet0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on vboxnet0, link-type EN10MB (Ethernet), capture size 65535 bytes
...
18:41:01.740607 ARP, Request who-has 192.168.50.19 tell 192.168.50.1, length 28
18:41:01.741104 ARP, Reply 192.168.50.19 is-at 08:00:27:0f:13:6d (oui Unknown), length 28
18:41:01.741116 IP 192.168.50.1 > 192.168.50.19: ICMP echo request, id 8402, seq 1, length 64
18:41:01.741211 IP 192.168.50.19 > 192.168.50.1: ICMP echo reply, id 8402, seq 1, length 64
18:41:02.741164 IP 192.168.50.1 > 192.168.50.19: ICMP echo request, id 8402, seq 2, length 64
18:41:02.741451 IP 192.168.50.19 > 192.168.50.1: ICMP echo reply, id 8402, seq 2, length 64
18:41:03.741163 IP 192.168.50.1 > 192.168.50.19: ICMP echo request, id 8402, seq 3, length 64
18:41:03.741471 IP 192.168.50.19 > 192.168.50.1: ICMP echo reply, id 8402, seq 3, length 64
18:41:06.747701 ARP, Request who-has 192.168.50.1 tell 192.168.50.19, length 28
18:41:06.747715 ARP, Reply 192.168.50.1 is-at 0a:00:27:00:00:00 (oui Unknown), length 28
```

Теперь коротко, в два слова, о том, как сделать полновесный виртуальный интерфейс, работающий только со своим трафиком и не нарушающий работу родительского интерфейса (то, что делает полная версия модуля в архиве). Для этого необходимо:

1. Объявить *два* раздельных обработчика протоколов (для протоколов разрешения имен ARP и собственно для протокола IP):

```
// обработчик фреймов ETH_P_ARP
int arp_pack_rcv(struct sk_buff *skb, struct net_device *dev,
                 struct packet_type *pt, struct net_device *odev) {
   ...
   return skb->len;
};
```

```
static struct packet_type arp_proto = {
   __constant_htons(ETH_P_ARP),
   NULL,
   arp_pack_rcv,  // фильтр пртокола ETH_P_ARP
   (void*)1,
   NULL
};

// обработчик фреймов ETH_P_IP
int ip4_pack_rcv(struct sk_buff *skb, struct net_device *dev,
                 struct packet_type *pt, struct net_device *odev) {
   ...
   return skb->len;
};

static struct packet_type ip4_proto = {
   __constant_htons(ETH_P_IP),
   NULL,
   ip4_pack_rcv,    // фильтр пртокола ETH_P_IP
   (void*)1,
   NULL
};
```

2. И оба их зарегистрировать в функции инициализации модуля:

```
arp_proto.dev = ip4_proto.dev = priv->parent; // перехват только
                                              // с родительского интерфейса
dev_add_pack(&arp_proto);
dev_add_pack(&ip4_proto);
```

   Каждый из обработчиков должен осуществлять *подмену* интерфейса только для тех фреймов, IP получателя которых совпадает с IP его интерфейса, а два раздельных обработчика удобны тем, что заголовки фреймов ARP и IP имеют совершенно разный формат, и выделять IP назначения в них приходится по-разному (весь полный код показан в архиве примера).

Используя такой полновесный модуль, можно открыть к хосту, например, две параллельные сессии SSH на разные интерфейсы (использующие разные IP из разных подсетей), которые будут в параллель реально использовать единый общий физический интерфейс:

```
$ ssh olej@192.168.50.17
olej@192.168.50.17's password:
Last login: Mon Jul 16 15:52:16 2012 from 192.168.1.9
...
$ ssh olej@192.168.56.101
olej@192.168.56.101's password:
Last login: Mon Jul 16 17:29:57 2012 from 192.168.50.1
...
```

```
$ who
olej     tty1         2012-07-16 09:29 (:0)
olej     pts/0        2012-07-16 09:33 (:0.0)
olej     pts/1        2012-07-16 12:22 (192.168.1.9)
olej     pts/4        2012-07-16 15:52 (192.168.1.9)
olej     pts/6        2012-07-16 17:29 (192.168.50.1)
olej     pts/7        2012-07-16 17:31 (192.168.56.1)
```

Последняя показанная команда (who) выполняется уже внутри сессии SSH, т. е. на том самом удаленном хосте, к которому и фиксируются два *независимых* подключения из двух различных подсетей (последние две строки вывода), которые на самом деле представляют один хост.

Протокол транспортного уровня

На этом уровне (L4) обеспечивается обработка таких протоколов, как: UDP, TCP, SCTP, DCCP... их число возрастает. Протоколы транспортного уровня (протоколы IP) описаны в <linux/in.h>:

```
/* Standard well-defined IP protocols.  */
enum {
  IPPROTO_IP =     0,   /* Dummy protocol for TCP               */
  IPPROTO_ICMP =   1,   /* Internet Control Message Protocol    */
  IPPROTO_IGMP =   2,   /* Internet Group Management Protocol   */
...
  IPPROTO_TCP =    6,   /* Transmission Control Protocol        */
...
  IPPROTO_UDP =   17,   /* User Datagram Protocol               */
...
  IPPROTO_SCTP = 132,   /* Stream Control Transport Protocol    */
...
  IPPROTO_RAW  = 255,   /* Raw IP packets                       */
}
```

Для установки обработчика протоколов транспортного уровня существует API <net/protocol.h>:

```
struct net_protocol {                 // This is used to register protocols
   int  (*handler)(struct sk_buff *skb);
   void (*err_handler)(struct sk_buff *skb, u32 info);
   int  (*gso_send_check)(struct sk_buff *skb);
   struct sk_buff *(*gso_segment)(struct sk_buff *skb, int features);
   struct sk_buff **(*gro_receive)(struct sk_buff **head, struct sk_buff *skb);
   int (*gro_complete)(struct sk_buff *skb);
   unsigned int no_policy:1,
                netns_ok:1;
};
int inet_add_protocol(const struct net_protocol *prot, unsigned char num);
int inet_del_protocol(const struct net_protocol *prot, unsigned char num);
```

Здесь второй параметр вызова функций (num) как раз и есть константа из числа IPPROTO_*.

Эта схема в общих чертах напоминает то, как это же делалось на сетевом уровне (L3): каждый пакет проходит через функцию-фильтр, где мы можем анализировать или изменять отдельные поля сокетного буфера, отображающего пакет, и пробрасывать его дальше.

Пример модуля, устанавливающего протокол:

trn_proto.c

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

int test_proto_rcv(struct sk_buff *skb) {
   printk(KERN_INFO "Packet received with length: %u\n", skb->len);
   return skb->len;
};

static struct net_protocol test_proto = {
   .handler = test_proto_rcv,
   .err_handler = 0,
   .no_policy = 0,
};

//#define PROTO IPPROTO_ICMP
//#define PROTO IPPROTO_TCP
#define PROTO IPPROTO_RAW
static int __init my_init(void) {
   int ret;
   if((ret = inet_add_protocol(&test_proto, PROTO)) < 0) {
      printk(KERN_INFO "proto init: can't add protocol\n");
      return ret;
   };
   printk(KERN_INFO "proto module loaded\n");
   return 0;
}

static void __exit my_exit(void) {
   inet_del_protocol(&test_proto, PROTO);
   printk(KERN_INFO "proto module unloaded\n");
}

module_init(my_init);
module_exit(my_exit);
```

```
MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_LICENSE("GPL v2");
```

Вот как будет выглядеть работа модуля для протокола IPPROTO_RAW:

```
$ sudo insmod trn_proto.ko
$ lsmod | head  -n2
Module                  Size  Used by
trn_proto                780  0
$ cat /proc/modules | grep proto
trn_proto 780 0 - Live 0xf9a26000
$ ls -R /sys/module/trn_proto
/sys/module/trn_proto:
holders  initstate  notes  refcnt  sections  srcversion
...
$ sudo rmmod trn_proto
$ dmesg | tail -n60 | grep -v ^audit
proto module loaded
proto module unloaded
```

Но если вы попытаетесь установить (добавить!) обработчик для уже *обрабатываемого* (установленного) протокола (например, IPPROTO_TPC), то получите ошибку:

```
$ sudo insmod trn_proto.ko
insmod: error inserting 'trn_proto.ko': -1 Operation not permitted
$ dmesg | tail -n60 | grep -v ^audit
proto init: can't add protocol
$ lsmod | grep proto
$
```

Здесь возникает уже названная раньше сложность:

- если вы планируете отрабатывать новый (или не использующийся в системе) протокол, то для его тестирования в системе нет инструментов, и прежде нужно подумать о том, чтобы создать тестовые приложения прикладного уровня;
- если пытаться моделировать работу нового протокола под видом уже существующего (например, IPPROTO_UDP), то вам прежде понадобится удалить существующий обработчик, чем можно радикально нарушить работоспособность системы (например, для IPPROTO_UDP разрушить систему разрешения доменных имен DNS).

Использование драйверов Windows

Есть еще одна «мелочь» (возможность), связанная с драйверами сетевой подсистемы, которую стоит упомянуть... Многие производители оборудования не предоставляют Linux-драйверы для своих сетевых устройств — особенно часто это касается устройств, подключаемых по USB, и беспроводных сетевых устройств. Но для таких устройств есть возможность использовать драйверы для системы Windows (на самом деле не сами драйверы, их код, а их учетные данные интерфейса, содер-

жащиеся в сопровождающем файле .inf). Это реализует пакет ndiswrapper, который надо установить стандартным образом:

```
$ aptitude search ndiswrapper
p   ndiswrapper               - пользовательские инструменты для модуля
                                ndiswrapper ядра Linux
v   ndiswrapper-common
p   ndiswrapper-dkms          - исходный код модуля ndiswrapper для ядра
                                Linux (DKMS)
p   ndiswrapper-source        - Source for the ndiswrapper Linux kernel
                                module
v   ndiswrapper-utils-1.9
```

Для создания интерфейса нужно извлечь *текстовый* файл .inf из установочного комплекта (файлов) драйверов Windows и передать его утилите:

```
# ndiswrapper -i NETMZQ345.INF
 Installing netmzq345
# ndiswrapper -l
 Installed drivers:
 netmzq345               driver installed, hardware present
```

Если вы получите в ответ нечто подобное, то значит ndiswrapper готов поддерживать ваше устройство, — примите поздравления. Далее загрузите *модуль* ndiswrapper — например, командой из числа тех, которыми мы уже пользовались (позже вы можете определить его загрузку через конфигурации):

```
# modprobe ndiswrapper
```

После этого вы должны получить *новый* сетевой интерфейс, который дальше конфигурируете так, как мы это делали ранее (ifconfig, ip, Network Manager...).

Обратите внимание, что при этом мы не используем *код драйвера* для Windows (системы слишком разнородные), а задействуем только детальную учетную информацию относительно сетевого интерфейса. И этот трюк возможен поэтому только для некоторых сетевых устройств, но не для оборудования других классов.

> ***Примечание***
>
> Последняя актуальная версия пакета ndiswrapper (**https://sourceforge.net/projects/ndiswrapper/files/stable/**) — это 1.63, датирована она 03.05.2020, и с 2022 года пакет не включается в стандартные репозитории многих дистрибутивов Linux. Возможно, в виду того, что нужда отпала, т. к. Linux научился подхватывать большинство устройств нативным образом... Но тем не менее пакет в таком виде может использоваться и помочь в отдельных сложных случаях.

Обсуждение

В порядке напоминания хотелось бы посоветовать взять на заметку тот факт, что успех развития, отладки и тестирования сетевых модулей зависит не только от качественно прописанного кода и тщательно продуманного плана тестирования. Важным элементом успеха становится грамотное управление роутингом в сетевой

системе и добавление соответствующих записей в таблицу маршрутизации. Таблица роутинга демонстрируется вот такой командой:

```
$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.0     *               255.255.255.0   U     2      0        0 wlan0
default         192.168.1.1     0.0.0.0         UG    0      0        0 wlan0
```

До тех пор, пока в этой таблице не появится строка (строки), определяющая поведение нового интерфейса, над которым вы работаете, никакого положительного результата в поведении модуля добиться невозможно. А появиться эта строка может, только если ее добавить туда вручную той же командой `route`. Позже подобные действия могут выполняться синхронно с инсталляцией модуля при установке пакета. Но на этапе отработки гибкое управление роутингом становится залогом успеха. Эта особенность существенно отличает отработку сетевых модулей ядра от драйверов потоковых устройств в /dev, о которых мы говорили ранее.

И, конечно же, при отработке сетевых модулей ядра незаменимым инструментом становится такая утилита анализа трафика, как `tcpdump`.

Замечания относительно технологии отладки (отработки) кода сетевых модулей:

- отлаживать сетевые модули ядра, по крайней мере на начальных этапах разработки, в высшей степени, как никакой иной класс драйверов, плодотворно в виртуальной машине (например, под управлением Virtual Box);
- и производить это с терминала (одного или многих одновременно) удаленного хоста, подключенного к отлаживаемой виртуальной машине сессиями SSH;
- объясняются это удобство и продуктивность тем обстоятельством, что на виртуальной машине можно создать много (с запасом) сетевых интерфейсов разного свойства, — таких как виртуальный сетевой адаптер, виртуальный интерфейс хоста и т. п.;
- при этом SSH-отработку (редактирование кода, компиляцию, тестирование, отладку, ...) разрабатываемого модуля производим *не через тот интерфейс*, для которого тестируем модуль (здесь могут быть самые разнообразные варианты: тестируем на реальном адаптере — связываемся через виртуальный, или, напротив, тестируем на виртуальном адаптере — связываемся через реальный, и другие подобные варианты);
- при такой схеме в случае «падения» тестируемого интерфейса (что происходит неизбежно, постоянно и многократно) не разрывается связь и управление по SSH-интерфейсу, и мы можем легко перезапустить поврежденную конфигурацию без перезагрузки хоста и сети, что недостижимо при испытаниях на реальном оборудовании.

Использование такого стенда для отработки сетевых модулей показало снижение затрат времени на отработку *в разы*, что радикально ускоряет продвижение проекта.

- Глава 5 -
Внутренние API ядра

Очень трудно видеть и понимать неизбежное в хаосе вероятного.
Андрей Ваджра
(псевдоним украинского политолога и публициста)

В отличие от предыдущей главы, где мы рассматривали 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. То есть если раньше мы в коде обсуждали, *зачем* мы это делаем, то теперь будем разбираться, *как* мы это делаем. Большинство механизмов и понятий этой части описания уже знакомы нам по API пользовательского пространства — они имеют там свои прямые аналогии. Но существуют и некоторые принципиальные расхождения.

Механизмы управления памятью

В общем виде управление памятью в ядре Linux в защищенной аппаратной архитектуре — это самая сложная часть функций операционной системы. Еще более громоздкой ее делает то, что эта модель управления памятью должна отображаться на разных платформах в конкретные для платформы механизмы отображения логических (виртуальных) регионов памяти в физические. К счастью для разработчиков *модулей* ядра, большая часть механизмов управления памятью скрыта из нашего поля зрения и не имеет практического применения. Основная потребность разработчика модулей ядра состоит в выделении — динамически по требованию — блоков памяти заданного размера (иногда — при построении динамических структур данных — это очень небольшие блоки в десятки байтов, а иногда большие области размерами в мегабайты — например, при выделении циклических буферов данных в драйверах). В дальнейшем управление памятью мы будем рассматривать только в смысле таких потребностей — вопросы выделения больших регионов памяти, из которых «нарезаются» подобные запросы в модуле, и то, как это реализуется в ядре, рассматриваться не будут.

Карта памяти

Мы не станем углубляться в тонкости управления памятью — как в аппаратные, которые радикально меняются от одной процессорной архитектуры к другой, так и

в детальные механизмы того, как это делает Linux на уровне страниц и сегментных регистров... Но нам понадобится картина самого общего вида — как выглядит память с точки зрения *потребителя* (мы уже вскользь бросили на эту картину взгляд при рассмотрении отображения указателей функцией `printk()`, но сейчас посмотрим на нее детальнее и под другим углом).

И начнем мы с *32-битной* архитектуры, которая, хотя и устарела (в основном для процессоров линии *x*86), но еще актуальна для процессоров ARM, да и вообще для малых, встраиваемых устройств, массово изготавливамых под Linux. Просто примем во внимание следующие факты:

- адресное пространство ядра и адресное пространство *текущего* процесса (на который указывает макрос-указатель `current`) разделяют единое «плоское» адресуемое пространство виртуальных адресов. Для 32-битной архитектуры это пространство имеет полный размер в 4 Гбайт. Переключение контекста (состояния сегментных регистров) при переходе из пространства пользователя в пространство ядра в Linux *не производится*;
- исходя из этого, общий объем адресов должен разделяться в фиксированном соотношении между диапазоном для пространства пользователя и диапазоном для пространства ядра. Конкретно на 32-битных платформах это соотношение обычно 3:1, и общий диапазон адресов от `0x00000000` до `0xffffffff` разделяется граничным адресом `0xc0000000`: ниже него 3 Гбайт адресов принадлежат пространству пользователя, выше него 1 Гбайт адресов относится к пространству ядра. При этом все имена из /proc/kallsyms, все адреса функций в вашем коде модуля, все адреса динамически выделяемых по `kmalloc()` данных — все они будут находится выше границы `current` (это хорошо показано в примерах из папки sys_call_table/hidden сопровождающего книгу файлового архива). С другой стороны, все адреса в пользовательских приложениях окажутся ниже этой границы;
- в принципе, и пользовательское пространство, и пространство ядра могли бы располагать каждое своим изолированным адресным пространством — полным максимально возможным диапазоном адресов (для 32-битных платформ — по 4 Гбайт для процессов и для ядра). Но при этом возникла бы необходимость перезагрузки сегментных регистров при переключении в режим ядра — при выполнении системного вызова. Разработчики ядра Linux сочли это излишне накладным из соображений производительности[1];
- сегменты пространства пользователя, таким образом, имеют фиксированный ограниченный предел — для 32-битных систем это `0xc0000000`. Обработчики системных вызовов производят проверку принадлежности пространству пользователя параметров-указателей на *непревышение* этой границы. Код модуля мог бы, в принципе, впрямую использовать указатели в пользовательском коде, если бы не возможность физического отсутствия требуемой страницы в памяти (виртуа-

[1] Утверждается [2], что ядра после 2.6 с дополнительным патчем могут быть сгенерированы с поддержкой режима 4 GbГбайт/4 GbГбайт, — при этом достигается средняя, но приемлемая производительность.

лизация). Поэтому используются операции `copy_from_user()` и `copy_to_user()` для взаимодействия с данными пользователя;

- поскольку в область ядра должны отображаться еще некоторые области — например, аппаратные области видеопамяти, и некоторые такие области расширения ROM не допускают операций записи, то все они должны исключаться из общего адресного диапазона ядра, поэтому он будет еще немногим менее 1 Гбайт. На типовой архитектуре *x*86 объем непосредственно адресуемых логических адресов составляет 892 Мбайт;
- а вот отображение *одинаковых* логических адресов пространства *пользователя* из различных процессов в отличающиеся физические адреса происходит не за счет различий *сегментных регистров*, а за счет различий на уровне страничного отображения памяти, осуществляемого MMU[2];
- соотношение 3:1 и соответственно граница разделения `0xc0000000` могут быть изменены при новой *генерации* ядра Linux. Это иногда и делается сборщиками дистрибутивов из разных соображений (из-за ограниченности объема RAM встраиваемых устройств или других). Граница определяется конфигурационным параметром ядра `CONFIG_PAGE_OFFSET`.

Вот как это выглядит в семействе устройств Raspberry Pi :

```
$ inxi -Smxxx
System:
  Host: raspberrypi Kernel: 5.15.32-v7+ armv7l bits: 32 compiler: gcc v: 10.2.1
  Console: tty 2 DM: LightDM 1.26.0 Distro: Raspbian GNU/Linux 11 (bullseye)
Memory:
  RAM: total: 999.1 MiB used: 320.1 MiB (32.0%) gpu: 76 MiB
  RAM Report: unknown-error: Unknown dmidecode error. Unable to generate data.
$ grep CONFIG_PAGE_OFFSET /lib/modules/`uname -r`/build/.config
CONFIG_PAGE_OFFSET=0x80000000
```

И это любопытно сравнить с еще меньшим его ARM-собратом из совершенно другого семейства устройств — Orange Pi One (рис. 5.1):

```
$ inxi -Smxxx
System:
  Host: orangepione Kernel: 5.15.48-sunxi armv7l bits: 32 compiler: N/A Console: tty 2
  dm: LightDM 1.26.0 Distro: Armbian GNU/Linux 10 (buster)
Memory:
  RAM: total: 491.7 MiB used: 265.7 MiB (54.0%)
  RAM Report: missing: Required program dmidecode not available
$ grep CONFIG_PAGE_OFFSET /lib/modules/`uname -r`/build/.config
CONFIG_PAGE_OFFSET=0xC0000000
$ grep CONFIG_PAGE_OFFSET /boot/config-`uname -r`
CONFIG_PAGE_OFFSET=0xC0000000
```

[2] Memory management unit, блок управления памятью (аппаратная реализация управления таблицами страниц).

Рис. 5.1. Самое миниатюрное Linux-устройство — Orange Pi One в работе (эквивалентная стоимость $9)

Теперь посмотрим на 64-битные архитектуры — что поменялось относительно только что приведенных фактов?

- Прежде всего, это *не есть* архитектура с 64-битными адресами. Для адресации используются только 48 (0–47) младших битов из 64, а старшие биты (48–63) *расширяются* битом 47 — подобно тому, как это делается при расширении знакового разряда отрицательных целых. Эта спецификация адреса в сегодняшних процессорах поддерживается аппаратно, что позволяет планировать легкость расширения в будущем до адресов 52, 57 и даже 64 битов (такие цифры называют в планах).
- В 64-битной Linux разделение виртуальных адресных пространств пользователь–ядро производится поровну: 1:1, по значению бита 47 (1 — ядро, 0 — пользовательский процесс). Поэтому адреса плоского адресного пространства для пользовательских процессов будут выглядеть так: `[000000000000000 - 00007fffffffffff)`, а для ядра — так: `[ffff800000000000 - ffffffffffffffff]` (скобки квадратные и круглые в обозначении этих диапазонов означают, как это и принято в математике, «включая» и «исключая» соответственно).
- Как виртуальные 48-битные адреса затем аппаратно, с помощью MMU, преобразуются в физические адреса реальной памяти, нас на текущем этапе не интересует. Но интересует, как они *представляются*.
- В этой модели памяти параметр конфигурации ядра `CONFIG_PAGE_OFFSET` не определяется (и значение `current` — не ищите его) — 48-битная граница фиксирована и в переопределении не нуждается:

```
$ inxi -Smxxx
System:    Host: R420 Kernel: 5.4.0-124-generic x86_64 bits: 64 compiler: gcc v: 9.4.0
                                                                    Desktop: Cinnamon 5.2.7
           wm: muffin 5.2.1 dm: LightDM 1.30.0 Distro: Linux Mint 20.3 Una base: Ubuntu 20.04
                                                                                  focal
Memory:    RAM: total: 94.35 GiB used: 7.64 GiB (8.1%)
           RAM Report: permissions: Unable to run dmidecode. Root privileges required.
$ grep CONFIG_PAGE_OFFSET /lib/modules/`uname -r`/build/.config
$
```

Динамическое выделение памяти

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()` и пр.) являются библиотечными вызовами, которые ретранслируются (транзитом через соответствующие системные вызовы[3]) 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 сохраняют прямой практический интерес и для прикладного программирования (пространства пользователя).

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

- однократное распределение буферов данных (иногда весьма и объемных, и сложно структурированных), которое выполняется, как правило, при начальной инициализации модуля (в сетевых драйверах часто при активизации интерфейса командой `ifconfig`);
- многократное динамическое создание/уничтожение временных структур, организованных в некоторые списочные структуры (характерный пример: создание и уничтожение сокетных буферов).

Первоначально мы рассматриваем механизмы первой названной группы (которые, собственно, и являются механизмами динамического управления памятью), но к концу изложения отклонимся и рассмотрим использование циклических двусвяз-

---

[3] В зависимости от размера запрашиваемой области и от реализации используемой библиотеки `libc`, `malloc()` может транслироваться в системные вызовы `brk()` или `mmap()` — если размер запрашиваемой области велик. Убедится в этом можно, написав простой цикл выделения памяти с различными запрашиваемыми размерами и запустив тестовую программу под `strace` (спасибо читателям, которые обратили внимание на необходимость такого дополнения).

ных списков, учитывая их максимально широкое применение в ядре Linux (и призывы разработчиков ядра использовать только эти или подобные им, там же описанные, структуры).

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

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

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

Впервые встретившийся нам параметр `flags` очень часто фигурирует в коде ядра (в отличие от пользовательского кода) и определяет то, какими *характеристиками* должен обладать запрошенный участок памяти. Возможных вариантов значений этого флага — великое множество. Они определены в отдельном файле <gfp.h> — например: `__GFP_WAIT`, `__GFP_HIGH`, `__GFP_MOVABLE` и пр. Рассмотрим только немногие из них, наиболее часто используемые, и те, которые нам активно потребуются в дальнейшем изложении:

- `GFP_KERNEL (__GFP__WAIT | __GFP__IO | __GFP__FS)` — выделение производится от имени процесса, который выполняет системный запрос в пространстве ядра. Такой запрос может быть временно переведен в пассивное состояние (блокирован);
- `GFP_ATOMIC (__GFP_HIGH)` — выделение памяти в обработчиках прерываний, тасклетах, таймерах ядра и другом коде, выполняющемся вне контекста процесса. Такой запрос не может быть блокирован (нет процесса, который следовало бы активировать после блокирования). Но это означает, что в случаях, когда память могла бы быть выделена после некоторого блокирования, будет сразу возвращаться ошибка;

Все эти флаги могут быть совместно (по «или») определены с большим числом других, например таким:

`GFP_DMA` — выделение памяти должно произойти в DMA-совместимой зоне памяти.

Выделенный в результате блок может быть больше запрошенного размера (что никогда не создает проблем пользователю) и ни при каких обстоятельствах не может быть меньше. В зависимости от размера страницы архитектуры минимальный размер возвращаемого блока может занять 32 или 64 байта, а максимальный размер зависит от архитектуры, но если рассчитывать на переносимость, то, как утверждается в литературе, это не должно быть больше 128 Кбайт. Но даже при размерах, превышающих одну страницу (несколько килобайт, для *x*86 — 4 Кбайт), есть лучшие способы, чем получение памяти по `kmalloc()`.

После использования всякий блок памяти должен быть освобожден. Это касается вообще любого способа выделения блока памяти, которые еще 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 неразмещенного блока приводит к тяжелым 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 возвращенная им при выгрузке явно, никогда больше не возвратится под управление системы.

Альтернативным `kmalloc()` способом выделения блока памяти, но *не обязательно в непрерывной области* в физической памяти, является вызов:

```
#include <linux/vmalloc.h>
void *vmalloc(unsigned long size);
void vfree(void *addr);
```

Распределение `vmalloc()` менее производительно, чем `kmalloc()`, но может стать предпочтительнее при выделении больших блоков памяти, когда `kmalloc()` вообще не сможет выделить блок требуемого размера и завершится аварийно. Отображение страниц физической памяти в непрерывную логическую область, возвращаемую `vmalloc()`, обеспечивает MMU, 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 с оборудованием по DMA).

Еще одним (итого три) принципиально иным способом выделения памяти будут те вызовы API ядра, 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 далее.

Управление памятью Linux производит *страницами*. Физически (на аппаратном уровне) 32-битная архитектура поддерживает двухуровневые таблицы трансляции адреса (TLB) и размер страниц памяти ядра 4 Кбайт. В архитектуре *x*86_64, как утверждается, возможно использовать страницы размером 4 Кбайт (4096 байтов), 2 Мбайт и (в некоторых AMD64) 1 Гбайт. В ядре размер *логических* (виртуальных) страниц определяется *константой компиляции* ядра `PAGESIZE`, которую в работающей системе можно посмотреть командой:

```
$ getconf PAGESIZE
4096
```

> ***Примечание***
>
> Я *никогда* не видел значения `PAGESIZE`, отличного от 4096, хотя и имел дело с десятками дистрибутивов под архитектуру *x*86, *x*86_64, ARM... Возможно, оно может отличаться в другой процессорной архитектуре или его можно переопределить, но это потребует *полной перекомпиляции* ядра.

Вопрос сравнения возможностей по выделению памяти различными способами актуален, но весьма запутан (по литературным источникам), т. к. радикально зависит от используемой архитектруры процессора, физических ресурсов оборудования (объем реальной RAM, число процессоров SMP, ...), версии ядра Linux и других факторов.

Вопрос динамического распределения памяти — важнейший для разработчика модулей ядра. Этот вопрос настолько важен, что отдельно заслуживает обстоятельного тестирования, и такие оценки были проделаны для нескольких конфигураций, но ввиду объемности сами тесты и их обсуждение вынесены в отдельное *приложение 2*, а здесь приведена только сводная таблица результатов (табл. 5.1).

Таблица 5.1. *Результаты тестов динамического распределения памяти*

Архитектура	**Максимальный выделенный блок* (байт)**		
	`kmalloc()`	`__get_free_pages()`	`vmalloc()`
Orange Pi One, ARM RAM 512 Mb kernel: 5.15.48	2 097 152 (2 Mb)	4 194 304 (4 Mb)	268 435 456 (256 Mb)
Raspberry Pi 2 B, ARM RAM 1 Gb kernel: 5.15.32	4 194 304 (4 Mb)	4 194 304 (4 Mb)	536 870 912 (512 Mb)
Antix21, 32 бит, VirtualBox RAM 4 Gb kernel: 4.9.0	4 194 304 (4 Mb)	4 194 304 (4 Mb)	67 108 864 (64 Mb)
AMD GX-212JC SOC, 64 бит RAM 8 Gb kernel: 4.9.0	4 194 304 (4 Mb)	4 194 304 (4 Mb)	4 294 967 296 (4 Gb)
Dell PowerEdge R420, Intel Xeon E5-2470 v2 RAM 96 Gb kernel: 5.0.4	4 194 304 (4 Mb)	4 194 304 (4 Mb)	68 719 476 736 (64 Gb)

* 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 *когерентные* методы (выделяющие *непрерывный* участок в *физической* памяти) `kmalloc()` и `__get_free_pages()` во всех современных версиях ядра Linux выделяют максимальный блок памяти в 10–15 страниц `PAGESIZE` — 4 Мбайт или немногим больше, *независимо* от размера присутствующей аппаратной памяти. Напротив, `vmalloc()`, выделяя блок *логической* (виртуальной) памяти без гарантии, что он будет *отображен* в непрерывный участок *физической* памяти, может выделить единый блок размером до 2/3 реально присутствующей (аппаратной) памяти.

Еще одно сравнение (оно описано полностью там же, в *приложении 2*) — по затратам процессорных тактов на одно выполнение запроса на выделение (табл. 5.2).

Таблица 5.2. *Затраты процессорных тактов на одно выполнение запроса на выделение*

Размер блока (байт)	Затраты (число процессорных тактов)**		
	kmalloc()	__get_free_pages()	vmalloc()
5*	67	391	1703
1000*	70	389	1467
4096	69	383	1743
65 536	672	608	4816
262 144	1158	1278	14 984
4 096 000	19 623	—	265 116
40 960 000	—	—	2 751 904

* Не кратно PAGE_SIZE.

** Оценки времени, связанные с диспетчеризацией процессов в системе, могут отличаться в 2–3 раза в ту или иную сторону и могут быть только грубыми ориентирами порядка величины.

Распределители памяти

Реально распределение памяти по запросам kmalloc() может поддерживаться различными механизмами более низкого уровня, называемыми *распределителями*. Совершенно не обязательно, что 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 находится «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 (куче — едином пространстве для динамического выделения памяти). В этом методе большой блок памяти (heap) используется для обеспечения памятью под любые цели. 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 поиска свободного участка, либо first-fit (первый встречающийся блок, превышающий запрошенный размер), либо best-fit (блок, вмещающий запрошенный размер с наименьшим превышением). Когда блок памяти больше не нужен, он возвращается в heap. Основная проблема этой стратегии распределения — *фрагментация* и деградация системы с течением длительного времени непрерывной эксплуатации (что особо актуально для серверов). Проблемой вторичного порядка малости является высокая затратность времени для управления свободным пространством heap.

Подход, применявшийся в Linux для выделения больших регионов (называемый buddy memory allocation), выделяет по запросу блок размером, кратным степени 2, и превышающий фактический запрошенный размер (по существу, используется подход best-fit). При освобождении блока предпринимается попытка объединить в освобождаемый свободный блок все свободные соседние блоки (слить). Такой подход позволяет снизить фрагментирование и повышает эффективность управления свободным пространством. Но он может существенно увеличить непродуктивное расходование памяти.

Алгоритм распределителя, использующийся `kmalloc()` в качестве основного механизма в версиях ядра 2.6 для текущего выделения небольших блоков памяти, — это *сляб-аллокатор* (slab allocator). Слябовый распределитель впервые предложен Джефом Бонвиком (Jeff Bonwick) и реализован и описан в SunOS (в середине 1990-х годов). Идея такого распределителя состоит в том, что последовательные запросы на выделение памяти под объекты *равного* размера удовлетворяются из области одного кеша (сляба), а запросы на объекты другого размера (пусть отличающиеся от первого случая самым незначительным образом) — удовлетворяются из совершенно другого такого же кеша, но под другой размер.

> ***Примечание***
>
> Сам термин «slab» переводится близко к «облицовочная плитка», и принцип очень похож: любую вынутую из выкладки облицовочную плитку можно заменить другой такой же, но это только потому, что их размеры в точности совпадают.

Использование аллокатора `SLAB` (по умолчанию) может быть отменено при новой сборке ядра (параметр `CONFIG_SLAB`). Это имеет смысл и используется для небольших и встроенных систем. При таком решении может быть включен аллокатор, который называют `SLOB` — тогда участки выделяемой памяти выстраиваются в единый линейный связный список. Этот способ распределения памяти может экономить до 512 Кбайт памяти (в сравнении со `SLAB`). Естественно, способ страдает названными уже недостатками, главный из которых — фрагментация.

Начиная с версии ядра 2.6.22, начинает использоваться распределитель `SLUB`, разработанный Кристофом Лэйметром (Christoph Lameter) из компании SGI, но это только отличающаяся *реализация* все той же идеи распределителя `SLAB`. В отличие от `SLOB`, ориентированного на малые конфигурации, `SLUB` ориентирован, напротив, на системы с большими и огромными (huge) объемами RAM. Идея состоит в том, чтобы уменьшить непроизводительные расходы на управляющие структуры слябов при их больших объемах. Для этого управление организуется не на основе единичных страниц памяти, а на основе объединения таких страниц в группы и управления на базе групп страниц.

Детально смотрите, какой распределитель используется по конфигурационным параметрам, с которыми собиралось ядро, например, так:

```
$ inxi -S
System:
  Host: raspberrypi Kernel: 5.15.32-v7+ armv7l bits:         32 Console: tty 2
  Distro: Raspbian GNU/Linux 11 (bullseye)
```

```
$ grep CONFIG_SLOB /lib/modules/`uname -r`/build/.config
# CONFIG_SLOB is not set
$ grep CONFIG_SLAB /lib/modules/`uname -r`/build/.config
# CONFIG_SLAB is not set
CONFIG_SLAB_MERGE_DEFAULT=y
# CONFIG_SLAB_FREELIST_RANDOM is not set
# CONFIG_SLAB_FREELIST_HARDENED is not set
$ grep CONFIG_SLUB /lib/modules/`uname -r`/build/.config
CONFIG_SLUB_DEBUG=y
CONFIG_SLUB=y
CONFIG_SLUB_CPU_PARTIAL=y
# CONFIG_SLUB_DEBUG_ON is not set
# CONFIG_SLUB_STATS is not set
```

Примечание

Далее более или менее детально мы будем рассматривать только слябовый распределитель SLAB (потому что выбор распределителя — это только альтернатива из области предпочтений).

Слябовый распределитель

Текущее состояние слябового распределителя мы можем рассмотреть в файловой системе /proc (что дает достаточно много для понимания самого принципа слябового распределения):

```
$ cat /proc/slabinfo
slabinfo - version: 2.1
# name     <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit>
<batchcount> <sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
...
kmalloc-8192     28     32   8192    4   8 : tunables    0   0   0 : slabdata     8     8     0
kmalloc-4096    589    648   4096    8   8 : tunables    0   0   0 : slabdata    81    81     0
kmalloc-2048    609    672   2048   16   8 : tunables    0   0   0 : slabdata    42    42     0
kmalloc-1024    489    512   1024   16   4 : tunables    0   0   0 : slabdata    32    32     0
kmalloc-512    3548   3648    512   16   2 : tunables    0   0   0 : slabdata   228   228     0
kmalloc-256     524    656    256   16   1 : tunables    0   0   0 : slabdata    41    41     0
kmalloc-128   13802  14304    128   32   1 : tunables    0   0   0 : slabdata   447   447     0
kmalloc-64    12460  13120     64   64   1 : tunables    0   0   0 : slabdata   205   205     0
kmalloc-32    12239  12800     32  128   1 : tunables    0   0   0 : slabdata   100   100     0
kmalloc-16    25638  25856     16  256   1 : tunables    0   0   0 : slabdata   101   101     0
kmalloc-8     11662  11776      8  512   1 : tunables    0   0   0 : slabdata    23    23     0
...
```

Этот принцип прост: сам сляб должен быть создан (зарегистрирован) вызовом kmem_cache_create(), а потом из него можно «черпать» элементы фиксированного размера (под который и был создан сляб) вызовами kmem_cache_alloc() (это и есть тот вызов, в который в конечном итоге с наибольшей частотой 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));
```

Здесь:

- `name` — строка имени кеша;
- `size` — размер элементов кеша (единый и общий для всех элементов);
- `offset` — смещение первого элемента от начала кеша (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 и до 3.14 можем наблюдать следующую фазу изменений (меняется прототип конструктора):

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

...и к 5.4 становится (почти то же, но типизация параметров заметно поменялась):

```
struct kmem_cache *kmem_cache_create(const char *name, unsigned int size,
                    unsigned int align, slab_flag_t flags,
                    void (*ctor)(void*));
```

Все это означает, что то, что компилировалось для одного ядра, перестанет компилироваться для следующего. Вообще-то, это вполне обычная практика для ядра, но к этому нужно быть готовым, и при использовании таких достаточно глубинных механизмов руководствоваться не навыками, а изучением заголовочных файлов текущего ядра. Хотя изменений ваш код потребует не столь уж существенных...

Из флагов создания, поскольку они также находятся в постоянном изменении, и большая часть из них относится к отладочным опциям, стоит назвать:

- SLAB_HWCACHE_ALIGN — расположение каждого элемента в слябе должно выравниваться по строкам процессорного кеша. Это может существенно поднять производительность, но непродуктивно расходует память;
- SLAB_POISON — начально заполняет сляб предопределенным значением (A5A5A5A5) для обнаружения выборки неинициализированных значений;
- если не нужны какие-то особые изыски, то нулевое значение будет вполне уместно для параметра flags.

Как для любой операции выделения, ей сопутствует обратная операция по уничтожению сляба:

```
int kmem_cache_destroy(kmem_cache_t *cache);
```

Операция уничтожения может быть успешна (здесь достаточно редкий случай, когда функция уничтожения возвращает значение результата), только если уже *все* объекты, полученные из кеша, были возвращены в него. Таким образом, модуль должен проверить статус, возвращенный kmem_cache_destroy(), — и ошибка указывает на какой-то вид утечки памяти в модуле (т. к. некоторые объекты не были возвращены).

Когда кеш объектов создан, вы можете выделять (черпать) 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 возвращен, когда в нем отпадет необходимость:

```
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

Несмотря на изменчивость API сляб-аллокатора, вы можете охватить даже диапазон версий ядра, пользуясь директивами условной трансляции препроцессора. Модуль, использующий такой аллокатор, может выглядеть подобно следующему (см. папку memory/slab в сопровождающем книгу файловом архиве):

slab.c

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");
MODULE_VERSION("6.2");

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
   *(long*)p = (long)p;
   sco++;
}

#define SLABNAME "my_cache"
struct kmem_cache *cache = NULL;

static int __init init(void) {
   int i;
   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 cleanup(void) {
   int i;
   for(i = 0; i < number; i++)
      kmem_cache_free(cache, line[i]);
   kmem_cache_destroy(cache);
   kfree(line);
}
module_exit(cleanup);
```

Вот как выглядит выполнение этого размещения (картина весьма поучительная, поэтому остановимся на ней подробнее) в достаточно свежих версиях ядра (разной разрядности, да и разных процессорных архитектур и дистрибутивов Linux):

```
$ inxi -S
System:    Host: raspberrypi Kernel: 5.15.32-v7+ armv7l bits: 32 Console: tty 3
           Distro: Raspbian GNU/Linux 11 (bullseye)
$ uname -a
Linux raspberrypi 5.15.32-v7+ #1538 SMP Thu Mar 31 19:38:48 BST 2022 armv7l GNU/Linux
$ sudo insmod slab.ko
$ lsmod | head -n3
Module                  Size  Used by
slab                   16384  0
sha256_generic         16384  0
$ dmesg | grep -v hwmon | tail -n3
[25650.044692] allocate 31 objects into slab: my_cache
[25650.044730] object size 7 bytes, full size 217 bytes
[25650.044747] constructor called 256 times
$ sudo cat /proc/slabinfo | grep my_
my_cache         256  256   16  256   1 : tunables    0    0    0 : slabdata      1   1      0
```

Итого: объекты размером 7 байтов благополучно разместились в новом слябе с именем `my_cache`, отображаемом в каталоге /proc/slabinfo, организованном с размером элементов 16 байтов (эффект выравнивания?). Конструктор при размещении 31 такого объекта вызывался 257 раз. Обратим внимание на чрезвычайно важное обстоятельство: при создании сляба никоим образом не указывается реальный или *максимальный* объем памяти, находящейся под управлением этого сляба, — это динамическая структура, «добирающая» столько *страниц* памяти, сколько нужно для поддержания размещения требуемого числа элементов данных (с учетом их размера). Увеличенное число вызовов конструктора можно отнести: а) на необходимость переразмещения существующих элементов при последующих запросах и б) на эффекты SMP (2 ядра) и перераспределения данных между процессорами.

Примечание

Если рассмотреть три первых поля вывода /proc/slabinfo, то видно, что под сляб размечено некоторое фиксированное количество фиксированных объекто-мест (256), которые укладываются в некоторый начальный объем сляба в одну страницу (или целое число страниц) физической памяти.

Проверим тест на 64-разрядном дистрибутиве antiX:

```
$ inxi -S
System:
  Host: antix21 Kernel: 4.9.0-279-antix.1-amd64-smp arch: x86_64 bits: 64 Console: pty pts/1
    Distro: antiX-21_x64-base Grup Yorum 31 October 2021
$ uname -a
Linux antix21 4.9.0-279-antix.1-amd64-smp #1 SMP PREEMPT Sun Aug 8 15:04:18 EEST 2021 x86_64
GNU/Linux
$ sudo insmod slab.ko size=3 number=1000
olej@antix21:~/2022/kernel/memory/slab
$ dmesg | tail -n3
[  817.243489] allocate 1000 objects into slab: my_cache
[  817.243493] object size 3 bytes, full size 3000 bytes
[  817.243494] constructor called 1024 times
$ sudo cat /proc/slabinfo | grep my_
my_cache          1024   1024    16  256   1 : tunables   0   0   0 : slabdata     4     4     0
```

Для полноты картины (вопросы переносимости) проверим тот же тест на однопроцессорном Celeron и насколько возможно более старой версии ядра:

```
$ uname -r
2.6.18-92.el5
$ sudo /sbin/insmod ./slab.ko
$ /sbin/lsmod | grep slab
slab                    7052  0
$ dmesg | tail -n3
allocate 31 objects into slab: my_cache
object size 7 bytes, full size 217 bytes
constructor called 339 times
$ cat /proc/slabinfo | grep my_
my_cache     31   339     8  339   1 : tunables  120  60   8 : slabdata     1     1     0
$ sudo /sbin/rmmod slab
```

Число вызовов конструктора не уменьшилось, а даже возросло, а вот размер объектов, под который создан сляб, изменился с 16 на 8. Но общий вид картины не меняется на протяжении... лет 15...

И для полноты картины для 64-битной реализации (размер объектов 11 байтов — 7 байтов слишком короткий объект для 64-битной реализации *mecma*):

```
$ uname -r
3.14.4-200.fc20.x86_64
$ sudo insmod ./slab.ko size=11
$ dmesg | tail -n3
```

```
[11835.972001] allocate 31 objects into slab: my_cache
[11835.972007] object size 11 bytes, full size 341 bytes
[11835.972009] constructor called 129 times
$ sudo cat /proc/slabinfo | grep my_
my_cache     128   128     32  128    1 : tunables    0    0    0 : slabdata      1      1      0
$ sudo rmmod slab
```

> ***Примечание***
>
> Последний рассматриваемый пример любопытен в своем поведении. Вообще-то «завалить» операционную систему Linux — ничего не стоит, когда вы пишете модули ядра. Тем не менее за несколько лет плотной (почти ежедневной) работы с микроядерной операционной системой QNX мне так и не удалось ее «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 здесь 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, а потом и вся подсистема X11... но остаются в живых черные текстовые консоли, в которых даже живут указатели мыши. Интереснейший получается эффект[4].

Еще одна вариация на тему распределителя памяти, в том числе и сляб-аллокатора, — 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 аллокатором, а всего лишь представляет собой *интерфейс* к аллокатору (*к любому* — к тому же кешу, 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. Параметр вызова `min_nr` определяет то минимальное число 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 должны иметь такие прототипы:

```
typedef void* (*mempool_alloc_t)(int gfp_mask, void *pool_data);
typedef void (*mempool_free_t)(void *element, void *pool_data);
```

Последний параметр `mempool_create()`: `pool_data` (блок данных) — передается последним параметром в вызовы `alloc_fn()` и `free_fn()`.

Но обычно просто дают обработчику-распределителю ядра выполнить за нас задачу — поскольку в <linux/mempool.h> объявлено несколько групп API для разных рас-

[4] Что напомнило высказывание классика отечественного юмора М. Жванецкого: «А вы пробовали принять слабительное одновременно со снотворным? Очень интересный эффект».

пределителей памяти. Так, например, существуют две функции: `mempool_alloc_slab()` и `mempool_free_slab()` — ориентированные на рассмотренный уже сляб-аллокатор и выполняющие соответствующие согласования между прототипами выделения пула памяти `kmem_cache_alloc()` и `kmem_cache_free()`. В результате код, инициализирующий пул памяти, который будет автоматически использовать сляб-аллокатор для управления памятью, часто выглядит следующим образом:

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

Когда пул 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 предопределенных объектов. После этого вызовы `mempool_alloc_slab()` пытаются получить новые объекты от функции выделения — возвращается один из предопределенных объектов (если таковые сохранились). Когда объект освобожден `mempool_free_slab()`, он сохраняется в пуле, если количество предопределенных объектов в настоящее время ниже минимального. В противном случае он будет возвращен 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 изменен:

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

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

Когда пул памяти больше не нужен, он возвращается системе:

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

В папке /memory/slab сопровождающего книгу файлового архива присутствует пример модуля (файл pool.c), демонстрирующий реализацию пула памяти как над кешем (`mempool_create_slab_pool()`), так и над распределителем (`mempool_kmalloc()`), и позволяющий наблюдать создание пула, размещение в нем элементов и тестирование корректности размещенных элементов. Тест позволяет наглядно поэкспериментировать с техникой создания пулов. Мы не станем рассматривать здесь этот пример — и код, и протокол его сборки и использования (файлы .hist) для дистрибутива Fedora 35 помещены в архив для самостоятельного экспериментирования.

Страничное выделение

Когда нужны крупные блоки, больше одной машинной страницы и кратные целому числу страниц, выполняется вызов:

```
#include <linux/gfp.h>
struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)
```

Такой вызов выделяет `2**order` смежных страниц (*непрерывный* участок) физической памяти. Полученный физический адрес требуется конвертировать в логический для использования:

```
void *page_address(struct page * page)
```

Если не требуется физический адрес, то сразу получить его логический эквивалент 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 ее 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, 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 — отличное от того, которое выделялось, карта памяти становится поврежденной, и система позднее (может, и через продолжительное время) будет *разрушена*.

Выделение больших буферов

Для выделения экстремально больших буферов иногда описывают (в литературе) и рекомендуют технику выделения памяти непосредственно при загрузке системы (ядра). Но эта техника доступна только модулям, загружаемым с ядром (при начальной загрузке), после чего они не подлежат выгрузке. Техника, приемлемая для команды разработчиков ядра, но очень сомнительная в своей ценности для сторонних разработчиков модулей ядра. Тем не менее вскользь упомянем и ее. Для ее 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 освобожденную память позже, если возникнет когда-либо такая необходимость, он будет 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

И статический, и динамический способы размещения структур данных имеют свои положительные и отрицательные стороны, главными из которых принято считать: для статической памяти — надежность, живучесть и меньшая подверженность ошибкам, а для динамической памяти — гибкость применения. Использование динамических структур всегда требует того или иного механизма управления памятью, обеспечивающего создание и уничтожение терминальных элементов динамически увязываемых структур.

Циклический двусвязный список

Чтобы уменьшить количество дублирующегося кода, разработчики ядра создали (начиная с ядра 2.6) стандартную реализацию на основе кругового двойного двусвязного списка. До этого в ядре использовалось много разнообразных списочных структур «по ситуации» (как это принято и в других областях программирования). После такой единообразной реорганизации всем нуждающимся в манипулировании динамическими списочными структурами (начиная с простейших линейных односвязных списков, пример которых показан на рис. 5.2, и до сложных древовидных структур) разработчиками рекомендуется использовать именно это средство (рис. 5.3). И поэтому оно заслуживают внимательного рассмотрения.

Рис. 5.2. Традиционный односвязный линейный список, завершающийся `NULL`

Примечание

При работе с интерфейсом связного списка всегда следует иметь в виду, что функции списка сами по себе выполняются без блокировки. Если есть вероятность того, что драйвер может попытаться выполнить на одном списке конкурентные операции, уже

вашей обязанностью является реализация схемы блокировки. Результаты некорректного параллельного доступа (поврежденные структуры списка, потеря данных, паники ядра), как правило, трудно диагностировать.

Рис. 5.3. Альтернативный двусвязный циклический список

Обратите внимание, что двусвязный циклический список нигде не содержит концевых указателей, имеющих значение `NULL`. Это гарантирует, что при ошибках манипуляции со списком вы можете двигаться ошибочными путями, но *никогда* не станете разыменовывать указатель `NULL`, чем сразу же гарантированно завалите ядро операционной системы.

Чтобы задействовать механизм списка, ваш драйвер должен подключить файл <linux/list.h>. Этот файл определяет базовую структуру типа `list_head`:

```
struct list_head {
   struct list_head *next, *prev;
};
```

Для использования в вашем коде средства списка Linux необходимо всего лишь вставлять `list_head` внутри собственных структур, входящих в список, — например:

```
struct todo_struct {
   struct list_head list;
   int priority;
   /* ... добавить другие зависимые от задачи поля */
};
```

Эта условная структура и будет в дальнейшем использоваться для иллюстрации работы макросов, обслуживающих связные списки.

Заголовки списков должны быть проинициализированы перед использованием с помощью макроса `INIT_LIST_HEAD`. Заголовок списка может быть объявлен и проинициализирован так (динамически):

```
struct list_head todo_list;
INIT_LIST_HEAD(&todo_list);
```

Альтернативно списки могут быть созданы и проинициализированы статически при компиляции:

```
LIST_HEAD(todo_list);
```

Некоторые функции для работы со списками определены в <linux/list.h>. Как мы видим, API работы с циклическим списком позволяет выразить любые операции с элементами списка, не вовлекая в операции манипулирования с внутренними полями связи списка, — это очень ценно для сохранения целостности списков.

Макрос:

```
list_add(struct list_head *new, struct list_head *head);
```

добавляет новую запись `new` сразу же *после головы* списка (`head`) — как правило, в начало списка (хотя в циклическом списке понятие «начала» достаточно условное — это вы сами должны понимать, что есть в нем «начало», и следить за этим). Таким образом, этот макрос может быть использован для создания стеков. Однако следует отметить, что голова не должна быть номинальной головой списка — если вы передадите в качестве `new` структуру `list_head`, которая окажется где-то в середине списка, новая запись пойдет сразу после нее. Так как списки Linux являются круговыми, голова списка обычно не отличается от любой другой записи.

Следующий макрос добавляет элемент `new` *перед головой* списка (`head`) — *в конец* списка:

```
list_add_tail(struct list_head *new, struct list_head *head);
```

Другими словами, функция `list_add_tail()` может, таким образом, быть использована для создания очередей FIFO: первый вошел — первый вышел.

Следующие макросы *удаляют* указанную запись из списка (обратим внимание, что указывать сам список в параметрах при этом не нужно). Если эта запись может быть когда-либо вставлена в другой список, вы должны использовать `list_del_init()`, которая инициализирует заново указатели связного списка:

```
list_del(struct list_head *entry);
list_del_init(struct list_head *entry);
```

Следующие макросы удаляют указанную запись из своего текущего положения и *перемещают* ее: а) в начало списка — с функцией `list_move()` или б) в конец списка — с функцией `list_move_tail()`:

```
list_move(struct list_head *entry, struct list_head *head);
list_move_tail(struct list_head *entry, struct list_head *head);
```

Макрос:

```
list_empty(struct list_head *head);
```

возвращает ненулевое значение, если список пуст (т. е. тестирует его).

Макрос:

```
list_splice(struct list_head *list, struct list_head *head);
```

обеспечивает объединение двух списков со *вставкой* нового списка `list` сразу *после* головы `head`.

Структуры `list_head` хороши для реализации линейных списков, но задействующие их программы почти всегда больше заинтересованы в некоторых более крупных

структурах, которые увязываются в список как его узлы. При этом широко используется макрос `list_entry()`, который по значению указателя структуры `list_head` восстанавливает указатель на экземпляр структуры, которая именно его и содержит. Он вызывается следующим образом:

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

Здесь `ptr` является указателем на используемую структуру `list_head`, `type_of_struct` является указателем на *имя типа* структуры, содержащей в себе этот элемент связи (`ptr`), а `field_name` является *именем поля* связи (`struct list_head`) в этой структуре.

В показанной раньше для примера структуре `todo_struct` поле списка *имеет имя* `list`. Таким образом, когда мы хотели бы получить по указателю поля связи `listptr` саму содержащую это поле структуру типа `struct todo_struct`, то могли бы выразить это такой строкой:

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

Реализовать подобное (разработчикам ядра) можно было только *макросом*, сделать это вызовом *функции* было бы нельзя. Макрос `list_entry()` несколько необычен и требует некоторого времени, чтобы привыкнуть, но его не так сложно использовать.

Обход двусвязных списков достаточно прост: надо только использовать указатели `prev` и `next`. В качестве примера предположим, что мы хотим добавлять новый элемент в список так, чтобы сохранять список объектов `todo_struct`, отсортированный в порядке убывания поля `priority`. Функция добавления новой записи будет выглядеть примерно следующим образом:

```
void todo_add_entry(struct todo_struct *new) {
   struct list_head *ptr;
   struct todo_struct *entry;
   /* голова списка поиска: todo_list */
   for(ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next) {
      entry = list_entry(ptr, struct todo_struct, list);
      if(entry->priority < new->priority) {
         list_add_tail(&new->list, ptr);
         return;
      }
   }
   list_add_tail(&new->list, &todo_list);
}
```

Однако, как правило, лучше использовать один из набора *предопределенных макросов* (*итераторов*) — для операций, которые перебирают списки. Например, предыдущий цикл мог бы быть написан 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 предопределенных макросов помогает избежать простых ошибок программирования. Разработчики этих макросов также приложили некоторые усилия, чтобы они выполнялись производительно. Существует несколько вариантов *итераторов*:

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

  Этот макрос создает цикл `for`, который выполняется по одному разу с указателем `cursor`, присвоенным поочередно 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 такие же 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 — она просто сохраняет следующую запись в списке в `next` (дополнительный параметр) для продолжения цикла, поэтому не запутается, если запись, на которую указывает `cursor`, удаляется.

- `list_for_each_entry(type *cursor, struct list_head *list, member)`
  `list_for_each_entry_safe(type *cursor, type *next, struct list_head *list, member)`

  Такие *макросы* облегчают процесс просмотра списка, содержащего структуры заданного типа `type`. Здесь `cursor` является указателем на экземпляр охватывающей структуры (результат), содержащей поле связи, `member` является *именем структуры* связи `list_head` внутри содержащей структуры. С этими макросами нет необходимости помещать внутри цикла вызов макроса `list_entry()`.

В заголовках <linux/list.h> определен еще ряд деклараций для описания динамических структур и манипуляций с ними.

Модуль, использующий динамические структуры

Здесь показан пример модуля ядра (см. папку memory/list в сопровождающем книгу файловом архиве), строящего, использующего и утилизирующего 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);
```

```
$ inxi -S
System:
  Host: orangepione Kernel: 5.15.48-sunxi armv7l bits: 32 Console: tty 3
  Distro: Armbian GNU/Linux 10 (buster)
$ /sbin/modinfo mod_list.ko
filename:       /home/olej/2022/kernel/examples/memory/list/mod_list.ko
license:        GPL
depends:
name:           mod_list
vermagic:       5.15.48-sunxi SMP mod_unload ARMv7 thumb2 p2v8
parm:           size:int
$ sudo insmod mod_list.ko size=7
```

```
insmod: ERROR: could not insert module mod_list.ko: Operation not permitted
$ dmesg | tail -n7
[41031.363628] [LIST] 6
[41031.363642] [LIST] 5
[41031.363647] [LIST] 4
[41031.363652] [LIST] 3
[41031.363656] [LIST] 2
[41031.363661] [LIST] 1
[41031.363665] [LIST] 0
```

Сложноструктурированные данные

Всего только одной ограниченной структуры данных `struct list_head` достаточно для построения динамических структур практически произвольной степени сложности — таких как, например, сбалансированные B-деревья, красно-черные списки и другие. Именно поэтому ядро 2.6 было полностью переписано в части задействуемых списочных 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 собрать собственный набор функций его создания/инициализации и манипуляций с узлами такого дерева.

Еще об инициализации объектов ядра

При обсуждении заголовков списков были показаны две (альтернативно, на выбор) возможности объявить и инициализировать такой заголовок:

- *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 ее адреса:

  ```
  struct list_head todo_list;
  INIT_LIST_HEAD(&todo_list);
  ```

Такая же дуальность (статика/динамика) возможностей будет наблюдаться далее много раз — например, во всех примитивах синхронизации (мьютексов, семафоров, ...). Напрашиваются вопросы: 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 прием-

лем только динамический способ инициализации. Единичные переменные проще создавать статически.

Служба времени

Всему свое время, и время всякой вещи под небом.
Екклесиаст, III:1

Как-то так сложилось мнение, что только что законченная нами в рассмотрении тема динамического управления памятью является сложной. Но она-то как раз относительно проста. По-настоящему сложная и неисчерпаемая тема (в любой операционной системе!) — это тема службы времени. Еще одной особенностью подсистемы времени, которой мы 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, проще изучать на коде пользовательского адресного пространства, чем ядра.

Во всех функциях времени основным принципом остается положение, сформулированное расширением реального времени POSIX 1003b: временные интервалы *никогда* не могут быть короче, чем затребованные, но могут быть *сколь угодно больше* затребованных (в условиях шедулирования более приоритетных задач).

Информация о времени в ядре

Сложность подсистемы времени усугубляется тем, что для повышения точности или функциональности API времени разработчики привлекают несколько разнородных и несинхронизированных источников временных меток (из тех, 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, *x*86) набор и возможности задатчиков времени обновляются с развитием и изменяются каждые 2–3 года, а соответственно, изменяется все поведение в деталях подсистемы времени. Но какие бы ни были платформенные или архитектурные различия, нужно отчетливо 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 системным таймером аппаратно через постоянные интервалы — этот интервал программируется во время загрузки Linux записью соответствующего коэффициента в аппаратный счетчик-делитель. Делается это в соответствии с одной из самых фундаментальных констант ядра — константой

периода компиляции (определенной директивой `#define`) с именем `HZ` (tick rate). Значение этой константы, вообще-то говоря, является архитектурно-зависимой величиной, определено оно в <linux/param.h>, значения по умолчанию в исходных текстах ядра имеют диапазон от 50 до 1200 тиков в секунду на различном реальном оборудовании, снижаясь до 24 в программных эмуляторах. Значение `HZ` — это временной масштаб происходящего в системе: меньшие значения создают более «тормозную» систему, большие — снижают производительность системы за счет накладных расходов на поддержание службы времени.

По прерыванию системного таймера происходят в системе все важнейшие события:

- обновление значения времени работы системы (uptime) абсолютного времени (time of day);
- проверка, не израсходовал ли текущий процесс свой квант времени, и если израсходовал, то выполняется 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 всей 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).

В литературе утверждается, что для большинства платформ Linux 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. На самом деле, экспериментальная проверка показывает, что это далеко не так. Поэтому нужно составить хотя бы поверхностное представление, в каких системах и как это происходит (в скобках названы дистрибутивы, в которых я наблюдал точно то же):

- Fedora 35 (то же значение в antiX-21_x64 ядро 4.9.0, antiX-21_386 ядро 4.9.0 VirtualBox):

```
$ inxi -S
System:
  Host: xenix.localdomain Kernel: 5.18.16-100.fc35.x86_64 arch: x86_64 bits: 64
    Console: pty pts/3 Distro: Fedora release 35 (Thirty Five)
$ uname -a
Linux xenix.localdomain 5.18.16-100.fc35.x86_64 #1 SMP PREEMPT_DYNAMIC Thu Aug 4 02:06:53
UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
```

```
$ grep CONFIG_HZ /boot/config-`uname -r`
# CONFIG_HZ_PERIODIC is not set
# CONFIG_HZ_100 is not set
# CONFIG_HZ_250 is not set
# CONFIG_HZ_300 is not set
CONFIG_HZ_1000=y
CONFIG_HZ=1000
```

- Mint 20.3 (очень неожиданно... то же значение в Astra Linux Orel 2.12.43 ядро 4.19.0, LMDE 5 64 бит ядро 5.10.0):

```
$ inxi -S
System:    Host: R420 Kernel: 5.4.0-124-generic x86_64 bits: 64 Desktop: Cinnamon 5.2.7
Distro: Linux Mint 20.3 Una
$ uname -a
Linux R420 5.4.0-124-generic #140-Ubuntu SMP Thu Aug 4 02:23:37 UTC 2022 x86_64 x86_64
x86_64 GNU/Linux
$ grep CONFIG_HZ /boot/config-`uname -r`
# CONFIG_HZ_PERIODIC is not set
# CONFIG_HZ_100 is not set
CONFIG_HZ_250=y
# CONFIG_HZ_300 is not set
# CONFIG_HZ_1000 is not set
CONFIG_HZ=250
```

- Orange Pi One, Armbian:

```
$ inxi -S
System:
  Host: orangepione Kernel: 5.15.48-sunxi armv7l bits: 32 Console: tty 3
  Distro: Armbian GNU/Linux 10 (buster)
$ uname -a
Linux orangepione 5.15.48-sunxi #22.05.3 SMP Wed Jun 22 07:35:10 UTC 2022 armv7l GNU/Linux
$ grep CONFIG_HZ /lib/modules/`uname -r`/build/.config
# CONFIG_HZ_PERIODIC is not set
CONFIG_HZ_FIXED=0
# CONFIG_HZ_100 is not set
# CONFIG_HZ_200 is not set
CONFIG_HZ_250=y
# CONFIG_HZ_300 is not set
# CONFIG_HZ_500 is not set
# CONFIG_HZ_1000 is not set
CONFIG_HZ=250
```

- Raspberry Pi, Raspbian OS (в высшей степени неожиданно) :

```
$ inxi -S
System:    Host: raspberrypi Kernel: 5.15.32-v7+ armv7l bits: 32 Console: tty 3
           Distro: Raspbian GNU/Linux 11 (bullseye)
$ uname -a
```

```
Linux raspberrypi 5.15.32-v7+ #1538 SMP Thu Mar 31 19:38:48 BST 2022 armv7l GNU/Linux
$ grep CONFIG_HZ /lib/modules/`uname -r`/build/.config
# CONFIG_HZ_PERIODIC is not set
CONFIG_HZ_FIXED=0
CONFIG_HZ_100=y
# CONFIG_HZ_200 is not set
# CONFIG_HZ_250 is not set
# CONFIG_HZ_300 is not set
# CONFIG_HZ_500 is not set
# CONFIG_HZ_1000 is not set
CONFIG_HZ=100
```

Как можно видеть, единого правила нет, авторы каждой сборки на сегодня руководствуются какими-то своими загадочными соображениями. И даже виртуальные инсталляции под VirtualBox вопреки публикациям определяют значения `HZ` временами как 1000. Поскольку значение `HZ` крайне важно для поведения системы, единственный совет, который можно дать: экспериментально *тестируйте* это значение в дистрибутиве, который вы используете (или — еще более того — планируете использовать в проекте)!

Понятно, что чем *меньше* `HZ` — тем выше при прочих равных условиях *производительность* системы, меньше постоянные издержки на реакцию на события временного тика. Чем *больше* HZ — тем выше *отзывчивость* системы на внешние события, действия пользователя с клавиатурой или мышью, или в реагировании на внешние датчики (это не реалтайм, но иногда ложно толкуется как реалтайм).

Источник прерываний системного таймера

Источник прерываний системного таймера (определяющий последовательность тиков частоты `HZ` и подсчитываемых в счетчике `jiffies`) — аппаратная микросхема системного таймера. В архитектуре *x*86 таким задатчиком является микросхема по типу Intel 82C54, работающая от отдельного кварца стандартизованной частоты 1,1931816 МГц, далее эта частота 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 близкая к выбранному в заголовочном файле <linux/param.h> значению `HZ`, равному 1000,152 Гц, что соответствует периоду (ticksize) 999 847 нс (рассогласование с 1 мс составляет –0,0153%).

> ***Примечание***
>
> Ближайшее соседнее значение делителя 1194 дает частоту и период 999,314 Гц и 1 000 680 нс (рассогласование с 1 мс составляет +0,068%) соответственно, но всегда

используется значение периода с недостатком. В противном случае задержка, величина которой определена такой программной строкой:

```
struct timespec ts = { 0 /* секунды */, 1000500 /* наносекунды */ };
```

могла бы (при некоторых прогонах) завершиться на первом тике, что противоречит требованиям стандарта POSIX 1003b о том, что временной интервал может быть *сколь угодно больше заказанного, но ни в коем случае не меньше*!

Тот же принцип формирования периода системных тиков соблюдается и на любой другой аппаратной платформе, на которой выполняется Linux: целочисленный делитель счетчика, задающий максимальное приближенное аппаратное значение частоты к выбранному значению константы `HZ` с избытком (т. е. период системного таймера с недостатком к значению 1/`HZ`). Это будет существенно важно для толкования полученных нами вскоре 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 *x*86, например) еще несколько источников событий для временной шкалы: часы реального времени (RTC), таймеры контроллеров прерываний APIC, специальные счетчики 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 (на 2 порядка и более) по сравнению с временем выполнения единичной команды процессора. В масштабе времени 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-ядерный ноутбук уровня 2007 г. :

```
$ 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-ядерный процесор образца 2011 г. :

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:        127          0          0          0   IO-APIC-edge      timer
...
  8:          0          0          0          0   IO-APIC-edge      rtc0
...
 LOC:  460580288  309522125 2269395963  161407978   Local timer interrupts
...
 RES:     919591     983178     315144     626188   Rescheduling interrupts
...
$ uname -r
2.6.35.11-83.fc14.i686
```

- 4-ядерный ноутбук 2014 г., 64-битная система:

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:         32          0          0          0   IO-APIC-edge      timer
...
  8:          0          1          0          0   IO-APIC-edge      rtc0
...
LOC:   10631735   11158944   10831185   10010565   Local timer interrupts
...
RES:      89820      58446      69061      36388   Rescheduling interrupts
...
$ uname -r
3.13.10-200.fc20.x86_64
```

В общем виде это выглядит так: если на каком-то конкретном компьютере обнаружен тот или иной источник информации для времени, то он будет использоваться для уточнения интервальных значений, если нет — то будет задействована шкала *системных тиков*. Это означает, что на подобных экземплярах оборудования (различных экземплярах *x*86-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 во временной области:

- измерение временных интервалов;
- выдержка пауз во времени;
- отложенные во времени действия.

В отношении задач каждого класса присутствуют свои ограничения, а также 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 отсчета в коде программы вполне может основываться на таких простейших механизмах, как считанные значения счетчика 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
```

Это мы и используем в первом примере модуля (см. папку time в сопровождающем книгу файловом архиве) из области механизмов времени, — наш модуль всего лишь замеряет интервал времени, за 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);

MODULE_LICENSE("GPL");
```

Вот результат выполнения такого модуля — обратите внимание на хорошее соответствие временного интервала (23 секунды), замеренного в пространстве пользователя (командным интерпретатором) и интервального измерения в ядре:

```
# date; insmod interv.ko
Сб 13 авг 2022 02:58:03 EEST

# date; rmmod interv
Сб 13 авг 2022 02:58:26 EEST
# dmesg | tail -n3
[101865.933176] module: jiffies on start = 1833FA0
[101889.068876] module: jiffies on finish = 1835635
[101889.068881] module: interval of life = 23
```

Счетчик 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 потянет и <linux/jiffies.h>. Определяются два подобных имени: `jiffies` и `jiffies_64` (независимо от разрядности системы), представляющие значение в 32 и 64 бита соответственно. Излишне говорить, что `jiffies` и `jiffies_64` должны рассматриваться как только читаемые. Счетчик `jiffies` считает системные тики от *момента последней загрузки* системы.

При последовательных считываниях `jiffies` может быть зафиксировано его переполнение (32 бита — это не такое большое значение). Чтобы не заморачиваться с анализом, ядро предоставляет четыре однотипных макроса для сравнения двух значений счетчика импульсов таймера, которые корректно обрабатывают переполнение счетчиков. Они определены в файле <linux/jiffies.h> следующим образом:

```
#define time_after(unknown, known) ((long)(known) - (long)(unknown) < 0)
#define time_before(unknown, known) ((long)(unknown) - (long)(known) < 0)
#define time_after_eq(unknown, known) ((long)(unknown) - (long)(known) >= 0)
#define time_before_eq(unknown, known) ((long)(known) - (long)(unknown) >= 0)
```

Здесь `unknown` — это обычно значение переменной `jiffies`, а параметр `known` — это значение, с которым его необходимо сравнить. Макросы возвращают значение `true`, если выполняются *соотношения* моментов времени `unknown` и `known`, в противном случае возвращается значение `false`.

Иногда, однако, необходимо обмениваться представлением времени с программами пользовательского пространства, которые, как правило, определяют значения времени структурами `timeval` и `timespec`. Эти две структуры предоставляют точное значение времени как структуру из двух чисел: секунды и микросекунды используются в старой и популярной структуре `timeval`, а в новой структуре `timespec` представляются секунды и наносекунды. Ядро экспортирует четыре вспомогательные функции для преобразования значений времени, выраженного в `jiffies`, в эти структуры и из них:

```
#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-разрядному счетчику тиков не так прост, как доступ к `jiffies`. В то время как на 64-разрядных архитектурах эти две переменные являются фактически одной, доступ к значению `jiffies_64` для 32-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 — учитывая именно простоту реализации такой задачи, что уже обсуждалось ранее, — для измерения временных интервалов могут быть использованы другие дополнительные механизмы. Такие дополнительные источники информации о времени позволяют получить много более высокое (на несколько порядков!) разрешение, чем при опоре на системный таймер (а иначе зачем нужно было бы привлекать новые механизмы?). Простейшим из таких прецизионных датчиков времени может быть регистр-счетчик периодов тактирующей частоты процессора с возможностью его программного считывания.

Наиболее известным примером такого регистра-счетчика является TSC (timestamp counter), введенный в *x*86-процессоры, начиная с Pentium, и с тех пор он присутствует во всех последующих моделях процессоров этого семейства, включая платформу *x*86_64. Это 64-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 *x*86, означающий machine-specific registers) можно использовать один из макросов:

- `rdtsc(low32, high32)` — атомарно читает 64-разрядное значение в две 32-разрядные переменные;
- `rdtscl(low32)` — читает младшую половину регистра в 32-разрядную переменную, отбрасывая старшую половину;
- `rdtscll(var64)` — читает 64-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 с использованием счетчика процессорных тактов можно найти в файле memtim.c (см. папку memory/mtest в сопровождающем книгу файловом архиве), посвященном тестированию распределителя памяти (обсуждается в *приложении 2*).

Большинство других процессорных платформ также предлагают аналогичную (но в чем-то отличающуюся в деталях) функциональную возможность. Заголовки ядра поэтому включают архитектурно-независимую функцию, скрывающую существующие различия реализации, которую можно использовать вместо `rdtsc()`. Она называется `get_cycles()` (определена в <asm/timex.h>). Ее 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 счетчика 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 кода самой системы Linux.

Чтобы наблюдать эффекты *измерений* в службе времени мы создадим небольшой тестовый проект. Для такого анализа это может быть еще более показательно реализовано как процесс в пространстве пользователя — наблюдаемые эффекты будут точно те же, что и в ядре. Это фактически «ручная» (для наглядности — на инлайновой ассемблерной вставке) реализация счетчика процессорных циклов `rdtsc()` для пользовательского пространства[5], которая выполняет те же функции, что и вызовы в ядре `rdtsc()`, `rdtscl()`, `rdtscll()` или `get_cycles()` (всё это, естественно, раз здесь ассемблер, работает *только* на процессорной архитектуре *x*86 — 32- или 64-битной):

[5] Опыт двух десятилетий участия в реальных разработческих проектах показал, что такая штука под рукой — это весьма полезное подспорье, когда речь идет об области системного программирования. И дополнительный аргумент в пользу того, чтобы поговорить об этом.

clock.c

```
#include "libdiag.h"

int main(int argc, char *argv[]) {
   printf("%016llX\n", rdtsc());
   printf("%016llX\n", rdtsc());
   printf("%016llX\n", rdtsc());
   printf("%d\n", proc_hz());
   return EXIT_SUCCESS;
}
```

Для измерения значений размерностей времени мы подготовим небольшую целевую статическую библиотеку (libdiag.h), которую станем применять не только в этом тесте, но и в других примерах, там где речь идет о *пользовательском пространстве*. Вот основные библиотечные модули. Основной компонент проекта — чтение процессорного счетчика TSC:

rdtsc.c

```
#include "libdiag.h"

uint64_t rdtsc(void) {
   union sc {
      struct {uint32_t lo, hi;} r;
      uint64_t f;
   } sc;
   __asm__ __volatile__ ("rdtsc" : "=a"(sc.r.lo), "=d"(sc.r.hi));
   return sc.f;
}
```

Функция калибровки затрат процессорных тактов на само выполнение вызова `rdtsc()` выполняется за два непосредственно следующих друг за другом последовательных вызова `rdtsc()`. Для снижения погрешностей это значение усредняется по циклу повторений:

calibr.c

```
#include "libdiag.h"

#define NUMB 10
unsigned calibr(int rep) {
   uint64_t n, m, sum = 0;
   n = m = (rep >= 0 ? NUMB : rep);
   while(n--) {
      uint64_t cf, cs;
```

```
      cf = rdtsc();
      cs = rdtsc();
      sum += cs - cf;
   }
   return (uint32_t)(sum / m);
}
```

Измерение частоты процессора (число процессорных тактов за секундный интервал) на основе предыдущих функций:

proc_hz.c

```
#include "libdiag.h"

uint64_t proc_hz(void) {
   time_t t1, t2;
   int64_t cf, cs;
   time(&t1);
   while(t1 == time(&t2)) cf  = rdtsc(); // начало след. секунды
   while(t2 == time(&t1)) cs  = rdtsc(); // начало след. секунды
   return cs - cf - calibr(1000);        // с учетом времени вызова rdtsc()
}
```

Перевод потока в реалтайм режима диспетчирования (в частности, на FIFO-дисциплину), что бывает очень полезно сделать при любых измерениях временных интервалов:

set_rt.c

```
void set_rt(void) {
   struct sched_param sched_p;           // Information related to scheduling priority
   sched_getparam(getpid(), &sched_p);   // Change the scheduling policy to SCHED_FIFO
   sched_p.sched_priority = 50;          // RT Priority
   sched_setscheduler(getpid(), SCHED_FIFO, &sched_p);
}
```

Выполним этот тест на компьютерах *x86* самой разной архитектуры (1, 2, 4, ... ядра), разного времени изготовления (поколений процессора), производительности и версий ядра (собственно, только для такого сравнения и есть смысл готовить этот тест) — чтобы убедиться, что это работает, и увидеть, как это работает:

```
$ inxi -C
CPU:
  Info: dual core model: AMD GX-212JC SOC with Radeon R2E Graphics bits: 64 type: MCP cache:
    L2: 1024 KiB
  Speed (MHz): avg: 1198 min/max: N/A cores: 1: 1198 2: 1198
```

```
$ ./clock
0000031FE09C9A7E
0000031FE09FF266
0000031FE0A02631
1197723673

$ inxi -C
CPU:
  Info: quad core model: Intel Core i7-4870HQ bits: 64 type: MT MCP cache:
    L2: 1024 KiB
  Speed (MHz): avg: 1438 min/max: 800/3700 cores: 1: 2487 2: 2171 3: 1796
    4: 1718 5: 937 6: 799 7: 799 8: 799
$ ./clock
0000079C6EA6200E
0000079C6EABA149
0000079C6EABE5BE
2494282644

$ inxi -C
CPU:       Info: Quad Core model: Intel Xeon E3-1240 v3 bits: 64 type: MT MCP L2 cache: 8 MiB
           Speed: 3392 MHz min/max: 800/3800 MHz Core speeds (MHz): 1: 3392 2: 3392 3: 3392
           4: 3392 5: 3392 6: 3395 7: 3393 8: 3393
$ ./clock
000C5BDDB70AA8C4
000C5BDDB70D5558
000C5BDDB70D6520
3392082385
```

Наблюдать подобную сравнительную картину на различном оборудовании — чрезвычайно полезное и любопытное занятие, но мы не станем сейчас останавливаться на деталях увиденного, отметим только высокую точность совпадения измерений со значениями, получаемыми независимыми системными способами, для того процессора (ядра), *на котором выполняется* пользовательский процесс (в принципе, вы можете выделить для этого отдельно конкретно выбранный процессор аффинити-маской запуска теста командой Linux `taskset`).

Наконец, мы соберем 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 (здесь показаны *x*86-процессоры и ARM, 64- и 32-битные):

```
$ sudo insmod tick.ko ; dmesg | tail -n3
insmod: ERROR: could not insert module tick.ko: Operation not permitted
[ 5136.704493] jiffies = 10049CE8A
[ 5136.704495] HZ value = 1000
[ 5136.704497] jiffies 64-bit = 000000010049CE8A
$ sudo insmod tick.ko ; dmesg | tail -n3
insmod: ERROR: could not insert module tick.ko: Operation not permitted
[ 4962.521382] jiffies = 10011C839
[ 4962.521384] HZ value = 250
[ 4962.521385] jiffies 64-bit = 000000010011C839
$ sudo insmod tick.ko ; dmesg | tail -n3
insmod: ERROR: could not insert module tick.ko: Operation not permitted
[94751.391193] jiffies = 901F2A
[94751.391227] HZ value = 100
[94751.391239] jiffies 64-bit = 0000000100901F2A
```

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 еще и с тем, что требуемая величина обеспечиваемой паузы может быть в очень широком диапазоне: от миллисекунд и короче — для обеспечения корректной работы оборудования и протоколов (например, обнаружения 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 малую величину и ни при каких условиях — короче. Это условие не так легко выполнить!

Реализация временной задержки 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 блокирован-

ное 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: все определяется конкретным контекстом использования. Например, любой переход в блокированное состояние — это очень трудоемкая операция со стороны системы (переключения контекста, смена адресного пространства и множество других действий), а реализация коротких пауз способом активного ожидания может просто оказаться эффективнее (прямую аналогию чего мы увидим при рассмотрении примитивов синхронизации: семафоров и спин-блокировки). 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 и 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 двух значений с учетом возможных переполнений (уже рассмотренный ранее);
- `cpu_relax()` — макрос, говорящий, что процессор ничем не занят, и в гипертрейдинговых системах могущий (в некоторой степени) занять процессор еще чем-то;

В конечном счете 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* с прототипами:

```
void ndelay(unsigned long nanoseconds);
void udelay(unsigned long microseconds);
void mdelay(unsigned long milliseconds);
```

Хотя они и определены на самом деле 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 не 0, если функция вернулась перед истечением заданного времени ожидания (в ответ на сигнал). Функция `schedule_timeout()` *требует*, чтобы 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 использующих 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 *миллисекунд*. Вызов `msleep()` является непрерываемым — можно быть уверенным, что процесс остановлен, по крайней мере, на заданное число миллисекунд. Если драйвер помещен в очередь ожидания и мы хотим использовать возможность принудительного пробуждения (сигналом) для прерывания пассивности, то 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.

Разберемся с различием между активными и пассивными задержками, причем учитывая, что различие это абсолютно одинаково что в ядре, что пользовательском процессе, рассмотрение его можно сделать с тем же успехом, но гораздо менее трудоемко, из процесса пространства пользователя:

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 разных ядрах могут давать самый разнообразный *характер* результатов — вот картина, наиболее характерная для относительно старых архитектур и ядер (мы видим именно классическую картину диспетчирования по тикам системного таймера — без привлечения дополнительных аппаратных уточняющих источников информации высокого разрешения. Это то поведение, что описывается во всех *учебниках* по POIX системам):

```
$ 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 одной 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 зачет» выдержки времени, потому как вызов `nanosleep()` происходит асинхронно относительно шкалы системных тиков и мог бы прийти 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 (0,999847 мс, как это отмечалось ранее), и вот этот остаток «чуть» и приводит к ожиданию еще одного очередного, неисчерпанного тика.

Как раз более необъяснимыми (хотя и более ожидаемыми по житейской логике) будут цифры на *новых* архитектурах (аппаратуры) и ядрах (системы):

```
$ uname -r
5.4.0-124-generic
$ sudo nice -n-19 ./pdelay -n 1000
1000 cycles passive delay [millisec. == tick !] :
processor speed: 2399970429 hz
set 0.010 => was 0.063
set 0.100 => was 0.153
set 0.200 => was 0.253
set 0.300 => was 0.361
set 0.500 => was 0.562
set 1.000 => was 1.063
set 1.500 => was 1.563
set 2.000 => was 2.063
set 5.000 => was 5.067
```

Здесь определенно для получения такой разрешающей способности использованы другие дополнительные датчики временных шкал *высокого разрешения*, 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 величины интервала ожидания, адекватной заказанной величине, — реально это может быть величина, бóльшая *в разы*...
- расчет на то, что активная задержка выполнится с большей точностью (и может быть задана с меньшей дискретностью), отчасти оправдан, но на это также не следует твердо рассчитывать: выполняющий активные циклы поток может быть вытеснен в блокированное состояние, и интервал ожидания будет суммироваться с временем блокировки, а это еще хуже (в смысле погрешности), чем в случае пассивных задержек;
- за счет возможности вытеснения в блокированное состояние временные паузы могут (с невысокой вероятностью, но все же) 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 являются таймерные функции. Понятие таймера существенно шире и сложнее в реализации, чем просто выжидание некоторого интервала времени, как это было отмечено ранее. Таймер (экземпляров которого 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);
```

Здесь:

- `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 в `void*` указатель на структуру.

Функция таймера в ядре выполняется в *контексте прерывания* (не в контексте процесса! А конкретнее: в контексте обработчика прерывания системного таймера). И это накладывает на нее дополнительные ограничения:

- не разрешен доступ к пользовательскому пространству. Из-за отсутствия контекста процесса нет пути к пользовательскому пространству, связанному с любым определенным процессом;
- указатель `current` не имеет смысла и не может быть использован, т. к. 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 (модуля) 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, и структуры представления времени для них определяются в файлах <linux/ktime.h>. Поддержка осуществляется только в тех архитектурах, где предусмотрена 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 определяется множество функций установки значений и преобразований представления времени (многие из них определены как макросы, но здесь записаны как прототипы — для наглядного представления типов данных):

```
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 реального времени 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 константное значение:

```
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 сущность из области документации POSIX — то, что называется *стандартом временного базиса* (тип задатчика времени): какую шкалу времени использовать из общего числа определенных 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

Все, что касается временных базисов в IT, — очень тонко и путанно... Относительно временных базисов в Linux известно следующее:

- `CLOCK_REALTIME` — системные часы со всеми их плюсами и минусами. Могут быть переведены вперед 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 счетчик, в связи с чем, естественно, не может быть изменен при переводе времени. Обычно это счетчик, идущий от последней загрузки системы;
- `CLOCK_PROCESS_CPUTIME_ID` — возвращает время, затрачиваемое процессором относительно пользовательского процесса, т. е. время, затраченное процессором на работу только с выполняемым приложением, вне зависимости 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 отличие от него, не подвержен изменению через сетевой протокол точного времени NTP.
- последние два базиса: `CLOCK_REALTIME_COARSE` и `CLOCK_MONOTONIC_COARSE` добавлены относительно недавно (2009 год), их 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 временным базисом таймеров вполне могут быть, например, CLOCK_REALTIME или CLOCK_MONOTONIC. Пример использования таймеров высокого разрешения в периодическом режиме может быть показан следующим 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
$ sudo rmmod htick.ko
$ dmesg | tail -n5
[10497.093195] timer start at jiffies=4297516322
[10498.093203] timer run #3 at jiffies=4297516572
[10499.093191] timer run #2 at jiffies=4297516822
[10500.093182] timer run #1 at jiffies=4297517072
[10501.093174] timer run #0 at jiffies=4297517322
```

Абсолютное время

Все предыдущее рассмотрение касалось измерения *относительных* временных интервалов (даже если эта относительность отсчитывается от достаточно отдаленной во времени точки загрузки системы, как в случае с `jiffies`). Реальное хронологическое время (абсолютное время) нужно ядру или модулю 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 1970 г.) хранится так:

```
struct timespec xtime;
```

В UNIX традиционно (исторически) существуют 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
```

Ввиду *неатомарности* `xtime`, непосредственно использовать его нельзя, но есть некоторый набор API ядра для преобразования из хронологического времени в одну из этих 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 поддержки для этого на используемой платформе). Оно очень сильно зависит от используемой платформы):

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
```

Часы реального времени (RTC)

Часы реального времени — это сугубо аппаратное расширение, которое принципиально зависит от аппаратной платформы, на которой используется Linux. Это еще одно расширение службы системных часов, и на некоторых архитектурах его может и *не быть*. Используя такое расширение, можно создать еще одну — независимую — шкалу отсчетов времени, с которой можно связать измерения или даже асинхронную активацию действий.

Убедиться в наличии или отсутствии такого расширения на используемой аппаратной платформе можно по присутствию интерфейса к таймеру часов реального времени в пространстве пользователя. Такой интерфейс предоставляется (о чем 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
```

Вот как характерно выглядит RTC (или его отсутствие) в двух миниатюрных образцах ARM одноплатных компьютеров (внешне кажущихся подобными) — Raspberry Pi и Orange Pi One):

```
$ inxi -M
Machine:   Type: ARM Device System: Raspberry Pi 2 Model B Rev 1.1 details: BCM2835
           rev: a21041 serial: 00000000f57e2ca8
$ ls -l /dev/rtc*
ls: невозможно получить доступ к '/dev/rtc*': Нет такого файла или каталога
$ inxi -M
Machine:   Type: ARM Device System: Xunlong Orange Pi One details: Allwinner sun8i Family
           rev: N/A
           serial: 02c000815fd5e717
$ ls -l /dev/rtc*
lrwxrwxrwx 1 root root      4 июл 22 17:17 /dev/rtc -> rtc0
crw------- 1 root root 253, 0 июл 22 17:17 /dev/rtc0
```

В архитектуре Intel *x*86 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 32 768 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, в южный мост) чипсета процессора. RTC возвращает не только время суток (UTC!), 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;
};
```

Вот 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:

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;
}
```

Вот как это выглядит в *x*86-архитектуре (обращение к /dev и показанный ранее вывод `ls` для устройства RTC демонстрируют, что работа с ним производится только от root):

```
$ sudo ./rtcr
current time: 11:29:27
$ date
Сб 13 авг 2022 14:29:41 EEST
```

И то же самое в архитектуре ARM — там, где есть RTC:

```
$ inxi -M
Machine:   Type: ARM Device System: Xunlong Orange Pi One details: Allwinner sun8i Family rev: N/A
           serial: 02c000815fd5e717
$ make rtcr
gcc     rtcr.c   -o rtcr
$ sudo ./rtcr
current time: 11:33:58
$ date
Суб авг 13 14:33:27 EEST 2022
```

Еще на одном примере (по мотивам [5], но сильно переделанном) покажем, как часы RTC могут быть использованы в качестве независимого источника 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 этом примере прерывания RTC прерывают блокирующую операцию `read()` гораздо чаще периода системного тика (микросекундного диапазона). Программа допускает параметризацию запуска (не обязательными) опциями:

- `-f` — делитель частоты аппаратного чипа RTC по степени числа 2: 8192, 4096, 2048, 1024, 512, 256, ...;
- `-r` — число циклов повторения (по умолчанию 1000);
- `-n` — не переводить процесс в высокоприоритетный режим диспетчеризации FIFO:

```
$ ./rtprd -?
./rtprd: invalid option -- '?'
usage: ./rtprd [-f 2**n] [-r #] [-n]
```

А вот как организуется *периодический* таймер RTC в *микросекундном* диапазоне:

```
$ sudo ./rtprd -f2048
interrupt period set 488.28 us
worst latency was 563.51 us (on cycle 845)
mean latency was 488.24 us
$ sudo ./rtprd -f4096
interrupt period set 244.14 us
worst latency was 265.33 us (on cycle 1)
mean latency was 244.12 us
$ sudo ./rtprd -f8192 -r10000
interrupt period set 122.07 us
```

```
worst latency was 201.21 us (on cycle 7848)
mean latency was 122.04 us
```

Показателен в этом примере запуск без перевода процесса (что делается по умолчанию) в реалтайм диспетчирования FIFO (ключ `-n`) — на старом оборудовании и старых версиях ядра это увеличивало дисперсию временной латентности сразу до двух порядков, но в новых реализациях этого не наблюдается, — эффекты вытесняющей диспетчеризации трудно поддаются интерпретации.

Показанный здесь код пространства пользователя в существенной 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 обертками, но радикально зависит от использованного конкретного оборудования RTC. Для наиболее часто используемого чипа Motorola 146818 (который в таком наименовании давно уже не производится (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 или записывается требуемое значение, — так традиционно (еще со времен IBM-PC и MS-DOS) организуется обмен с данными памяти CMOS.

Время и диспетчеризация в ядре

Никто не стремится к равенству — все стремятся к превосходству,
и нетерпимость в мир приходит не от сильных,
а от слабых, раздраженных чужой силой.

П. Крусанов, «Мертвый язык»

Диспетчеризация (планирование) процессов в Linux выполняется строго *по системному таймеру* на основании *динамически* пересчитываемых приоритетов. При-

оритетов этих 140 (MAX_PRIO): 100 — реального времени + 40 приоритетов «обычной» диспетчеризации, называемых еще 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 со сформированным приоритетом nice на каждом периоде диспетчирования — в зависимости от этого значения приоритета процесса — назначается *период активности* (timeslice): от 10 до 200 системных тиков, который динамически в ходе выполнения этого процесса может быть еще расширен в пределах от 5 до 800, в зависимости от характера интерактивности процесса (процессам, активно загружающим процессор, timeslice задается ниже, а активно взаимодействующим с пользователем — *повышается*). На этом построена схема диспетчеризации процессов в Linux сложности O(1) — не зависящая по производительности от числа подлежащих планированию процессов, — которой очень гордятся разработчики ядра Linux (возможно, вполне оправданно). Но это не является на 100% *вытесняющей* диспетчеризацией... и уж точно не имеет никакого отношения к диспетчеризации реального времени, как она описана в POSIX 1003b.

> ***Примечание***
>
> Новая система диспетчеризации O(1) построена на основе двух очередей: очереди ожидающих выполнения процессов и очереди отработавших свой квант процессов. Из первой очереди выбирается поочередно следующий процесс на выполнение, и после выработки им своего кванта (если он раньше сам не перейдет в блокированное состояние) этот процесс сбрасывается во вторую. Когда очередь ожидающих опустошается, очереди (указатели на очереди) просто меняются местами: очередь отработавших становится новой очередью ожидающих, а пустая очередь ожидающих становится очередью отработавших. Но все это происходит так только при отсутствии процессов с установленной реалтайм-диспетчеризацией (RR или FIFO) с ненулевым приоритетом реального времени. До тех пор, пока в системе будет находиться хотя бы один реалтайм-процесс в состоянии готовности к выполнению (активный), ни один процесс нормального приоритета не будет выбираться на исполнение (на данном процессоре!).

Все, что касается диспетчеризации процессов в ядре, весьма интересно и сложно. Но это уже далеко выходит за пределы нашего рассмотрения... Нам же здесь важно зафиксировать, что все диспетчеризуемые изменения состояний системы происходят строго в привязке к шкале *системных тиков*.

Параллелизм и синхронизация

> Две передние, старшие, ноги вели животное в одну сторону —
> за большой головой, а две задние, младшие, ноги —
> в противоположную, за снабженным головой женским хвостом.
>
> *Милорад Павич, «Смерть святого Савы, или Невидимая сторона Луны»*

Само ядро Linux является *вытесняющим* (приемптивным, preemptive) — код ядра в состоянии вытеснить некоторые другие выполняющиеся задания, даже если они работают в режиме ядра. Среди разнообразных операционных систем (за многие

десятки лет) весьма немногие имеют вытесняющее ядро — это некоторые коммерческие реализации UNIX — например, Solaris, AIX®. Но вытеснение появляется и имеет смысл только тогда, когда возникают возможности параллелизма, когда в ядре могут одновременно существовать одновременно параллельные *потоки* выполнения.

Механизм *потоков ядра* (kernel thread — появляющийся с ядра 2.5) предоставляет средство параллельного выполнения задач в ядре. Общая особенность и механизмов потоков ядра, и примитивов для их синхронизации заключается в том, что они в принципиальной своей основе единообразны: что для пользовательского пространства, что для ядра — различаются лишь тонкие нюансы и функции доступного API их использования. Поэтому рассмотрение (и тестирование на примерах) работы механизмов синхронизации часто можно с равной степенью общности (или параллельно) проводить как в пространстве ядра, так и в пространстве пользователя, — например, так, как это сделано в [9].

Нужно отчетливо разделить два класса параллелизма (а особенно — требуемых для их обеспечения синхронизаций), *природа* которых совершенно *различного* происхождения:

- *логический параллелизм* (или квазипараллелизм), обусловленный только удобством разделения разнородных сервисов ядра, когда *один* физический поток разделяется (последовательно) между несколькими ветвями кода, создавая только иллюзию параллельности. При этом синхронизация осуществляется исключительно классическими блокирующими механизмами, когда ветвь выполнения ожидает недоступных ей ресурсов, переводясь в блокированное состояние;
- *физический параллелизм* (или реальный параллелизм), возникший только с широким распространением SMP (в виде многопроцессорности, многоядерности и даже гипертрейдинга), 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 потоков просто ожидает требуемых ресурсов, *выполняя пустые циклы ожидания* (операции `nop`). Этот второй класс (активно развиваемый, начиная примерно с 2003–2005 гг.) многократно усложняет картину происходящего (существуя одновременно с предыдущим классом) и доставляет большую головную боль разработчику. Но с ним придется считаться, прогнозируя достаточно динамичное развитие тех направлений, что уже сегодня называются *массивно-параллельными системами* (примером чего может быть модель программирования CUDA компании NVIDIA), когда от систем с двумя, четырьмя или восемью процессорами в SMP происходит переход к сотням и тысячам процессоров.

Механизм потоков ядра начал все шире и шире использоваться от версии к версии ядер 2.6.*x*, и на него даже был перенесен (переписан) 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
5.4.0-124-generic
$ ps -ef | head -n10
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 11:02 ?        00:00:03 /sbin/init splash
root           2       0  0 11:02 ?        00:00:00 [kthreadd]
root           3       2  0 11:02 ?        00:00:00 [rcu_gp]
root           4       2  0 11:02 ?        00:00:00 [rcu_par_gp]
root           5       2  0 11:02 ?        00:00:00 [kworker/0:0-events]
root           6       2  0 11:02 ?        00:00:00 [kworker/0:0H-kblockd]
root           8       2  0 11:02 ?        00:00:00 [kworker/u96:0-netns]
root          10       2  0 11:02 ?        00:00:00 [mm_percpu_wq]
root          11       2  0 11:02 ?        00:00:00 [ksoftirqd/0]
```

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 запускается непосредственно при старте ядра. Число потоков ядра может быть весьма значительным:

```
$ inxi -S
System:    Host: R420 Kernel: 5.4.0-124-generic x86_64 bits: 64 Desktop: Cinnamon 5.2.7
Distro: Linux Mint 20.3 Una
$ ps -ef | grep -F '[' | wc -l
411

$ inxi -S
System:
  Host: xenix.localdomain Kernel: 5.18.16-100.fc35.x86_64 arch: x86_64 bits: 64
    Console: pty pts/4 Distro: Fedora release 35 (Thirty Five)
$ ps -ef | grep -F '[' | wc -l
161
```

Функции организации работы с потоками (и механизмы синхронизации для них) доступны после включения заголовочного файла <linux/sched.h>. *Макрос* current возвращает указатель текущей исполняющейся задачи 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, экземпляр которой однозначно соответствует любой выполняющейся задаче: будь то поток ядра (созданный вызовом `kernel_thread()`), пользовательский процесс (точнее — главный поток этого процесса) или каждый из пользовательских потоков POSIX, созданных вызовом `pthread_create(...)` в рамках единого многопоточного процесса, — Linux не знает разницы (исполнительной) между потоками и процессами, все они порождаются одним системным вызовом `clone()`. Единственный случай, когда 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 определенную запись задачи, только это — последняя (до прерывания) выполнявшаяся задача, не имеющая никакого касательства к текущему выполняющемуся коду (текущему контексту), — `current` здесь указывает на мусор. И на это обстоятельство нужно обращать особое внимание — оно может стать предметом очень серьезных ошибок!

Потоки ядра

Уже было сказано, что ядро Linux является *вытесняющим*, в отличие от большинства ядер других операционных систем. Но вытеснение имеет смысл только в контексте наличия механизма параллельных ветвей выполнения. Этот механизм и предоставляется таким понятием, как *потоки ядра*. Потоки ядра в Linux имеют много общего с потоками пользовательского пространства (`pthread_t`) и процессами пользовательского пространства (приложениями). Объединяет их то, что каждый поток имеет свою единственную структуру `struct task_struct`, содержащую всю информацию о потоке и составляющую *контекст потока*. Все такие структуры (контекстные структуры задач) связаны в сложную циклическую динамическую структуру (списочную) — начав от *любой* структуры, можно обойти структуры *всех* задач, существующих в системе (что и делает нам команда: `ps -ef`). Это отличает все упомянутые сущности (задачи) от кода обработчиков прерываний, которые не имеют своей структуры `struct task_struct` и о которых говорят, что они выполняются в *контексте прерываний*. Указателем на структуру текущего контекста (если он есть) служит *макрос* без параметров `current`.

Создание потока ядра

Для создания нового потока ядра можно использовать «старый» API потоков[6] (его еще упоминают как низкоуровневый механизм), выполнив вызов:

```
int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags);
```

Параметры такого вызова понятны: функция потока — `fn`, нетипизированный указатель `arg` — указатель на блок данных, передаваемых этой функции, и `flags` — фла-

[6] Этот API позже был исключен из *экспортируемых* символов ядра, но он присутствует, перенесен в <linux/sched/task.h> и должен иметься в виду в реализациях для более старых ядер.

ги, обычные для API пользовательского пространства Linux, — вызова `clone()`. Возвращаемое функцией значение — это PID вновь созданного потока (если он больше нуля, а если он отрицательный, то это признак того, что что-то не заладилось, и это код ошибки).

А вот он же среди экспортируемых символов ядра (64-битная система и 32-битная ARM соответственно):

```
$ sudo grep kernel_thread$ /proc/kallsyms
ffffffff85898df0 T kernel_thread
$ sudo grep kernel_thread$ /proc/kallsyms
c011b8f0 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, — подобный тем, что массово приводятся в публикациях:

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

В принципе, такой 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
```

Последний параметр `flags` вызова `kernel_thread()` определяет детальный, побитово устанавливаемый набор тех свойств, которыми будет обладать созданный поток ядра, т. к. это вообще делается в практике Linux вызовом `clone()` (в этом месте — в 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);
```

Но с некоторых времен (я предполагаю, что это относится к ядру 3.13) такой, долго существовавший в Linux механизм создания `kernel_thread()` был объявлен устаревшим, неподдерживаемым (deprecated). В ядрах 4.19 показанный ранее модуль собирается, но не загружается по неразрешенной ссылке... а в 5.04 — уже не компилируется. Я этого не понимаю, потому что в ядре 5.04 символ `kernel_thread` присутствует в числе экспортируемых (этот вопрос требует дополнительного уточнения):

```
$ sudo cat /proc/kallsyms | grep ' T ' | grep kernel_thread$
ffffffff85898df0 T kernel_thread
```

И хотя сам описанный здесь низкоуровневый механизм не экспортируется для внешнего применения (из модулей), он используется по исходному коду ядра.

Однако на смену этому механизму пришел механизм более высокого уровня, о котором мы поговорим вскоре...

Свойства потока

Созданному потоку ядра (как и пользовательским процессам и потокам) присущ целый ряд параметров (<linux/sched.h>), часть которых будет иметь значения по умолчанию (такие, например, как параметры диспетчеризации), но которые могут

быть и изменены. Для работы с параметрами потока мы 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 всем связанном с созданием нового потока в ядре прослеживаются прямые аналогии с созданием параллельных ветвей в пользовательском пространстве (`pthread_t`), что очень сильно облегчает работу с такими механизмами.

Новый интерфейс потоков

Несколько позже для потоков был добавлен API высокого уровня (в сравнении с `kernel_thread()`), упрощающий создание и заверение потоков (<linux/kthread.h>), а вызов `kernel_thread()` был объявлен устаревшим. Всё, сказанное ранее относительно свойств и использования созданных потоков, остается в силе, поскольку новый интерфейс касается только самих фактов методов потока (создание, завершение, манипулирование) — таких методов стало существенно больше:

```
$ sudo cat /proc/kallsyms | grep ' T ' | grep ' kthread'
ffffffff858bf540 T kthread_blkcg
ffffffff858bf570 T kthread_should_stop
ffffffff858bf5d0 T kthread_should_park
ffffffff858bf600 T kthread_freezable_should_stop
ffffffff858bf700 T kthread_parkme
ffffffff858bf900 T kthread_create_on_node
ffffffff858bf960 T kthread_park
ffffffff858bfc70 T kthread_queue_work
ffffffff858bfce0 T kthread_flush_worker
ffffffff858bfd70 T kthread_delayed_work_timer_fn
ffffffff858bfe10 T kthread_flush_work
ffffffff858c0010 T kthread_cancel_work_sync
ffffffff858c0030 T kthread_cancel_delayed_work_sync
ffffffff858c00e0 T kthread_queue_delayed_work
ffffffff858c0160 T kthread_mod_delayed_work
ffffffff858c02c0 T kthread_bind
ffffffff858c0420 T kthread_create_worker
ffffffff858c04a0 T kthread_create_worker_on_cpu
ffffffff858c0510 T kthread_unpark
ffffffff858c0580 T kthread_stop
ffffffff858c0690 T kthread_destroy_worker
ffffffff858c06e0 T kthread_associate_blkcg
ffffffff858c0780 T kthread_worker_fn
ffffffff858c0980 T kthread_data
ffffffff858c09b0 T kthread_probe_data
ffffffff858c0a50 T kthread_bind_mask
```

```
ffffffff858c0a70 T kthread_create_on_cpu
ffffffff858c0b00 T kthread_set_per_cpu
ffffffff858c0b50 T kthread_is_per_cpu
ffffffff858c0b80 T kthreadd
```

Создание потока в высокоуровневом интерфейсе выполняет операция:

```
struct task_struct *kthread_create(int (*threadfn)(void *data),
                                   void *data, const char namefmt[], ...)
```

Принципиальное отличие здесь то, что возвращается не PID созданного потока, а указатель структуры задачи. Поток таким вызовом создается в *блокированном* (ожидающем) состоянии, и для запуска (выполнения) должен быть разбужен вызовом `wake_up_process()`. Поскольку это весьма частая последовательность действий, то там же (<linux/kthread.h>) определен макрос (поэтому не ищите его в именах /proc/kallsyms):

```
kthread_run(threadfn, data, namefmt, ...)
```

Приведенная операция создания потока в высокоуровневом интерфейсе также возвращает `struct task_struct*` или отрицательный код ошибки `ERR_PTR(-ENOMEM`.

Начиная с третьего параметра, функция `kthread_create()` и макрос `kthread_run()` подобны вызовам `printk()` или `sprintf()` с переменным числом параметров: третий параметр — это форматная строка (шаблон), а все последующие параметры — это значения, заполняющие этот формат. Получившаяся в итоге строка является идентификатором (именем) потока, под которым его знает ядро и под которым его показывает, например, команда `ps`.

Гораздо интереснее дело обстоит с завершением. Созданный поток может проверять (чаще всего периодически) необходимость завершения неблокирующим вызовом:

```
bool kthread_should_stop(void);
```

Если *внешний* по отношению к выполняющейся функции потока код хочет завершить поток, то он вызывает для этого потока:

```
int kthread_stop(struct task_struct*);
```

Обнаружив это (по результату `kthread_should_stop()`), функция потока завершается. Обычно в коде потоковой функции это выглядит примерно так:

```
...
while(!kthread_should_stop()) {
   // выполняемая работа потоковой функции
}
return 0;
...
```

Все сказанное гораздо проще в комплексе увидеть на примере. Для «маркирования» всех последующих примеров относительно потоков ядра сделаем такой общий включаемый файл маркеров:

prefix.c:

```
static char *sj(void) {    // метка времени
   static char s[40];
   sprintf(s, "%08ld : ", jiffies);
   return s;
}

static char *st(int lvl) { // метка потока
   static char s[40];
   sprintf(s, "%skthread [%05d:%d]", sj(), current->pid, lvl);
   return s;
}
```

И сам пример в новом интерфейсе — этот пример сложнее предыдущих, но он стоит того, чтобы его изучить детально:

mod_thr3.c

```
#include <linux/module.h>
#include <linux/delay.h>
#include <linux/jiffies.h>
#include <linux/kthread.h>

#include "../prefix.c"

static int N = 2;            // N - число потоков
module_param(N, uint, 0);

static int thread_fun1(void* data) {
   uintptr_t N = *(uint*)data;
   struct task_struct *t1 = NULL;
   printk("%s is parent [%05d]\n", st(N), current->parent->pid);
   if(--N > 0)
      t1 = kthread_run(thread_fun1, (void*)&N, "my_thread_%ld", N);
   while(!kthread_should_stop()) {
      // выполняемая работа потоковой функции
      msleep(100);
   }
   printk("%s find signal!\n", st(N));
   if(t1 != NULL) kthread_stop(t1);
   printk("%s is completed\n", st(N));
   return 0;
}

static int test_thread(void) {
   struct task_struct *t1;
```

```
    printk("%smain process [%d] is running\n", sj(), current->pid);
    t1 = kthread_run(thread_fun1, (void*)&N, "my_thread_%d", N);
    msleep(10000);
    kthread_stop(t1);
    printk("%smain process [%d] is completed\n", sj(), current->pid);
    return -1;
}

module_init(test_thread);
MODULE_LICENSE("GPL");
```

В этом коде запускается сколь угодно много потоков ядра (параметр `N=...` загрузки модуля), причем:

- потоки запускают последовательно один другого: поток `i` запускает поток `i + 1` (некоторое подобие рекурсии, но динамической...);
- все потоки используют *одну и ту же* потоковую функцию;
- весь диагностический вывод сопровождается метками времени (`jiffies`), что позволяет подробно проследить хронологию событий;
- после выдержки паузы (10 с) главный поток посылает команду завершения (`kthread_stop()`) первому запущенному им дочернему потоку, а далее потоки также по цепочке посылают такую же команду друг другу в порядке порождения.

Вот как это выглядит при исполнении:

```
$ time sudo insmod mod_thr3.ko N=3
insmod: ERROR: could not insert module mod_thr3.ko: Operation not permitted
real    0m13,246s
user    0m0,117s
sys     0m0,030s
```

Во время паузы выполнения этой команды мы можем *в другом терминале* посмотреть состояние потоков ядра в динамике:

```
$ ps -ef | grep '\[' | grep 'my_'
root     10037     2  0 00:17 ?        00:00:00 [my_thread_3]
root     10038     2  0 00:17 ?        00:00:00 [my_thread_2]
root     10039     2  0 00:17 ?        00:00:00 [my_thread_1]
```

И итоговый вывод (в журнал) по итогам выполнения приложения:

```
$ dmesg | tail -n14
[ 3039.003701] mod_thr3: loading out-of-tree module taints kernel.
[ 3039.003708] mod_thr3: module license 'GPL2' taints kernel.
[ 3039.003710] Disabling lock debugging due to kernel taint
[ 3039.003921] 4297706215 : main process [10036] is running
[ 3039.004023] 4297706215 : kthread [10037:3] is parent [00002]
[ 3039.004290] 4297706215 : kthread [10038:2] is parent [00002]
[ 3039.004376] 4297706216 : kthread [10039:1] is parent [00002]
```

```
[ 3049.092651] 4297716304 : kthread [10037:2] find signal!
[ 3049.092659] 4297716304 : kthread [10038:1] find signal!
[ 3049.196648] 4297716408 : kthread [10039:0] find signal!
[ 3049.196651] 4297716408 : kthread [10039:0] is completed
[ 3049.196676] 4297716408 : kthread [10038:1] is completed
[ 3049.196734] 4297716408 : kthread [10037:2] is completed
[ 3049.196752] 4297716408 : main process [10036] is completed
```

Здесь хорошо видно, что:

- порядок, в котором потоки создаются, и порядок, в котором они получают сигнал на завершение, — *совпадают*...
- но порядок фактического завершения — в точности *обратный*, потому что на выполнении `kthread_stop()` поток блокируется и ожидает завершения дочернего потока, и только после этого имеет право завершиться;
- видны задержки с малой дискретностью (порядка 100 мс — см. код) между получением команды завершения и отправкой ее из основного цикла потоковой функции дальше по иерархии потоков;
- видно, что для выполняющегося потока родительским потоком становится `PID=2` (демон `kthreadd`), хотя все происходящее мы и наблюдаем еще при выполнении запускающего *процесса* `insmod` (в функции инициализации модуля), — т. е. для созданных потоков выполняется операция *демонизации*.

Важно, что всё это единообразно работает в разных процессорных архитектурах, в 32- и 64-битных реализациях разных дистрибутивов, и особенно важно (мы будем говорить об этом позже, рассуждая об отладке модулей) — единообразно как в нативной инсталляции Linux («в железе»), так и инсталлированной *в виртуальной машине* под VirtualBox:

```
$ sudo inxi -MCS
System:    Host: lmde32 Kernel: 5.10.0-16-686 i686 bits: 32 Console: tty 2 Distro: LMDE 5 Elsie
Machine:   Type: Virtualbox System: innotek product: VirtualBox v: 1.2 serial: N/A
           Mobo: Oracle model: VirtualBox v: 1.2 serial: N/A BIOS: innotek v: VirtualBox date:
                                                                                    12/01/2006
CPU:       Info: Single Core model: Intel Xeon E5-2470 v2 bits: 32 type: MCP L2 cache: 25 MiB
           Speed: 2451 MHz min/max: N/A Core speed (MHz): 1: 2451
$ time sudo insmod mod_thr3.ko N=5
insmod: ERROR: could not insert module mod_thr3.ko: Operation not permitted
real    0m10,212s
user    0m0,012s
sys     0m0,009s
$ ps -ef | grep '\[' | grep 'my_'
root      6750     2  0 00:22 ?        00:00:00 [my_thread_5]
root      6751     2  0 00:22 ?        00:00:00 [my_thread_4]
root      6752     2  0 00:22 ?        00:00:00 [my_thread_3]
root      6753     2  0 00:22 ?        00:00:00 [my_thread_2]
root      6754     2  0 00:22 ?        00:00:00 [my_thread_1]
```

```
$ dmesg | tail -n17
[34685.460246] 08600620 : main process [6749] is running
[34685.461990] 08600621 : kthread [06750:5] is parent [00002]
[34685.462015] 08600621 : kthread [06751:4] is parent [00002]
[34685.462033] 08600621 : kthread [06752:3] is parent [00002]
[34685.462049] 08600621 : kthread [06753:2] is parent [00002]
[34685.462065] 08600621 : kthread [06754:1] is parent [00002]
[34695.608415] 08603159 : kthread [06750:4] find signal!
[34695.608424] 08603159 : kthread [06751:3] find signal!
[34695.608428] 08603159 : kthread [06752:2] find signal!
[34695.608431] 08603159 : kthread [06753:1] find signal!
[34695.608434] 08603159 : kthread [06754:0] find signal!
[34695.608435] 08603159 : kthread [06754:0] is completed
[34695.608494] 08603159 : kthread [06753:1] is completed
[34695.608513] 08603159 : kthread [06752:2] is completed
[34695.608524] 08603159 : kthread [06751:3] is completed
[34695.608557] 08603159 : kthread [06750:4] is completed
[34695.608674] 08603159 : main process [6749] is completed
```

Как уже легко видеть (даже без детального анализа), такой подход существенно упрощает и синхронизацию завершения потоков, к рассмотрению которой мы и переходим...

Синхронизация завершения

В предыдущем примере, попутно с основной иллюстрацией, была показана синхронизированная работа потоков: порождавшие потоки сигнализировали порожденным о необходимости их завершения, после чего дожидались этого их завершения. Это один из возможных видов синхронизации потоков. Другой, очень часто возникающий случай — это так называемый *пул потоков* (статический или динамический, но сейчас мы будем говорит о статическом пуле): для распараллеливания работы создается несколько однотипных потоков, с *единой* для всех потоковой функцией. После последовательного запуска всех потоков пула порождающая единица ожидает завершения работы *всех* порожденных потоков.

В следующем примере показан типовой и не совсем понятный при первом знакомстве трюк, но повсеместно применяемый для решения такой задачи (в рассматриваемом случае нас совершенно не интересует то, на каких примитивах синхронизации выполняется синхронизация завершения потоков, — для этого могут использоваться разные примитивы, здесь же важен принцип того, как это делается):

mod_for.c

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

```
#include <linux/jiffies.h>
#include <linux/semaphore.h>

#include "../prefix.c"

#define NUM 3
static struct completion finished[NUM];

#define DELAY 1000
static int thread_func(void* data) {
   uint num = *(uint*)data;
   printk("! %s is running\n", st(num));
   msleep(DELAY - num * 200);
   complete(finished + num);
   printk("! %s is finished\n", st(num));
   return 0;
}

#define IDENT "for_thread_%d"
static int test_mlock(void) {
   struct task_struct *t[NUM];
   uint i;
   for(i = 0; i < NUM; i++)
      init_completion(finished + i);
   for(i = 0; i < NUM; i++)
      t[i] = kthread_run(thread_func, (void*)&i, IDENT, i);
   for(i = 0; i < NUM; i++)
      wait_for_completion(finished + i);
   printk("! %s is finished\n", st(NUM));
   return -1;
}

module_init(test_mlock);
MODULE_LICENSE("GPL");
```

Пояснение

Включаемый файл `prefix.c` содержит описания диагностических функций `sj()` и `st()`, которые мы уже видели в предыдущем примере.

Дочерние потоки ядра в этом примере создаются *последовательно* (от нулевого до второго) в цикле. Завершаться они могут в реальной ситуации в произвольные времена и в произвольном порядке — в примере же искусственно смоделирована самая неблагоприятная ситуация, при которой потоки завершаются в порядке, обратном их порождению (но это могут быть и случайные моменты времени). Однако ожидается завершение потоков (программной единицы, ожидающей в блокированном состоянии) опять-таки всё в том же *последовательном* порядке — от нулевого до второго:

```
for(i = 0; i < NUM; i++)
   wait_for_completion(finished + i);
```

Это не ошибка — это общеупотребимая практика! Это и есть то, что я ранее назвал «трюк»... Нам нужно ожидание завершения *всех* порожденных потоков. Независимо от порядка завершения, цикл, освободившись на очередном блокирующем ожидании завершения *i*-го потока, просто не будет блокировать ожидающий код на последующих номерах уже *ранее* завершившихся потоков $j > i$, — он не задерживается и мгновенно проскакивает их в цикле. Полностью такой цикл выйдет из блокированных состояний *только* когда завершатся *все* порожденные потоки.

Это прямая аналогия с тем трюком, который очень часто применяют в программировании потоков POSIX `pthread_t`, и он прекрасно известен программистам, работающим с потоками пользовательского пространства. А иначе и быть не могло! Потому что *все* потоки — как ядра, так и пользовательские, — представлены записями `struct task_struct`, которые связаны в *единый* циклический список внутри ядра. С точки зрения ядра системы именно *потоки* являются основными runtime-динамическими единицами в системе (задача, task), а не *процессы*, которые кажутся нам таковыми с позиции внешнего наблюдателя или командного интерпретатора Shell.

Вот как это выглядит при исполнении:

```
$ time sudo insmod mod_for.ko
insmod: ERROR: could not insert module mod_for.ko: Operation not permitted
real    0m1,077s
user    0m0,009s
sys     0m0,021s
$ dmesg | tail -n6
[33242.368422] ! 32942846 : kthread [17213:2] is running
[33242.368427] ! 32942846 : kthread [17214:0] is running
[33243.010874] ! 32943488 : kthread [17213:2] is finished
[33243.202412] ! 32943680 : kthread [17212:1] is finished
[33243.393998] ! 32943872 : kthread [17214:0] is finished
[33243.394007] ! 32943872 : kthread [17211:3] is finished
```

В завершение посмотрим более реалистичный пример (см. папку thread/loop в сопровождающем книгу файловом архиве), который часто востребован на практике: модуль после загрузки должен периодически, с периодом `period` секунд, выполнять некоторое действие... не важно какое — у нас это будет просто вывод приветственного сообщения с меткой времени (`jiffies`). Для того чтобы расширить возможности, мы сделаем то, что уже умеем, — добавим модулю интерфейс в /proc, через который сможем записывать командой `echo` «на ходу» новые значения `period` и считывать текущее установленное значение командой `cat`. Очевидно, что мы не можем создать цикл действия (или вызвать функцию, выполняющую такой цикл) непосредственно из функции инициализации (загрузки) модуля, а должны это делать в отдельном потоке. И сосредоточимся на синхронизации с этим потоком при загрузке, а еще более — при завершении модуля.

Основной код модуля:

mod_loop.c

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/proc_fs.h>
#include <linux/uaccess.h>
#include <linux/jiffies.h>
#include <linux/kthread.h>
#include <linux/delay.h>

MODULE_LICENSE("GPL");
static int  __init proc_init(void);
static void __exit proc_exit(void);
module_init(proc_init);
module_exit(proc_exit);

static uint period = 5;          // целочисленный парамер - период
module_param(period, uint, 0);

#include "fops_rw.c"             // операции чтения/записи

#if LINUX_VERSION_CODE < KERNEL_VERSION(5,9,0)
const struct file_operations node_fops = {
   .owner = THIS_MODULE,
   .read  = node_read,
   .write  = node_write
#else
const struct proc_ops node_fops = {
   .proc_read  = node_read,
   .proc_write = node_write
#endif
};

static struct task_struct* ts;
static struct completion final;   // флаг завершения

static int loop(void *data) {
   printk("!! thread %u is running\n", task_pid_nr(ts));
   while(!kthread_should_stop()) {
      printk("!! [%lu] Hello from module!\n", jiffies);
      ssleep(period);
   }
   printk("!! [%lu] thread %u finished\n",  jiffies, task_pid_nr(ts));
   complete(&final);
   return 0;
}
```

```
struct proc_dir_entry *own_proc_dir;
struct proc_dir_entry *own_proc_node;

#define NAME_DIR  "mod_loop"
#define NAME_NODE "period"

static int __init proc_init( void ) {
   own_proc_dir = proc_mkdir(NAME_DIR, NULL);
   if(NULL == own_proc_dir) {
      printk("! can't create directory /proc/%s\n", NAME_NODE);
      return -ENOENT;
   }
   printk("! /proc/%s directory created\n", NAME_DIR);
   own_proc_node = proc_create(NAME_NODE, S_IFREG | S_IRUGO | S_IWUGO,
                               own_proc_dir, &node_fops);
   if(NULL == own_proc_node) {
      printk("! can't create node /proc/%s/%s\n", NAME_DIR, NAME_NODE);
      remove_proc_entry(NAME_DIR, NULL);
      return -ENOENT;
   }
   printk("! /proc/%s/%s created\n", NAME_DIR, NAME_NODE);
   ts = kthread_run(loop, (void*)&period, "loop_thread");
   return 0;
}

static void __exit proc_exit(void) {
   init_completion(&final);
   kthread_stop(ts);
   wait_for_completion(&final);
   printk("! [%lu] child thread finished\n", jiffies);
   remove_proc_entry(NAME_NODE, own_proc_dir);
   remove_proc_entry(NAME_DIR, NULL);
   printk("! /proc/%s/%s removed\n", NAME_DIR, NAME_NODE);
}
```

Операции чтения/записи с /proc вынесены в отдельный файл:

fops_rw.c

```
#define LEN_MSG 160
static char buf_msg[LEN_MSG + 1];

static ssize_t node_read(struct file *file, char *buf,
                         size_t count, loff_t *ppos) {
   static bool first = false;
   int res;
```

```
   if ((first = !first)) {
      sprintf(buf_msg, "%u", period);
      printk("! return %lu bytes: <%s>\n", strlen(buf_msg), buf_msg);
      strcat(buf_msg, "\n");
      res = copy_to_user((void*)buf, buf_msg, strlen(buf_msg));
      return strlen(buf_msg);
   } else {
      *ppos = 0;
      printk("! return EOF\n");
      return 0;
   }
}

static ssize_t node_write(struct file *file, const char *buf,
                          size_t count, loff_t *ppos) {
   int res;
   char *eptr;
   uint len = count < LEN_MSG ? count : LEN_MSG, input;
   res = copy_from_user(buf_msg, (void*)buf, len);
   buf_msg[len] = '\0';
   input = simple_strtoul(buf_msg, &eptr, 0);
   if (0 == input) return -EINVAL;
   period = input;
   printk("! write: %lu bytes - %u\n", (uintptr_t)(eptr - buf_msg), period);
   return len;
}
```

С запуском рабочего потока относительно просто: функция инициализации модуля запускает поток, создает управляющий интерфейс `/proc/mod_loop/period` и сама завершается. Гораздо хуже с завершением...

- когда затребована *выгрузки* модуля (командой `rmmod`), запускается функция завершения `proc_exit()` ...
- но она не может просто так взять и остановить выполняющийся поток, хотя и знает его идентификатор (`struct task_struct*`)...
- она через `kthread_stop()` *сигнализирует* потоку: «пора, мол, завершаться»...
- поток *видит* это уведомление через `kthread_should_stop()`, но не спешит и завершается тогда, когда он может это сделать (в нашем случае — после истечения времени текущего цикла ожидания);
- но функция `proc_exit()` не имеет права *завершать* (выгружать) модуль до полного завершения выполняющегося потока — для этого она инициализировала (раньше) флаг завершения (типа `struct completion`)... и *ждет* его на вызове `wait_for_completion()`;
- поток *сигнализирует*, что он готов завершиться (вот-вот завершится), через вызов `complete()` для флага `struct completion`;

- и только получив подтверждение завершения потока, функция `proc_exit()` может выгрузить модуль.

Теперь, как это выглядит в деле:

```
$ sudo insmod mod_loop.ko period=10
$ ps -ef | grep loop_
root       26911       2  0 17:59 ?        00:00:00 [loop_thread]
$ ls -l /proc/mod_loop/
итого 0
-rw-rw-rw- 1 root root 0 авг 29 01:08 period
$ tree /proc/mod_loop/
/proc/mod_loop/
└── period
0 directories, 1 file
$ ls -l /proc/mod_loop/period
-rw-rw-rw- 1 root root 0 авг 29 01:08 /proc/mod_loop/period
$ dmesg | tail -n5
[39765.110729] ! /proc/mod_loop directory created
[39765.110734] ! /proc/mod_loop/period created
[39765.111051] !! thread 26911 is running
[39765.111053] !! [4334434560] Hello fron module!
[39775.606061] !! [4334445056] Hello fron module!
```

Хорошо видно (последние 2 строки), что интервал между сообщениями равен 10 секундам. Контролируем это значение:

```
$ cat /proc/mod_loop/period
10
$ dmesg | tail -n7
[39765.111053] !! [4334434560] Hello fron module!
[39775.606061] !! [4334445056] Hello fron module!
[39783.855500] ! return 2 bytes: <10>
[39783.855533] ! return EOF
[39785.845508] !! [4334455296] Hello fron module!
[39796.084959] !! [4334465536] Hello fron module!
[39806.324395] !! [4334475776] Hello fron module!
```

Изменяем период:

```
$ echo 20 > /proc/mod_loop/period
$ cat /proc/mod_loop/period
20
```

И наблюдаем изменения (3 последние строки):

```
$ dmesg | tail -n7
[39826.803296] !! [4334496256] Hello fron module!
[39828.006697] ! write: 2 bytes - 20
[39831.319768] ! return 2 bytes: <20>
[39831.319798] ! return EOF
```

```
[39837.042731] !! [4334506496] Hello fron module!
[39857.521630] !! [4334526976] Hello fron module!
[39878.000528] !! [4334547456] Hello fron module!
```

Особенно интересно наблюдать завершение работы модуля:

```
$ dmesg | tail -n4
[39918.958315] !! [4334588416] Hello fron module!
[39939.437194] !! [4334608896] thread 26911 finished
[39939.437225] ! [4334608896] child thread finished
[39939.437236] ! /proc/mod_loop/period removed
```

Синхронизация в коде

Синхронизация родительского потока с завершением потока порожденного — это очень важный случай: завершение родительского потока прежде завершения им порожденного создает «бесхозный», открепленный поток и вызывает серьезные проблемы даже в пользовательских процессах, а в ядре рано или поздно чревато крахом операционной системы. Это очень грубая ошибка! И приводимое в некоторых публикациях ее решение путем введения фиксированных (больших) пауз ожидания («...дочерний поток за это время наверняка завершится») является безответственной глупостью. Все такие проблемы параллелизма должны решаться через *синхронизацию*. Но синхронизация по завершении является далеко не единственным случаем ее применения. Спектр использования синхронизации очень широк и разнообразен. А из-за этого разнообразия вводится в обиход много разных *примитивов* синхронизации.

Таких примитивов — как теоретически проработанных, так и конкретно используемых и доступных в ядре Linux, — существует множество, 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 таких примитивов остается одинаковой — в том ровно виде, как ее впервые описал Э. Дейкстра в своей знаменитой работе 1968 года «Взаимодействие последовательных процессов».

Критические секции кода и защищаемые области данных

Для решения задач синхронизации в ядре Linux существует множество механизмов синхронизации и появляются все новые и новые — некоторые из механизмов вводятся даже для поддержки единичных потребностей. Условно по функциональному использованию примитивы синхронизации можно разделить на следующие (хотя такое разделение часто оказывается весьма условным):

- примитивы для защиты фрагментов исполняемого кода (критических секций) от одновременного (или псевдоодновременного) исполнения. Классические примеры: мьютекс, блокировки чтения/записи;

- примитивы для защиты областей данных от несанкционированного изменения: атомарные переменные и операции, счетные семафоры.

Механизмы синхронизации

Обычно все предусмотренные версией ядра примитивы синхронизации доступны после включения заголовочного файла <linux/sched.h>. Мы рассмотрим здесь только некоторые из таких механизмов:

- 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 синхронизации далее проведено как раз в обратном порядке — с ожидания завершения, потому что это естественным образом продолжает начатое ранее 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 механизмами синхронизации — 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. В ядре нет аналога функции ожидания завершения потока — вместо нее требуется явно использовать механизмы синхронизации (аналогичные предусмотренному 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 ее 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 приведенного ранее примера запуска потока — для сравнения):

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!):

```
$ time sudo insmod mod_thr2.ko param=10
insmod: ERROR: could not insert module mod_thr2.ko: Operation not permitted
real    0m10,095s
user    0m0,011s
sys     0m0,008s
$ ps -A | grep my_thread
  16446 ?        00:00:00 my_thread
$ dmesg | tail -n4
[14166.035608] create new kernel thread [16446]
[14166.035613] child process [16446] is running
[14176.075824] child process [16446] is completed
[14176.075881] main process [16445] is completed
```

```
$ sudo cat /var/log/kern.log | tail -n4
Aug 14 15:14:59 R420 kernel: [14166.035608] create new kernel thread [16446]
Aug 14 15:14:59 R420 kernel: [14166.035613] child process [16446] is running
Aug 14 15:15:09 R420 kernel: [14176.075824] child process [16446] is completed
Aug 14 15:15:09 R420 kernel: [14176.075881] main process [16445] is completed
```

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***
>
> Все 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()`, при котором все различия создания определяются набором флагов вида `CLONE_*` для создаваемой задачи (последний параметр `kernel_thread()` нашего примера).

Объект на сегодня описан (<linux/completion.h>) как очередь FIFO (стек) потоков, ожидающих события:

```
/*
 * struct completion - structure used to maintain state for a "completion"
 * This is the opaque structure used to maintain the state for a "completion".
 * Completions currently use a FIFO to queue threads that have to wait for
 * the "completion" event.
 */
struct completion {
        unsigned int done;
        wait_queue_head_t wait;
};
```

Основные операции:

- это операции блокирования и ожидания освобождения:

```
extern void wait_for_completion(struct completion*);
extern unsigned long wait_for_completion_timeout(struct completion *x,
                                                 unsigned long timeout);
extern bool try_wait_for_completion(struct completion *x);
```

- а это операции освобождения ожидающих потоков: одного на вершине стека, последнего пришедшего или всех ожидающих в очереди соответственно:

```
extern void complete(struct completion*);
extern void complete_all(struct completion*);
```

Атомарные переменные и операции

Атомарные переменные — это наименее ресурсоемкие средства обеспечения атомарного выполнения операций (там, где их минимальных возможностей достаточно). Реализуются они 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 и б) атомарные операции скрывают различия между реализациями для различных аппаратных платформ.

Функции, реализующие атомарные операции, можно разделить на две 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 две группы по типу объекта: а) оперирующие с целочисленными значениями (арифметические) и б) оперирующие с последовательным набором битов. Атомарных операций в итоге великое множество, и далее обсуждаются только некоторые из них.

Битовые атомарные операции

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-битных процессоров это 31, для 64-битных — 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

Арифметические атомарные операции реализуются в машинно-зависимом коде и описаны, например, здесь:

```
$ 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 значения `i` переменной типа `atomic_t`;
- `int atomic_read(atomic_t *v);` — считывание значения в целочисленную переменную `v`;
- `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 вызывают сложностей (аналогичны 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 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

Локальные переменные (данные) процессора — это переменные, закрепленные за процессором (per-CPU data). Определены они в <linux/percpu.h>. Основное достоинство таких переменных в том, что если некоторую функциональность можно разумно между ними распределить, то они не потребуют взаимных блокировок доступа в SMP. API, предоставляемые для работы с локальными данными процессора, на время работы с этими переменными запрещают вытеснение в режиме ядра.

Вторым свойством локальных данных процессора является то, что они позволяют существенно уменьшить недостоверность данных, хранящихся в кеше. Это происходит потому, что процессоры поддерживают свои кеши в синхронизированном состоянии: если один процессор начинает работать с данными, которые находятся в кеше другого процессора, то первый процессор должен обновить содержимое своего кеша. Постоянное аннулирование находящихся в кеше данных, именуемое *перегрузкой кеша* (cash thrashing), существенно снижает производительность системы (до 3–4 раз). Использование данных, связанных с процессорами, позволяет приблизить эффективность работы с кешем 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 остается 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 гонки за ресурс.

Новая модель

Новая модель введена с расчетом на будущее развитие и на обслуживание весьма большого числа процессоров в системе — она упрощает работу с локальными переменными процессора, но на настоящее время еще не так широко используется.

- *Статические определения* (на этапе компиляции):

```
DEFINE_PER_CPU(type, name);
```

  Здесь создается переменная типа `type` с именем `name`, которая имеет отдельный экземпляр для каждого процессора в системе. Если необходимо объявить такую переменную с целью избежать предупреждения 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;
  - `per_cpu(name, int cpu);` — еще один вызов, который возвращает L-value экземпляра локальной переменной другого процессора. Этот вызов не запрещает вытеснение кода в режиме ядра и не обеспечивает никаких блокировок — для его использования необходимы внешние блокировки в коде.

  Пример статически определенной переменной:

```
DECLARE_PER_CPU(long long, xxx);
get_cpu_var(xxx)++;
put_cpu_var(xxx);
```

- *Динамические определения* (на этапе выполнения) — это другая группа API. Они динамически выделяют области фиксированного размера, закрепленные 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 экземпляр 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 экземпляр указанной переменной на *другом* процессоре.

Пример динамически определенной переменной:

```
long long *xxx = (long long*)alloc_percpu(long long);
++*get_cpu_ptr(xxx);
put_cpu_var(xxx);
```

Требование неблокируемости кода, работающего с локальными данными процесса, остается актуальным и в этом случае.

> ***Примечание***
>
> Легко видеть, что новая модель (будь это группа API, работающая с самими переменными или с указателями на них) не так уж значительно отличается от предыдущей: устранена необходимость явного индексирования массива экземпляров по номеру процессора — это делается внутренними скрытыми механизмами, и окончательно возвращается уже индексированный экземпляр, связанный с текущим процессором.

Блокировки

Различные виды блокировок используются для того, чтобы оградить критический участок кода от одновременного исполнения. В этом смысле блокировки гораздо ближе к защите участков кода, чем к защите областей данных, хотя семафоры, например небинарные, используются главным образом именно для ограничения доступа к данным — классические задачи производители–потребители.

До появления и широкого распространения SMP, когда параллелизмы были квазипараллелизмами, блокировки использовались в своем 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», ожидая освобождения ресурса, «пустые» циклы. В этом случае процессор не освобождается на выполнение другого ожидающего процесса в системе, а продолжает активное выполнение («пустых» циклов) в контексте текущего процесса.

Эти два рода блокировок (каждый из которых включает несколько подвидов) принципиально отличаются:

- возможностью использования — пассивно заблокировать (переключить контекст) можно только такую последовательность кода, которая имеет свой собственный контекст (запись задачи), куда позже можно вернуться (активировать процесс), — в обработчиках прерываний или тасклетах это не так;
- эффективностью — активные блокировки не всегда проигрывают пассивным в производительности. Переключение контекста в системе — это очень трудоемкий процесс, поэтому для ожидания короткого интервала времени активные блокировки могут оказаться даже эффективнее, чем пассивные.

Семафоры и мьютексы

Семафоры ядра определены в <linux/semaphore>. Так как задачи, конфликтующие 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, поскольку 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 задействовать 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 задачи в любой момент времени, количество задач (`count`), которым разрешено одновременно удерживать семафор (владеть семафором), может быть задано при декларации переменной семафора (поэтому небинарные семафоры еще называют для определенности *счетными семафорами*):

```
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 случае семафоры называются *бинарными семафорами*.

Во многом ту же функциональность, что и бинарные семафоры, обеспечивают *мьютексы* (`mutex`) — взаимоисключающие блокировки (потому что они гарантируют взаимоисключающий доступ, mutual exclusion). Мьютексы определяются в <linux/mutex.h> (исключая отладочную часть структуры):

```
struct mutex {
      atomic_lomg_t     owner
      spinlock_t        wait_lock;
      struct list_head  wait_list;
};
```

Бинарные семафоры и мьютексы используются зачастую для обеспечения взаимоисключающего доступа к фрагментам кода, называемым *критической секцией*, — в таком качестве и состоит их наиболее частое использование. Тогда как счетные семафоры чаще применяются для защиты монопольного доступа к элементам данных.

При всей схожести бинарного светофора и мьютекса (их даже путают по использованию), они принципиально различаются: у заблокированного мьютекса, в отличие от семафора, всегда имеется *владелец* — поток, заблокировавший (захвативший) мьютекс, и *только* этот поток может освободить мьютекс.

> ***Примечание***
>
> Независимо от того, определено ли поле владельца, захватившего мьютекс (как это делается по-разному в различных POSIX-совместимых ОС), принципиальными особенностями мьютекса, вытекающими из его логики, в отличие от счетного семафора, будет то, что: а) у захваченного мьютекса всегда будет и единственный владелец, его захвативший, и б) освободить блокированные на мьютексе потоки (освободить мьютекс) может только один владеющий мьютексом поток. В случае счетного семафора освободить блокированные на семафоре потоки может любой из потоков, владеющий семафором.

Из того, что захваченный мьютекс может освободить только его владелец (поток), вытекает очень важное следствие: повторный захват того же мьютекса в потоке (например рекурсивно) будет приводить к бесконечному блокированию потока (deadlock, мертвые объятия).

Как и большинство объектов ядра, и мьютексы, и семафоры могут создаваться и инициализироваться *статически* (в момент объявления в коде) и *динамически* (выполнением инициализации в ходе выполнения).

Статическое определение бинарного семафора выполняется макросом:

```
DEFINE_SEMAPHORE(name);
```

Этим создается `struct semaphore` с указанным именем `name`.

Счетный семафор должен инициализироваться динамически.

Для создания взаимоисключающей блокировки (mutex), что используется наиболее часто, служит запись:

```
DEFINE_MUTEX(name);
```

Этим создается `struct mutex` с указанным именем `name`.

Но чаще семафоры создаются динамически — как часть больших структур данных. В таком случае для инициализации счетного семафора используется функция:

```
inline void sema_init(struct semaphore *sem, int val);
```

А вот такая же инициализация для мьютекса в разблокированном состоянии — макрос:

```
mutex_init(name)
```

В операционной системе Linux для захвата семафора 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 блокируется, помещается в очередь ожидания, и процессор переходит к выполнению других задач. Чтобы освободить семафор (после завершения выполнения критического участка), используется метод `up()` — его выполнение увеличивает счетчик семафора на единицу.

Операции над семафорами:

```
void down(struct semaphore *sem);
void up(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);
```

Здесь:

- `down()` — переводит задачу в блокированное состояние ожидания с флагом `TASK_UNINTERRUPTIBLE`. Это часто может быть нежелательно, т. к. процесс, который ожидает освобождения семафора, не будет отвечать на сигналы;
- `up()` — инкрементирует счетчик семафора, и если есть блокированные на семафоре потоки, то *один* из них может захватить блокировку (принципиальным является то, что *невозможно предсказать*, какой конкретно поток из числа заблокированных будет освобожден);
- `down_interruptible()` — выполняет попытку захватить семафор. Если попытка оказывается неудачной, то задача переводится в блокированное состояние с флагом `TASK_INTERRUPTIBLE` (в структуре задачи). Это 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_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.

Для мьютексов существует аналогичный по смыслу выполняемых действий (основной) набор операций:

```
void mutex_lock(struct mutex *lock);
void mutex_unlock(struct mutex *lock);
bool mutex_is_locked(struct mutex *lock);
int mutex_trylock(struct mutex *lock);
int mutex_lock_interruptible(struct mutex *lock);
int mutex_lock_killable(struct mutex *lock);
```

Что означает синхронизация для параллельных вычислений, а, кроме того, первый пример использования мьютексов, мы рассмотрим на следующем коде:

mlock.c

```
#include <linux/module.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/mutex.h>

#include "../prefix.c"

#define num 4                    // num - число рабочих потоков

static ulong rep = 10000;        // rep - число повторений (объем работы)
module_param(rep, ulong, 0);

static bool sync = 1;            // sync - синхронизация: 0 - да, 1 - нет
module_param(sync, bool, 0);

DEFINE_MUTEX(mtx);
static ulong locked = 0;         // накапливаемая сумма

static struct completion finished[num];

static int thread_func(void* data) {
   uint i, n = *(uint*)data;     // порядковый номер потока (локальный!)
   printk("! %s is started\n", st(n));
   for(i = 0; i < rep; i++) {
      if(!sync) mutex_lock(&mtx);
      locked++;
      if(!sync) mutex_unlock(&mtx);
   }
   printk("! %s is finished\n", st(n));
   complete(finished + n);
```

```
   return 0;
}

static int test_mlock(void) {
   unsigned long j1 = jiffies;   // время точки старта
   struct task_struct *t[num];
   uint i;
   printk("! потоков = %d | повторов = %ld | синхронизация: %s\n",
          num, rep, sync ? "нет" : "да" );
   for(i = 0; i < num; i++)
      init_completion(finished + i);
   for(i = 0; i < num; i++)
      #define IDENT "mlock_thread_%d"
      t[i] = kthread_run(thread_func, (void*)&i, IDENT, i);
   for(i = 0; i < num; i++)
      wait_for_completion(finished + i);
   j1 = jiffies - j1;
   printk("! время работы: %ld.%02ld сек. (%ld), сумма=%ld\n",
           j1 / HZ, (j1 * 100) / HZ % 100, j1, locked);
   return -1;
}

module_init(test_mlock);
MODULE_LICENSE("GPL");
```

Логика кода проста: в четыре параллельных потока (фиксированное число для простоты) с идентичной функцией потока мы инкрементируем целочисленную переменную `locked` в большом числе циклов повторения. Причем в качестве критической секции кода мы сознательно выбираем единичную операцию *инкремента* (исходя из того, что в некоторых публикациях утверждается, что инкремент — это атомарная операция). Но выполнять критическую секцию кода мы можем блокированием ее мьютексом `mtx` (или нет) — в зависимости от опции загрузки модуля `sync` (0 — да, 1 — нет, по умолчанию — да). Теперь смотрим:

```
$ time sudo insmod mlock.ko sync=0 rep=10000000
insmod: ERROR: could not insert module mlock.ko: Operation not permitted
real    0m10,166s
user    0m0,042s
sys     0m0,013s
$ dmesg | tail -n10
[ 4026.438466] ! потоков = 4 | повторов = 10000000 | синхронизация: да
[ 4026.438570] ! 4298693807 : kthread [19517:1] is started
[ 4026.438790] ! 4298693807 : kthread [19518:2] is started
[ 4026.443870] ! 4298693813 : kthread [19519:3] is started
[ 4026.450833] ! 4298693820 : kthread [19520:0] is started
[ 4036.071615] ! 4298703440 : kthread [19520:0] is finished
[ 4036.250586] ! 4298703619 : kthread [19519:3] is finished
```

```
[ 4036.527005] ! 4298703896 : kthread [19517:1] is finished
[ 4036.539002] ! 4298703908 : kthread [19518:2] is finished
[ 4036.539019] ! время работы: 10.10 сек. (10101), сумма=40000000
```

Это было выполнение с синхронизацией — как и ожидалось, 4 × 10 000 000 инкрементов создали значение 40 000 000. А затем, раз все так хорошо, и, понадеявшись на авось, мы уберем синхронизацию:

```
$ time sudo insmod mlock.ko sync=1 rep=10000000
insmod: ERROR: could not insert module mlock.ko: Operation not permitted
real    0m0,597s
user    0m0,042s
sys     0m0,013s
```

Получается, что время выполнения уменьшилось почти в 20 раз! Это большая победа оптимизации... Но:

```
$ dmesg | tail -n10
[ 4009.641894] ! потоков = 4 | повторов = 10000000 | синхронизация: нет
[ 4009.642139] ! 4298677010 : kthread [19462:1] is started
[ 4009.642161] ! 4298677011 : kthread [19463:2] is started
[ 4009.651142] ! 4298677020 : kthread [19464:3] is started
[ 4009.658137] ! 4298677027 : kthread [19465:0] is started
[ 4010.144128] ! 4298677513 : kthread [19463:2] is finished
[ 4010.168708] ! 4298677537 : kthread [19465:0] is finished
[ 4010.172651] ! 4298677541 : kthread [19462:1] is finished
[ 4010.172889] ! 4298677541 : kthread [19464:3] is finished
[ 4010.172908] ! время работы: 0.53 сек. (531), сумма=20370339
```

... но из ожидаемых 40 000 000 выполнились только 20 370 339, т. е. около 50%. Это и есть наблюдаемые эффекты синхронизации (или ее отсутствия):

- любой параллельный доступ к данным, который может хоть каким-то образом модифицировать эти данные, без синхронизации порождает *грубейшие* ошибки;
- синхронизации — дорогие операции (в смысле производительности)! Правильное использование синхронизации может увеличивать время выполнения на порядок или даже на порядки... И здесь есть простор для оптимизации.

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 переключения 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. Достаточно много определений spinlock_t разбросано по нескольким заголовочным файлам:

```
$ ls -w80 /lib/modules/`uname -r`/build/include/linux/spinlock*
/lib/modules/5.4.0-124-generic/build/include/linux/spinlock_api_smp.h
/lib/modules/5.4.0-124-generic/build/include/linux/spinlock_api_up.h
/lib/modules/5.4.0-124-generic/build/include/linux/spinlock.h
/lib/modules/5.4.0-124-generic/build/include/linux/spinlock_types.h
/lib/modules/5.4.0-124-generic/build/include/linux/spinlock_types_up.h
/lib/modules/5.4.0-124-generic/build/include/linux/spinlock_up.h

typedef struct {
   raw_spinlock_t raw_lock;
...
} spinlock_t;
```

Для инициализации spinlock_t (и родственного типа rwlock_t, о котором детально далее) раньше (и в литературе) использовались константные макросы:

```
spinlock_t lock = SPIN_LOCK_UNLOCKED;
rwlock_t lock = RW_LOCK_UNLOCKED;
```

Но сейчас мы можем читать в комментариях:

```
// SPIN_LOCK_UNLOCKED and RW_LOCK_UNLOCKED defeat lockdep state tracking and are hence
deprecated.
```

То есть эти макроопределения объявлены неподдерживаемыми и могут быть исключены в любой последующей версии. Поэтому для определения и инициализации spinlock_t мы сейчас используем новые макросы (эквивалентные по смыслу записанным ранее) вида:

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

Это, как и обычно, статические определения отдельных (автономных) переменных типа spinlock_t. И для них, так же как и для других примитивов, может быть применена динамическая инициализация ранее объявленной переменной (чаще эта переменная — поле в составе более сложной структуры):

```
void spin_lock_init(spinlock_t *sl);
```

Основной интерфейс spinlock_t (основная пара операций: захват и освобождение):

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

Если при компиляции ядра не установлено SMP (использование многопроцессорности) и не конфигурировано вытеснение кода в ядре (наличие одновременно обоих этих условий), то spinlock_t вообще не компилируются (на их месте остаются пустые места) за счет 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);
```

просто обречен на дедлок — процессор будет активно выполнять этот фрагмент до бесконечности (т. е. происходит деградация системы — число доступных в системе процессоров уменьшается)...

Такой рекурсивный захват спин-блокировки может неявно происходить в обработчике прерываний, поэтому перед захватом этой блокировки нужно запретить прерывания на локальном процессоре. Это общий случай, и 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 еще такие интерфейсы, как:

- попытка захвата без блокирования — если блокировка уже захвачена, функция возвратит ненулевое значение:

```
int spin_try_lock(spinlock_t *sl);
```

- возвращает ненулевое значение, если блокировка в текущий момент захвачена:

```
int spin_is_locked(spinlock_t *sl);
```

Блокировки чтения/записи

Особым, но весьма часто встречающимся случаем синхронизации является ситуация «читателей» и «писателей». «Читатели» только читают состояние некоторого ресурса и поэтому могут осуществлять к нему одновременный параллельный доступ. «Писатели» же 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 также должно быть заблокировано. Для реализации блокировок чтения/записи в ядре Linux существуют отдельные версии, применяющиеся к семафорам и спин-блокировкам. Мьютексы реального времени не имеют реализации для случая «читателей» и «писателей».

Примечание

Обратим здесь внимание на то, что в точности той же функциональности мы могли бы достичь, используя классические примитивы синхронизации (мьютекс или спин-лок), просто захватывая критический участок, независимо от типа предстоящей операции. Блокировки чтения/записи введены из соображений эффективности реализации для очень типового случая применения.

В случае семафоров для реализации блокировок чтения/записи вместо структуры `struct semaphore` вводится `struct rw_semaphore`, а набор интерфейсных функций захвата/освобождения (простые `down()` и `up()`) расширяется до:

- `down_read(&rwsem);` — попытка захватить семафор для чтения;
- `up_read(&rwsem);` — освобождение семафора для чтения;
- `down_write(&rwsem);` — попытка захватить семафор для записи;
- `up_write(&rwsem);` — освобождение семафора для записи.

Семантика этих операций следующая:

- если семафор еще не захвачен, то любой захват (`down_read()`, `down_write()`) будет успешным (без блокирования);
- если семафор уже захвачен 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 уже захвачен для *записи*, то любая последующая попытка захвата семафора (независимо `down_read()` это или `down_write()`) закончится блокированием.

Статически определенный семафор чтения/записи создается 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 (несчетными).

Пример того, как могут быть использованы семафоры чтения/записи при работе (обновлении и считывании) циклических списков Linux (о которых мы говорили ранее):

```
struct data {
   int value;
   struct list_head list;
};
static struct list_head list;
static struct rw_semaphore rw_sem;
int add_value(int value) {
   struct data *item;
   item = kmalloc(sizeof(*item), GFP_ATOMIC);
   if (!item) goto out;
   item->value = value;
   down_write(&rw_sem);              /* захватить для записи */
```

```
   list_add(&(item->list), &list);
   up_write(&rw_sem);                /* освободить по записи */
   return 0;
out:
   return -ENOMEM;
}
int is_value(int value) {
   int result = 0;
   struct data *item;
   struct list_head *iter;
   down_read(&rw_sem);               /* захватить для чтения */
   list_for_each(iter, &list) {
      item = list_entry(iter, struct data, list);
      if(item->value == value) {
         result = 1; goto out;
      }
   }
out:
   up_read(&rw_sem);                 /* освободить по чтению */
   return result;
}
void init_list(void) {
   init_rwsem(&rw_sem);
   INIT_LIST_HEAD(&list);
}
```

Точно так же, как это сделано для семафоров, вводится и блокировка чтения/записи для спин-блокировки:

```
typedef struct {
   raw_rwlock_t raw_lock;
...
} rwlock_t;
```

С набором операций (макросы):

```
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` тоже.

> ***Примечание***
>
> Блокировку, захваченную для чтения, уже нельзя далее «повышать» до блокировки, захваченной для записи. Последовательность операторов:

```
read_lock(&rwlock);
write_lock(&rwlock);
```

гарантирует нам дедлок, т. к. при захвате блокировки на запись будет выполняться периодическая проверка того, все ли потоки, которые захватили блокировку для чтения, ее освободили. То же касается и текущего потока, который не сделает этого никогда... Но несколько потоков *чтения* безопасно могут захватывать одну и ту же блокировку чтения/записи, поэтому один поток также может безопасно рекурсивно захватывать одну и ту же блокировку для чтения несколько раз — например, в обработчике прерываний без запрета прерываний.

На момент создания механизма блокировок чтения/записи их использованию прогнозировали значительное повышение производительности, и они вызывали заметный энтузиазм разработчиков. Но последующая практика показала, что этому механизму присуща скрытая опасность того, что при *высокой и равномерной* плотности запросов чтения запрос на модификацию (запись) структуры записи может отсрочиваться на неограниченно большие интервалы времени. Об этой особенности нужно помнить, взвешивая применение этого механизма в своем коде. Частично смягчить это ограничение пытается следующий подвид блокировок — сериальные блокировки.

Сериальные (последовательные) блокировки

Сериальные блокировки — это пример одного только из нескольких механизмов синхронизации, которые и блокировками по существу (в полной мере) не являются... По сути, они представляют собой, как уже отмечено, подвид блокировок чтения/записи. Такой механизм добавлен для получения эффективных по времени реализаций. Описаны они в <linux/seqlock.h>, и для их представления вводится тип `seqlock_t`:

```
typedef struct {
   unsigned sequence;
   spinlock_t lock;
} seqlock_t;
```

Такой элемент блокировки создается и инициализируется статически:

```
DEFINE_SEQLOCK(name);
```

Или вот эквивалентная динамическая инициализация:

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

Доступ на чтение работает, получая целочисленное значение (без знака) последовательности (ключ) на входе в защищаемую критическую секцию. На выходе из этой секции полученное значение должно сравниваться с текущим таким значением (на момент завершения). Если выявляется несоответствие, это означает, что секция (за это время!) обрабатывалась операциями записи, и проделанное чтение должно быть повторено. В результате код «читателя» имеет вид, подобный следующему:

```
DEFINE_SEQLOCK(lock);
unsigned int seq;
```

```
do {
   seq = read_seqbegin(&lock);
   /* ... */
} while read_seqretry(&lock, seq);
```

Блокировка по записи реализована через спин-блокировку. «Писатели» должны получить эксклюзивную спин-блокировку, чтобы войти в критическую секцию, защищаемую последовательной блокировкой. Чтобы это сделать, код «писателя» делает вызов функции:

```
static inline void write_seqlock(seqlock_t *sl) {
   spin_lock(&sl->lock);
   ++sl->sequence;
   smp_wmb();
}
```

Снятие блокировки записи выполняет другая функция:

```
static inline void write_sequnlock(seqlock_t *sl) {
   smp_wmb();
   sl->sequence++;
   spin_unlock(&sl->lock);
}
```

Здесь любопытно то, что «писатель» делает инкремент ключа последовательности дважды: после захвата спин-блокировки и перед ее освобождением. В этой связи интересно посмотреть реализацию того, как «читатель» получает свое начальное значение ключа вызовом `read_seqbegin()`:

```
static __always_inline unsigned read_seqbegin(const seqlock_t *sl) {
   unsigned ret;
repeat:
   ret = sl->sequence;
   smp_rmb();
   if(unlikely(ret & 1)) {
      cpu_relax();
      goto repeat;
   }
   return ret;
}
```

Отсюда понятно, что если «читатель» запросит код последовательного доступа в то время, когда в критической секции находится «писатель» («писатель» сделал начальный инкремент, но не сделал завершающего), то запросивший «читатель» будет выполнять пустые циклы ожидания до тех пор, пока «писатель» не покинет секцию.

Существует также вариант `write_tryseqlock()`, которая возвращает ненулевое значение, если «писатель» не смог получить блокировку.

Если механизмы последовательной блокировки должны быть задействованы в обработчике прерываний, то следует использовать специальные (безопасные) версии API всех показанных ранее вызовов (макросов):

```
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 флагов IRQ.

Мьютексы реального времени

Кроме обычных мьютексов (как бинарного подвида семафоров), в ядре создан новый интерфейс для мьютексов реального времени (`rt_mutex`). Структура мьютекса реального времени (<linux/rtmutex.h>), если исключить из рассмотрения ее отладочную часть, имеет вид:

```
struct rt_mutex {                    //  The rt_mutex structure
   raw_spinlock_t         wait_lock; // spinlock to protect the structure
   struct rb_root_cached  waiters;   // head to enqueue waiters in priority order
   struct task_struct     *owner;    // the mutex owner
};
```

Примечательным является явное присутствие поля `owner` (владелец), что характерно для любых вообще мьютексов POSIX (и отличает их от семафоров), — это уже обсуждалось ранее. Там же (в <linux/rtmutex.h>) 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 проверяется так:

```
$ grep RT_MUTEX /boot/config-`uname -r`
CONFIG_RT_MUTEXES=y
# CONFIG_DEBUG_RT_MUTEXES is not set
```

В отличие от регулярных мьютексов, мютексы реального времени обеспечивают *наследование приоритетов* (priority inheritance, PI), что является одним из нескольких немногих известных способов, препятствующих возникновению *инверсии при-*

оритетов (priority inversion)[7]. Логика наследования приоритетов проста: 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 `struct rb_root_cached` (в описании структуры `rt_mutex`);
- при необходимости этот список ожидающих процессов переупорядочивается в порядке *приоритетов* ожидающих процессов;
- приоритет владельца `owner` мьютекса (текущего владеющего, захватившего поток) B повышается до приоритета ожидающего процесса A (или максимального приоритета из ожидающих в очереди процессов);
- после освобождения мютекса приоритет владельца B восстанавливается до его начального значения;
- это и обеспечивает возможность избежать потенциальной инверсии приоритетов.

Примечание

Эти действия затрагивают глубины управления потоками, и 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 потока необходимо теперь иметь несколько полей приоритета:

```
struct task_struct {
...
   int prio, static_prio, normal_prio;
...
}
```

из которых поле `prio` является динамическим приоритетом, согласно которому и происходит диспетчеризация процессов в системе, а поле приоритета `normal_prio` остается неизменным, и по его значению происходит восстановление приоритета после освобождения мьютекса реального времени.

Множественное блокирование

В системах с большим количеством блокировок (а ядро — именно такая система) необходимость проведения более чем одной блокировки за раз не является для кода

[7] Инверсия приоритетов — очень значимая и очень тяжелая проблема в инженерных проектах *реального времени*. И ситуация инверсии приоритетов весьма тяжело моделируется, возникает с малой частотой и тяжело диагностируется.

необычной. 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);
DEFINE_SPINLOCK(lock2);
...
spin_lock (&lock1); /* 1-й фрагмент кода */
spin_lock (&lock2);
...
spin_lock (&lock2); /* где-то в совсем другом месте кода... обратный порядок */
spin_lock (&lock1);
```

Такой образец кода в конечном итоге когда-то обречен на бесконечное блокирование (deadlock).

Если есть необходимость захвата нескольких блокировок, то единственная возможность ее обеспечить — это предусмотреть: а) один тот же порядок захвата и б) освобождения блокировок; в) порядок освобождения, обратный порядку захвата, и г) чтобы это так же выглядело в каждом из фрагментов кода. В этом смысле предыдущий пример может быть переписан так:

```
spin_lock (&lock1); /* так должно быть везде, где использованы lock1 и lock2 */
spin_lock (&lock2);
   /* ... здесь выполняется действие */
spin_unlock (&lock2);
spin_unlock (&lock1);
```

На практике обеспечить такую синхронность работы с блокировками в различных фрагментах кода крайне проблематично! (потому что это может касаться фрагментов кода разных авторов).

Уровень блокирования

Нужно также обратить внимание на такой отдельный вопрос, как уровень блокирования. Очень часто, особенно когда это касается защиты критического участка кода, а не ограждения структуры данных, блокирование для синхронизации можно осуществлять на разных уровнях *вложенности*. Простейший пример этого покажем на фрагменте вида:

```
static DECLARE_MUTEX(sema);
down(&sema);
for(int i = 0; i < n; i++) {
   // здесь делается нечто монопольное за время T
}
up(&sema);
```

Этот же фрагмент может быть выполнен по-другому, что *функционально* эквивалентно:

```
static DECLARE_MUTEX(sema);
for(int i = 0; i < n; i++) {
   down(&sema);
   // здесь делается нечто монопольное за время T
   up(&sema);
}
```

Но во втором случае потенциальное блокирование любых других потоков, потребовавших семафор, будет представляться как `n` раздельных интервалов длительностью `T`, а в первом — как один сплошной интервал протяженностью `n*T`. Однако часто такой интервал ожидания — это время латентности системы. Реальные программные системы представляют собой сложные образования, где глубина вложенных компонентов во много раз больше единицы, как в показанном условном примере. Общее правило состоит в том, что блокирование (синхронизация) должно осуществляться *на как можно более глубоком уровне*, — даже если это потребует многократного увеличения числа обращений к примитиву синхронизации.

> ***Примечание***
>
> На протяжении многих лет (фактически от рождения в 1991 г.) в ядре Linux существовала так называемая *глобальная блокировка ядра*: если кто-либо в системе захватывал такую блокировку, то все последующие запросы глобальной блокировки, откуда бы они ни исходили, блокировались. Это страшная вещь! И радикально избавиться от глобальной блокировки (до этого только постоянно сужалась область ее применимости) с большим трудом удалось только в ядре 3.*x* к 2011 году.

А теперь пример, который не столько ценен сам по себе, но живые эксперименты с которым позволяют очень тонко прочувствовать, как происходит синхронизация потоков и как эта синхронизация может быть сделана на самых различных уровнях (кроме того, это пример использования семафоров, обещанный ранее):

slock.c

```
#include <linux/module.h>
#include <linux/delay.h>
#include <linux/kthread.h>
#include <linux/semaphore.h>

#include "../prefix.c"

static int num = 2;                // num - число рабочих потоков
module_param(num, int, 0);
static int rep = 100;              // rep - число повторений (объем работы)
module_param(rep, int, 0);
static int sync = -1;              // sync - уровень, на котором синхронизация
module_param(sync, int, 0);
static int max_level = 2;          // max_level - уровень глубины вложенности
module_param(max_level, int, 0);
```

```
DEFINE_SEMAPHORE(sema);

static long locked = 0;

static long loop_func(int lvl){
   long n = 0;
   if(lvl == sync){ down(&sema); locked++; }
   if(0 == lvl) {
      const int tick = 1;
      msleep(tick);                // выполняемая работа потока
      n = 1;
   }
   else {
      int i;
      for(i = 0; i < rep; i++) {
         n += loop_func(lvl - 1);
      }
   }
   if(lvl == sync) up(&sema);
   return n;
}

struct param {                      // параметр потока
   int    num;
   struct completion finished;
};

#define IDENT "mlock_thread_%d"
static int thread_func(void* data) {
   long n = 0;
   struct param *parent = (struct param*)data;
   int num = parent->num - 1;       // порядковый номер потока (локальный!)
   struct task_struct *t1 = NULL;
   struct param parm;
   printk("! %s is running\n", st(num));
   if(num > 0) {
      init_completion(&parm.finished);
      parm.num = num;
      t1 = kthread_run(thread_func, (void*)&parm, IDENT, num);
   }
   n = loop_func(max_level);        // рекурсивный вызов вложенных циклов
   if(t1 != NULL)
      wait_for_completion(&parm.finished);
   complete(&parent->finished);
   printk("! %s do %ld units\n", st(num), n);
   return 0;
}
```

```
static int test_mlock(void) {
   struct task_struct *t1;
   struct param parm;
   unsigned j1 = jiffies, j2;
   if(sync > max_level) sync = -1; // без синхронизации
   printk("! repeat %d times in %d levels; synch. in level %d\n",
           rep, max_level, sync);
   init_completion(&parm.finished);
   parm.num = num;
   t1 = kthread_run(thread_func, (void*)&parm, IDENT, num);
   wait_for_completion(&parm.finished);
   printk("! %s is finished\n", st(num));
   j2 = jiffies - j1;
   printk("!! working time was %d.%1d seconds, locked %ld times\n",
           j2 / HZ, (j2 * 10) / HZ % 10, locked);
   return -1;
}

module_init(test_mlock);
MODULE_LICENSE("GPL");
```

Модуль достаточно сложный (не по коду, а по логике), поэтому есть смысл привести некоторые краткие комментарии:

- включаемый файл определяет функцию `st()` для форматирования диагностики о потоке (с меткой времени `jiffies`) — эту функцию мы уже видели ранее в обсуждении создания потоков;
- потоки (числом `num` — параметр запуска модуля) запускают друг друга последовательно и завершаются в обратном порядке: каждый поток ожидает завершения им порожденного (по типу того, как это делается в рекурсии, но только в динамике... здесь нет как таковой рекурсии);
- работа потока состоит в циклическом (параметр: `rep` — раз) выполнении рекурсивной функции `loop_func()`:

  ```
  for(j1; ...)
  ```

- рекурсия, вообще-то говоря, крайне рискованная вещь в модулях ядра — из-за ограниченности и фиксированного размера стека ядра, но в нашем случае: а) это иллюстративная задача (и она попутно показывает возможность рекурсии в коде ядра); б) функция сознательно имеет минимальное число локальных (стековых) переменных и в) причина, которая «весит» больше всех остальных, вместе взятых, — рекурсия позволяет создать структуру вложенных циклов *переменной* и произвольно большой глубины вложенности (параметр `max_level` модуля), вызов `loop_func(N)` эквивалентен:

  ```
  for(j2; ...)
     for(j3; ...)
     ...
        for(jN; ...)
  ```

- варьируя параметр модуля sync, можно заказывать, на какой глубине вложенных циклов потоки станут пытаться синхронизироваться захватом семафора sema: sync=0 — на самом глубоком уровне имитации работы потока, sync=1 — уровнем выше, ..., sync=max_level — на максимально возможном верхнем уровне охватывающего цикла, sync<0 или sync>max_level — вообще не синхронизироваться, не пытаться получить доступ к семафору sema;
- модуль выполнен в уже полюбившейся нам манере исполнения — как пользовательская задача, ничего не устанавливающая в ядре, но выполняющаяся в режиме защиты супервизора.

Ну а дальше остается только многократно экспериментировать... Вот *экспоненциальная* степень роста полного объема работы в зависимости от глубины вложенности (94,6 раза при росте max_level с 2 до 4 — без синхронизации, с синхронизацией все будет значительно хуже):

```
$ time sudo insmod slock.ko rep=10 num=2 max_level=2 sync=-1
insmod: ERROR: could not insert module slock.ko: Operation not permitted
real    0m1,272s
user    0m0,008s
sys     0m0,004s
$ dmesg | grep ! | tail -n7
[17675.562501] ! repeat 10 times in 2 levels; synch. in level -1
[17675.562584] ! 4299311045 : kthread [20837:1] is running
[17675.562629] ! 4299311045 : kthread [20838:0] is running
[17676.766882] ! 4299311347 : kthread [20838:0] do 100 units
[17676.766885] ! 4299311347 : kthread [20837:1] do 100 units
[17676.766926] ! 4299311347 : kthread [20836:2] is finished
[17676.766928] !! working time was 1.2 seconds, locked 0 times
$ time sudo insmod slock.ko rep=10 num=2 max_level=4 sync=-1
insmod: ERROR: could not insert module slock.ko: Operation not permitted
real    2m0,379s
user    0m0,008s
sys     0m0,005s
$ dmesg | grep ! | tail -n7
[17830.610924] ! repeat 10 times in 4 levels; synch. in level -1
[17830.611044] ! 4299349808 : kthread [20915:1] is running
[17830.611099] ! 4299349808 : kthread [20916:0] is running
[17950.836423] ! 4299379865 : kthread [20916:0] do 10000 units
[17950.912423] ! 4299379884 : kthread [20915:1] do 10000 units
[17950.912433] ! 4299379884 : kthread [20914:2] is finished
[17950.912435] !! working time was 120.3 seconds, locked 0 times
```

Такое заметное торможение синхронизированной задачи, возможно, объясняется еще и тем, что в *отсутствие* блокирования критических областей примитивами синхронизации (семафором — в этом случае) в выполнение работы (какой бы бессмысленной она ни была) параллельно включаются другие ядра многопроцессорной системы. При включении синхронизации подключение к выполнению других

процессоров пресекается (проверить это предположение у меня не было возможности, потому что одноядерные процессоры остались в доступе только в политехнических музеях, а в Linux консольная команда `taskset` управляет аффинити-маской запускаемого пользовательского процесса, но никак не модуля ядра).

А вот различия в *числе* выполненных синхронизаций — в зависимости от того, на какой глубине вложения они применяются:

```
$ time sudo insmod slock.ko rep=7 num=5 max_level=3 sync=-1
insmod: ERROR: could not insert module slock.ko: Operation not permitted
real    0m4,233s
user    0m0,006s
sys     0m0,007s
$ dmesg | grep '!!' | tail -n1
[18805.948779] !! working time was 4.1 seconds, locked 0 times
$ time sudo insmod slock.ko rep=7 num=5 max_level=3 sync=0
insmod: ERROR: could not insert module slock.ko: Operation not permitted
real    0m20,668s
user    0m0,006s
sys     0m0,007s
$ dmesg | grep '!!' | tail -n1
[18907.343872] !! working time was 20.6 seconds, locked 1715 times
$ time sudo insmod slock.ko rep=7 num=5 max_level=3 sync=1
insmod: ERROR: could not insert module slock.ko: Operation not permitted
real    0m20,676s
user    0m0,008s
sys     0m0,004s
$ dmesg | grep '!!' | tail -n1
[18939.039549] !! working time was 20.6 seconds, locked 245 times
$ time sudo insmod slock.ko rep=7 num=5 max_level=3 sync=2
insmod: ERROR: could not insert module slock.ko: Operation not permitted
real    0m20,712s
user    0m0,009s
sys     0m0,003s
$ dmesg | grep '!!' | tail -n1
[18982.239179] !! working time was 20.6 seconds, locked 35 times
$ time sudo insmod slock.ko rep=7 num=5 max_level=3 sync=3
insmod: ERROR: could not insert module slock.ko: Operation not permitted
real    0m20,764s
user    0m0,009s
sys     0m0,003s
$ dmesg | grep '!!' | tail -n1
[19015.890922] !! working time was 20.6 seconds, locked 5 times
$ time sudo insmod slock.ko rep=7 num=5 max_level=3 sync=4
insmod: ERROR: could not insert module slock.ko: Operation not permitted
real    0m4,195s
user    0m0,008s
sys     0m0,005s
```

```
$ dmesg | grep '!!' | tail -n1
[19030.418666] !! working time was 4.1 seconds, locked 0 times
```

Очень показательно в выводе число обращений (`locked`) к семафору: на одном и том же периоде выполнения число обращений изменяется на три *порядка*, во столько же раз «гуляет» продолжительность единичного акта захвата примитива синхронизации, — то, с чего началось обсуждение этого раздела.

Приведенный пример не показал зависимости общего итогового времени выполнения от глубины уровня синхронизации, но здесь задействованы такие продолжительные циклы работы в критической секции, что они нивелируют издержки синхронизации, маскируют разницу. Но не хотелось еще больше усложнять пример... В реальных задачах тем не менее соблюдается общее правило: чем выше (по уровням вложенности) выбран уровень синхронизации — тем больше затраты времени на выполнение.

Предписание порядка выполнения

Механизмы, предписывающие порядок выполнения кода, к синхронизирующим механизмам относятся весьма очень условно, — они не являются непосредственно синхронизирующими механизмами, но, как уже отмечалось ранее, рассматриваются всегда вместе с ними (по принципу «ну, надо же их где-то рассматривать»). А рассматривают их совместно с синхронизациями потому, что их роднит *единственное* сходство: и те и другие могут влиять на порядок выполнения операторов кода и изменять его в «плавном» последовательном выполнении.

Аннотация ветвлений

Одним из таких механизмов являются определенные в <linux/compiler.h> макросы `likely()` и `unlikely()`, которые иногда называют *аннотацией ветвлений*, например:

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

Или:

```
if(likely(...)) {
   /* обычное прохождение вычислений */
}
else {
   /* что-то нетрадиционное */
};
```

Те, кто знают в минимальном объеме специфику выполнения процессорной инструкции `jmp`, вспомнят, что при ее выполнении происходит разгрузка (и перезагрузка) конвейера уже частично декодированных последующих команд. Кроме того, в игру способен включиться и кеш памяти, который может потребовать перезагрузки в дальнюю `jmp`-область (а это разница в скорости обычно в 2–4 **раза** и более). А если это так и если во всяком ветвлении (условном переходе) одна из ветвей

обязательно должна выполнять jmp, то можно дать указание компилятору компилировать код так, чтобы веткой с дальним jmp оказалась ветка с наименьшей вероятностью (частотой) выполнения. Таким образом, аннотации ветвлений нужны для повышения производительности выполнения кода (иногда заметно ощутимого), — никаким другим образом на выполнение они не влияют. Кроме того, такие предписания: а) делают код более читабельным и б) недопустимы (не определены) в пространстве пользовательского кода (*только в ядре*).

> ***Примечание***
>
> Подобные оптимизации становятся актуальными с появлением в процессорах конвейерных вычислений с предсказыванием. На других платформах, отличных от Intel *x*86, они могут быть на сегодня и не столь ощутимыми (это нужно уточнять детальным просмотром архитектуры перед началом разработки).

Барьеры

Другим примером предписаний порядка выполнения являются барьеры в памяти, препятствующие в процессе оптимизации переносу операций чтения и записи через объявленный барьер. Например, при записи фрагмента кода:

```
a = 1;
b = 2;
```

Порядок выполнения операций, вообще-то говоря, непредсказуем, причем последовательность (во времени) выполнения операций могут изменить: а) компилятор — из соображений оптимизации и б) процессор (периода выполнения) — из соображений аппаратной оптимизации работы с шиной памяти. Тогда это совершенно нормально — более того, даже запись операторов:

```
a = 1;
b = a + 1;
```

будет гарантировать отсутствие перестановок в процессе оптимизации, т. к. 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 определенные барьеры. Операции (макросы) с барьерами объявлены в </asm-generic/system.h>, и на сегодня все они (rmb(), wmb(), mb(), ...) определены одинаково:

```
#define   mb()  asm volatile (""): : :"memory")
```

И все они препятствуют выполнению операций с памятью после такого вызова до завершения всех операций, записанных до вызова.

Еще один макрос:

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

объявлен в <linux/compiler.h> — он запрещает компилятору при оптимизации переставлять между собой операторы до этого вызова и после него.

Обработка прерываний

Трудное — это то, что может быть сделано немедленно;
невозможное — то, что потребует немного больше времени.
Сантаяна

Мы закончили рассмотрение механизмов параллелизмов для ситуаций, когда это действительно параллельно выполняющиеся фрагменты кода (в случае SMP и наличия нескольких процессоров) или когда это квазипараллельность, а различные ветви асинхронно вытесняют друг друга, занимая единый процессор. Глядя на сложности, порождаемые параллельными вычислениями, можно было бы попытаться и вовсе отказаться от параллельных механизмов в угоду простоте и детерминированности последовательного вычислительного процесса. И так и стараются поступать часто в малых и встраиваемых архитектурах. Можно было бы... если бы не один вид естественного *асинхронного параллелизма*, который возникает вне наших желаний 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 последовательный, как принято в сугубо теоретическом рассмотрении, — параллелизм присутствует всегда!

Обмен с периферийным оборудованием, по существу, может происходить *только двумя* способами: циклическим программным опросом (пулингом) и по прерываниям от периферийного оборудования.

> ***Примечание***
>
> Есть одна область практических применений средств компьютерной индустрии, которая развивается совершенно автономно и в которой попытались уйти от асинхронного обслуживания аппаратных прерываний, относя именно к наличию этих механизмов риски отказов, снижения надежности 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 надежности (это совершенно отдельный класс компьютерного оборудования). Такие PLC строятся на абсолютно тех же процессорах общего применения, но обмениваются с многочисленной периферией не по прерываниям, а только методами непрерывного циклического программного опроса (пулинга), часто с периодом опроса миллисекундного диапазона или даже ниже.
>
> Невзирая на некоторую обособленность этой ветви развития, она занимает (в финансовых объемах) весьма существенную часть компьютерной индустрии, где преуспели такие мировые бренды, как: Modicon (ныне Schneider Electric), Siemens, Allen-Bradley и некоторые другие. Примечательно, что целый ряд известных моделей PLC работает в том числе и под операционной системой Linux, но работа с данными в них основывается на совершенно иных принципах, что, собственно, и превращает их в PLC. Вся эта отрасль стоит особняком, и к ее особенностям мы не будем больше обращаться.

Может присутствовать и комбинированное использование указанных способов — остроумное решение в этом направлении являет собой сетевая подсистема Linux (так называемый NAPI, New API): в периоды «затишья» сетевой активности подсистема ожидает прихода первых сетевых пакетов по прерыванию. Но после полу-

чения первого пакета серии, ожидая следующий всплеск активности, сетевая подсистема переходит в режим пулинга, циклически опрашивая сетевой адаптер. Как только сетевая активность затихает, пулинг прекращается и подсистема возвращается в режим ожидания следующего прерывания.

Общая модель обработки прерывания

Схема обработки аппаратных прерываний — это принципиально архитектурно-зависимое действие, связанное с непосредственным взаимодействием с контроллером прерываний. Но в основных своих чертах схема остается неизменной, независимо от архитектуры. Вот как она выглядела, к примеру, в системе MS-DOS для процессоров *x*86 и «старого» контроллера прерываний (чип 8259) — на уровне ассемблера это нечто подобное следующей последовательности действий:

1. После возникновения аппаратного прерывания управление асинхронно получает функция (ваша функция!), адрес которой записан в векторе (вентиле) прерывания.
2. Обработку прерывания функция обработчика выполняет при *запрещенных* следующих прерываниях.
3. После завершения обработки прерывания функция-обработчик восстанавливает контроллер прерываний (чип 8259), посылая ему сигнал о завершении прерывания. Это осуществляется отправкой команды `EOI` (End Of Interrupt, конец прерывания — код `20h`) в командный регистр (порт) микросхемы 8259 (этот однобайтовый регистр адресуется через порт ввода/вывода `20h`).
4. Функция-обработчик завершается, возвращая управление командой `iret` (не `ret`, как все прочие привычные нам функции, вызываемые синхронно!).

Показанная схема слишком неэффективна, ненадежна (в отношении пропущенных прерываний на время аппаратного запрета прерываний) и просто не может быть распространена на многозадачное окружение и на SMP-архитектуры, использующие более современные чипы APIC контроллера прерываний (принципиально отличающиеся работой). В Linux обработка прерывания разделяется на *две последовательные* фазы и трансформируется в следующую схему:

1. Для линии IRQ регистрируется функция обработчика «верхней половины» (это та же ISR-функция по смыслу — «верхняя половина» обработчика), которая выполняется *при запрещенных прерываниях локального* процессора. Именно этой функции передается управление при возникновении аппаратного прерывания.
2. Синтаксически функция-обработчик должна иметь строго описанный функциональный тип `irq_handler_t` и возвращать управление *ядру системы* традиционным `return` с возвращаемым значением `IRQ_NONE` или `IRQ_HANDLED`.
3. При возникновении аппаратного прерывания по линии IRQ функция-обработчик получит управление. Эта функция выполняется *в контексте прерывания* — это одно из самых важных ограничений, накладываемых Linux, мы не раз будем к нему возвращаться. Перед своим завершением функция-обработчик *регистри-*

рует для последующего выполнения функцию *нижней половины* обработчика, которая и завершит *позже* начатую работу по обработке этого прерывания.

4. В этой точке (после `return` из обработчика *верхней половины*) ядро завершает всё взаимодействие с аппаратурой контроллера прерываний, *разрешает* последующие прерывания на локальном процессоре, восстанавливает контроллер командой завершения обработки прерывания (посылает в порт команду `EOI`) и возвращает управление из прерывания (из ядра!) уже именно командой `iret`. После этого будет восстановлен *контекст прерванного* процесса (потока) — действительный указатель `current`.
5. А вот запланированная ранее к выполнению функция *нижней половины* будет вызвана ядром в некоторый момент позже (часто это может быть и непосредственно после завершения `return` из верхней половины, но это непредсказуемо) — тогда, когда удобнее будет ядру системы. Принципиально важное отличие функции нижней половины состоит в том, что она выполняется уже *при разрешенных 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* (рис. 5.4).

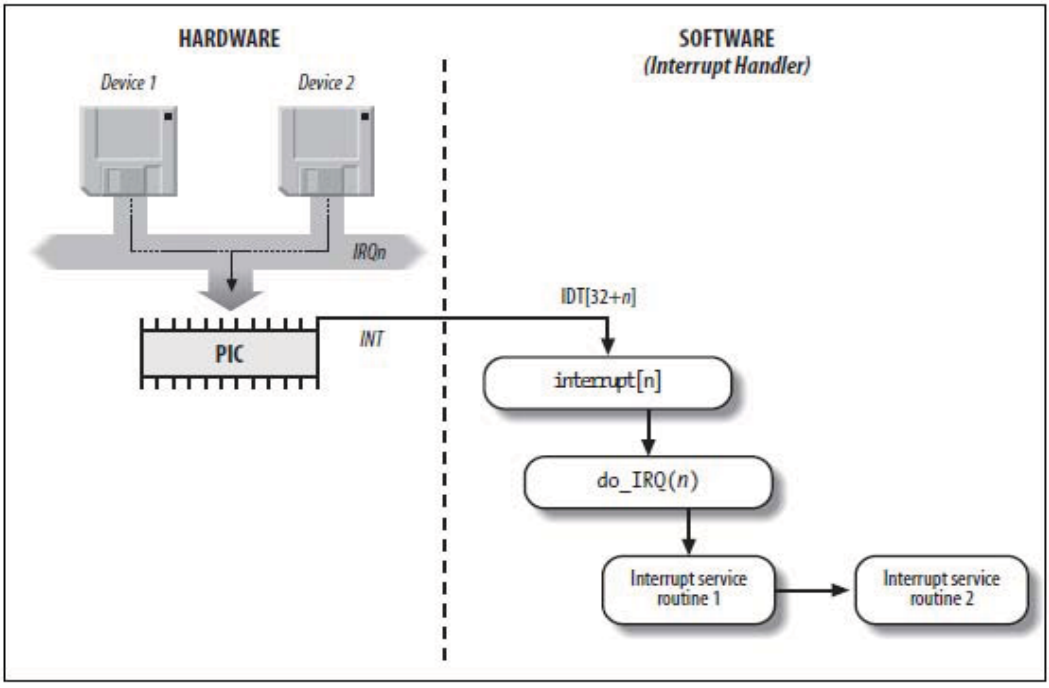


Рис. 5.4. Траектория прохождения прерывания в Linux

Такого укрупненного изложения схемы обработки аппаратных прерываний нам уже достаточно для всего дальнейшего экскурса в детали прерываний. Но прежде нужно очень тщательно сформулировать ту самую отличительную сторону прерываний, делающую работу с ними сложной, — что же такое есть *контекст выполнения* в Linux? В Linux все синхронные ветви последовательно выполняющегося кода могут принадлежать к категориям:

1. Пользовательский *процесс* (приложение).
2. Отдельный *поток* (POSIX `pthread_t`) в многопоточном пользовательском процессе.

   Пояснение

   Как мы знаем, в Linux ядро диспетчирует именно потоки, *задачи*, поэтому в отношении предыдущего пункта было бы правильнее сказать: главный *поток* пользовательского приложения, представленный функцией `main()`.

3. Linux — операционная система с *вытеснением в ядре*. Поэтому в ядре существуют автономные *потоки ядра* — они и представляют третью категорию.

Невзирая на различия в предназначении, выполняемые единицы всех этих категорий представляются в ядре (<linux/sched.h>) *одинаковой* структурой *задачи*: `struct task_struct`. Более того, для диспетчирования ядром *все* они включены в *единую* систему приоритетных циклических очередей. Структура задачи `struct task_struct` — это очень крупная структура, содержащая, кроме прочего, полный набор текущих параметров в состоянии, например, когда задача прерывается (совершенно не обязательно асинхронно и аппаратным прерыванием — просто у задачи может истечь выделенный ей квант времени, и она будет прервана диспетчером ядра). То есть `struct task_struct` содержит полный *необходимый и достаточный* набор всех параметров, чтобы приостановленную задачу активировать в сколь угодно отдаленном будущем. Говорят при этом, что все указанные категории «активностей» выполняются *в контексте задачи*.

При возникновении аппаратного прерывания текущая задача прерывается, ее контекст сохраняется и управление передается функции ISR (верхней половины). Но для выполняющегося ISR-кода не создается `struct task_struct`. И вот такое выполнение называют уже *контекстом прерывания*. И стоит функции ISR вызвать любую другую функцию API ядра, задающую блокирование вызывающего кода (начиная с такой безобидной паузы, как `msleep()`), и управление будет передано коду какой-то из других задач (`struct task_struct`), но оно *никогда не возвратится обратно*, потому что у ISR нет контекста задачи, в который можно возвратиться. И произойдет подобное не только при прямом вызове таких блокирующих функций API, но и при вызове в сколь угодно длинной последовательной цепочке вызовов (некоторые из них могут быть написаны другим автором). А последствием такого неосмотрительного действия программиста станет *общий крах ядра системы*. В этом состоит, пожалуй, главная сложность обработки прерываний в модуле ядра — непрерывное отслеживание того, в каком контексте отрабатывается код, который вы создаете.

Наблюдение прерываний в */proc*

Но прежде, чем дальше углубляться в организацию обработки прерывания, коротко остановимся на том, как мы можем наблюдать и контролировать то, что происходит с прерываниями. Всякий раз, когда аппаратное прерывание обрабатывается процессором, внутренний счетчик прерываний увеличивается, предоставляя возможность контроля за подсистемой прерываний. Счетчики отображаются в /proc/interrupts (последняя колонка в приведенной далее «раскладке» прерываний — это и есть имя обработчика, зарегистрированное, как это будет показано далее, параметром `name` при регистрации в вызове `request_irq()`). Итак, рассмотрим «раскладку» прерываний в архитектуре *x*86 при использовании старого, стандартного программируемого контроллера прерываний PC 8259 (если точнее, то здесь описана схема с двумя каскадно-объединенными по линии IRQ2 контроллерами 8259, которая была классикой более 20 лет, — чип контроллера прерываний 8259 создавался еще под 8-битный процессор 8080). Такую картину можно наблюдать только на компьютере с одним (и очень старым) процессором:

```
$ cat /proc/interrupts
           CPU0
  0:   33675789          XT-PIC  timer
  1:      41076          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  5:         18          XT-PIC  uhci_hcd:usb1, CS46XX
  6:          3          XT-PIC  floppy
  7:          0          XT-PIC  parport0
  8:          1          XT-PIC  rtc
  9:          0          XT-PIC  acpi
 11:    2153158          XT-PIC  ide2, eth0, mga@pci:0000:01:00.0
 12:     347114          XT-PIC  i8042
 14:         38          XT-PIC  ide0
```

Те линии IRQ, для которых не установлены текущие обработчики прерываний, не отображаются в /proc/interrupts. Здесь уже хорошо видно разделение линии IRQ 11 между тремя различными PCI-устройствами.

То же самое, но на существенно более новом компьютере с несколькими процессорами (ядрами), когда источником прерываний является усовершенствованный контроллер прерываний IO-APIC (отслеживаются прерывания как *по фронту*: `IO-APIC-edge`, так и *по уровню*: `IO-APIC-level`/`IO-APIC-fasteoi`):

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:         49          0          0          0   IO-APIC-edge      timer
  1:          8          0          0          0   IO-APIC-edge      i8042
  4:          2          0          0          0   IO-APIC-edge
  7:          0          0          0          0   IO-APIC-edge      parport0
  8:          1          0          0          0   IO-APIC-edge      rtc0
  9:          0          0          0          0   IO-APIC-fasteoi   acpi
```

```
 12:        144          0          0          0   IO-APIC-edge      i8042
 14:          0          0          0          0   IO-APIC-edge      ata_piix
 15:          0          0          0          0   IO-APIC-edge      ata_piix
 16:         30          0          0          0   IO-APIC-fasteoi   uhci_hcd:usb5, i915
 18:          0          0          0          0   IO-APIC-fasteoi   uhci_hcd:usb4
 19:       5620       7184          0          0   IO-APIC-fasteoi   ata_piix, uhci_hcd:usb3
 22:        572          0          0          0   IO-APIC-fasteoi   HDA Intel
 23:          0          0          0          0   IO-APIC-fasteoi   ehci_hcd:usb1, uhci_hcd:usb2
 27:         83          0        157          0   PCI-MSI-edge      eth0
NMI:          0          0          0          0   Non-maskable interrupts
LOC:       6646       8926       5769       5409   Local timer interrupts
...
```

Примечание

Обратите внимание на прерывания по линии IRQ 27, когда прерывания от сетевого адаптера `eth0`, чередуясь, обрабатываются на разных процессорах.

Чрезвычайно важным является то, уже отмеченное чуть ранее обстоятельство, что различные линии IRQ могут быть запрограммированы в контроллере IO-APIC на разные механизмы срабатывания: *по фронту*: `IO-APIC-edge` или *по уровню*: `IO-APIC-fasteoi`. Механизм срабатывания вы, как разработчик проекта, можете выбирать по собственному усмотрению при установке обработчика прерывания (ISR), что будет показано немного далее. Это может иметь принципиальное значение: обработка по фронту способна пропускать прерывания, происходящие на одной линии IRQ от двух разных устройств, а обработка по уровню — приводить к ложным срабатываниям при неправильной обработке. Для нескольких устройств на шине PCI, разделяющих одну линию IRQ, чаще всего выбирают обработку прерываний по уровню (смысл и процесс обработки по уровню и по фронту лучше всех показаны в [14]).

А вот как это может выглядеть на ARM-компьютере (т. е. оборудование может быть самым разным, но методы наблюдения остаются адекватными):

```
$ inxi -MC
Machine:   Type: ARM Device System: Xunlong Orange Pi One details: Allwinner sun8i Family rev: N/A
           serial: 02c000815fd5e717
CPU:       Topology: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32 type: MCP
           Speed: 1008 MHz min/max: 480/1008 MHz Core speeds (MHz): 1: 1008 2: 1008 3: 1008 4:
                                                                                        1008
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
 25:          0          0          0          0     GICv2  50 Level     timer@1c20c00
 26:          0          0          0          0     GICv2  29 Level     arch_timer
 27:    2684922    1225055    1778816     470413      GICv2  30 Level      arch_timer
 30:        206          0          0          0     GICv2  82 Level     1c02000.dma-controller
 31:       1452          0          0          0     GICv2 118 Level     1c0c000.lcd-controller
 32:     331414          0          0          0     GICv2  92 Level     sunxi-mmc
 34:          0          0          0          0     GICv2 103 Level     musb-hdrc.4.auto
```

```
 35:          0          0          0          0     GICv2 104 Level     ehci_hcd:usb1
 36:          0          0          0          0     GICv2 105 Level     ohci_hcd:usb2
 37:          2          0          0          0     GICv2 106 Level     ehci_hcd:usb3
 38:         34          0       1402          0     GICv2 107 Level     ohci_hcd:usb4
 42:     197166          0          0          0     GICv2 114 Level     eth0
 45:          6          0          0          0     GICv2  32 Level     ttyS0
 46:        904          0          0          0     GICv2 120 Level     1ee0000.hdmi, dw-hdmi-cec
 47:          0          0          0          0  sun6i-r-intc  40 Level     1f00000.rtc
 51:          0          0          0          0     GICv2  90 Level     1c0e000.video-codec
 52:          0          0          0          0     GICv2 126 Level     sun8i-ce-ns
 53:          0          0          0          0     GICv2 129 Level     gp
 54:          0          0          0          0     GICv2 130 Level     gpmmu
 55:          0          0          0          0     GICv2 131 Level     pp0
 56:          0          0          0          0     GICv2 132 Level     ppmmu0
 57:          0          0          0          0     GICv2 134 Level     pp1
 58:          0          0          0          0     GICv2 135 Level     ppmmu1
 60:    1203879          0          0          0     GICv2  63 Level     ths
 61:          0          0          0          0     GICv2 152 Level     arm-pmu
 62:          0          0          0          0     GICv2 153 Level     arm-pmu
 63:          0          0          0          0     GICv2 154 Level     arm-pmu
 64:          0          0          0          0     GICv2 155 Level     arm-pmu
109:          1          0          0          0  sunxi_pio_edge  44 Edge      usb0-id-det
132:          1          0          0          0  sunxi_pio_edge   3 Edge      sw4
IPI0:         0          0          0          0  CPU wakeup interrupts
IPI1:         0          0          0          0  Timer broadcast interrupts
IPI2:    188159     559359     266176     229523  Rescheduling interrupts
IPI3:     35362      29678      15459       9251  Function call interrupts
IPI4:         0          0          0          0  CPU stop interrupts
IPI5:       431        457        290        440  IRQ work interrupts
IPI6:         0          0          0          0  completion interrupts
Err:          0
```

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

```
$ cat /proc/stat
cpu  960 352 3120 226661 670 9 60 0 0
cpu0 265 103 367 56843 304 8 47 0 0
cpu1 246 102 824 56630 204 1 12 0 0
cpu2 224 51 863 56717 80 0 0 0 0
cpu3 224 94 1065 56470 80 0 0 0 0
intr 45959 49 8 0 0 2 0 0 0 1 0 0 0 144 0 0 0 30 0 0 12825 0 0 572 0 0 0 0 249 0 0 ...
```

Здесь строка, начинающаяся с `intr`, содержит (начиная со второго числового значения) *суммарное* число обработанных прерываний по всем процессорам для всех последовательных номеров линий IRQ (IRQ 0, IRQ 1, IRQ 2, ... — сравните с ранее показанным предыдущим выводом для *x*86, они сделаны **почти** одновременно).

Мы можем не только наблюдать распределение прерываний по процессорам, но и *управлять* им (что понадобится вскоре):

```
# cat /proc/irq/27/smp_affinity
4
# cat /proc/interrupts | grep eth0
 27:          83          0       4102          0   PCI-MSI-edge      eth0
```

Видим, что уже после загрузки аффинити-маска (битовая маска разрешенных процессоров) для обработчика прерывания IRQ 27 была установлена системой в 4(100). Мы можем изменить это распределение на 2(10):

```
# echo 2 > /proc/irq/27/smp_affinity
# cat /proc/irq/27/smp_affinity
2
# watch -n1 'cat /proc/interrupts | grep eth0'
 27:          83        117       4111          0   PCI-MSI-edge      eth0
```

Регистрация обработчика прерывания

Все функции и определения, реализующие интерфейс обработчика прерывания, объявлены в <linux/interrupt.h>. Первое, что мы всегда должны делать, — это зарегистрировать функцию-обработчик (ISR) прерываний (все прототипы показаны для ядра 5.4, но определения в этой части мало меняются с ядра 2.6.37):

```
typedef irqreturn_t (*irq_handler_t)(int, void*);
inline int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
                const char *name, void *dev);
const void free_irq(unsigned int irq, void *dev);
```

Здесь:

- `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 неразделяемых 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 допустимой могла бы быть конструкция:

```
for(int i = 0; i < N; i++) request_irq(irq, handler, 0, const char *name, (void*)i);
...
for(int i = 0; i < N; i++) free_irq(irq, (void*)i);
```

Здесь указатель `(void*)i` абсолютно бессмысленный в *качестве указателя*, но совершенно нормальный по назначению своего использования. В некоторых случаях показанный фрагмент — это вполне пригодный фрагмент кода. Но позже оказалось целесообразным и удобным использовать именно в качестве `*dev` указатель на специфическую для устройства структуру, которая содержит все характерные данные экземпляра устройства, — поскольку для каждого экземпляра размещается своя копия структуры, то указатели на них и будут уникальными, что и требовалось. Этот же указатель будет передаваться в функцию-обработчик `handler()` вторым параметром при *каждом* прерывании, тем самым передавая уникальный идентификатор источника произошедшего прерывания. На сегодня это общеупотребимая практика — увязывать обработчик прерывания со структурами данных устройства.

> ***Примечание***
>
> Прототипы `irq_handler_t` и флаги установки обработчика существенно меняются от версии к версии. Например, радикально поменялись после 2.6.19 все флаги, именуемые сейчас `IRQF_*`, — до этого они именовались `SA_*`. В результате 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, а 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, закрепленное 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!), он указывает на то, что эта линия запроса на прерывание уже занята (или при те-

кущем вызове, или при предыдущем вызове для этой линии не был указан флаг IRQF_SHARED).

С регистрацией нового обработчика прерываний все просто. Но здесь есть одна маленькая (нигде не документированная — мне, по крайней мере, не удалось найти) деталь, которая может вызвать большую досаду при работе. Параметр name вызова request_irq() — это просто *указатель* на константную строку имени, но эта строка никуда не копируется (как это обычно принято в API пространства пользователя), и указатель указывает на строку все время, пока загружен модуль. А отсюда следуют далеко идущие последствия. Вот такой вызов в функции инициализации модуля будет замечательно работать:

```
request_irq(irq, handler, 0, "my_interrupt", NULL);
```

И с таким будет всё, как вы ожидали:

```
int init_module(void) {
   char *dev = "my_interrupt";
   ...
   request_irq(irq, handler, 0, dev, NULL);
}
```

Но уже вот такой, очень похожий код даст вам в /proc/interrupts нечитаемую ерунду (и это в лучшем случае, если вам сильно повезет, — вы в ядре!):

```
int init_module(void) {
   char dev[] = "my_interrupt";
   ...
   request_irq(irq, handler, 0, dev, NULL);
}
```

Здесь строка имени размещена и инициализирована в стеке, и после завершения функции инициализации она уже не существует... но модуль-то существует? Перепишем этот фрагмент, и всё опять заработает:

```
char dev[] = "my_interrupt";
...
int init_module(void) {
   ...
   request_irq(irq, handler, 0, dev, NULL);
}
```

Особенно сложно подлежащие толкованию результаты вы получите из-за этой особенности, если в одном модуле собираетесь зарегистрировать несколько обработчиков прерываний:

```
int init_module(void) {
   int i;
   char *dev = "serial_xx";
   for(i = 0; i < num; i++) {
      sprintf(dev, "serial_%02d", i + 1);
      request_irq(irq, handler, IRQF_SHARED, dev, (void*)(i + 1));
   }
```

И что мы получим в этом случае? Правильно, мы получим num идентичных копий имен обработчиков (идентичных последнему заполнению), поскольку все экземпляры обработчиков будут использовать одну копию строки имени:

```
$ cat /proc/interrupts
...
  22:       1652   IO-APIC-fasteoi   ohci_hcd:usb2, serial_04, serial_04, serial_04, serial_04
...
```

И еще одно очень серьезное предупреждение относительно удаления обработчика free_irq(). В приложениях пространства пользователя мы можем себе позволить изрядную небрежность относительно завершающих действий программы: не выполнять close() для открытых дескрипторов файлов, не выполнять free() для динамически выделенных блоков памяти... Эти шалости прощаются пользователю, поскольку при завершении программы выполняется так называемый *эпилог*, в процессе которого система выполнит все не указанные явно действия. Если же в модуле при его завершении (выгрузке) вы не выполните явно free_irq(), то почти со 100-процентной вероятностью произойдет следующее:

- модуль будет выгружен, но вектор прерывания будет установлен на тот адрес, который перед тем занимала зарегистрированная функция handler();
- по истечении некоторого времени эта область памяти будет переписана ядром под какие-то иные цели;
- и первое же произошедшее затем аппаратное прерывание по этой линии IRQ приведет к немедленному краху всей системы.

Обработчик прерываний: верхняя половина

Сам обработчик прерывания разработчики аппаратных решений в других операционных системах часто называют ISR (Interrupt Service Routine). Это и есть то, что в Linux обозначают как «верхняя половина» обработчика прерывания.

Прототип функции обработчика прерывания уже показывался ранее:

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

где :

- irq — линия IRQ. Чем меньше номер линии IRQ, тем выше приоритет обработки этого прерывания;
- dev — уникальный указатель экземпляра обработчика (именно тот, который передавался последним параметром request_irq() при регистрации обработчика).

Это именно та функция, которая будет вызываться в первую очередь при каждом возникновении аппаратного прерывания. Но нет никакой гарантии, что при возврате из нее работа по обработке текущего прерывания будет завершена (хотя и такой вариант вполне допустим). Из-за такой «неполноты» этот обработчик и получил название «верхней половины» обработчика прерывания. Дальнейшие действия по обработке могут быть запланированы этим обработчиком на более позднее время,

с использованием нескольких различных механизмов, обобщенно называемых «нижней половиной».

Важно то, что код обработчика верхней половины выполняется при *запрещенных* 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)
```

Здесь:

- `IRQ_HANDLED` — устройство прерывания распознано как обслуживаемое обработчиком, и прерывание успешно обработано;
- `IRQ_NONE` — устройство не является источником прерывания для этого обработчика. Прерывание должно быть передано далее другим обработчикам, зарегистрированным на этой линии IRQ.

Типичная схема обработчика при этом будет выглядеть так:

```
static irqreturn_t intr_handler (int irq, void *dev) {
   if(! /* проверка того, что обслуживаемое устройство запросило прерывание*/)
      return IRQ_NONE;
   /* код обслуживания устройства */
   return IRQ_HANDLED;
}
```

Пока мы не углубились в дальнейшую обработку, производимую в нижней половине, хотелось бы отметить следующее: в ряде случаев (при крайне простой обработке, но, самое главное, при отсутствии возможности очень быстрых наступлений повторных прерываний) оказывается вполне достаточно простого обработчика верхней половины, и нет необходимости мудрить со сложно диагностируемыми механизмами отложенной обработки.

Здесь же напомним *самую большую сложность* в программировании функции обработчика: код функции ни непосредственно, ни косвенно, через сколь угодно длинную цепочку вложенных вызовов, не имеет права вызывать ни единой функции API ядра, которые вызывают переход выполняющегося кода в *блокированное* состояние ожидания (`msleep()` и др.) или даже только *предполагают* возможность такого перехода при некоторых условиях (`kmalloc()` и др.)! Это связано с тем, что функция обработчика верхней половины выполняется *в контексте прерывания*, и после завершения блокированного ожидания будет просто некуда возвратиться для продолжения ее выполнения. Результатом такого ошибочного вызова станет с большой вероятностью крах всей операционной системы (отметим, что таких блокирующих вызовов в составе API ядра достаточно много).

Управление линиями прерывания

Под управлением линиями прерываний в этом месте описания мы будем понимать запрет/разрешение прерываний по одной или нескольким линиям `irq`. Раньше существовала возможность вообще запретить прерывания (на время, естественно). Но сейчас («заточенный» под SMP) набор API для этих целей выглядит так: либо вы запрещаете прерывания по всем линиям `irq`, но локального процессора, либо на всех процессорах, но только для одной линии `irq`.

Макросы управления линиями прерываний определены в <linux/irqflags.h>. Управление запретом и разрешением прерываний на локальном процессоре:

- `local_irq_disable()` — запретить прерывания на локальном CPU;
- `local_irq_enable()` — разрешить прерывания на локальном CPU;
- `int irqs_disabled()` — возвратить ненулевое значение, если запрещены прерывания на локальном CPU, в противном случае возвращается нуль.

Напротив, управление (запрет и разрешение) одной выбранной линией `irq`, но уже относительно всех процессоров в системе, делают макросы:

- `void disable_irq(unsigned int irq)` и `void disable_irq_nosync(unsigned int irq)` — обе эти функции запрещают прерывания с линии `irq` на контроллере (для всех CPU), причем `disable_irq()` не возвращается до тех пор, пока все обработчики прерываний, которые в текущий 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 целом хорошая идея — всегда вызывать эту функцию перед выгрузкой модуля, использующего указанную 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 бывает затруднительно показать работающий код обработчика прерываний, потому что такой код должен был бы быть связан с реальным аппаратным расширением, в результате чего он окажется перегружен специфическими деталями, скрывающими суть происходящего. Но оригинальный пример приведен в [6], откуда мы его и заимствуем (см. папку IRQ в сопровождающем книгу файловом архиве):

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 еще 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, еще не имея самого устройства, и имитировать его прерывания одним из штатных источников прерываний компьютера — с тем, чтобы позже все 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 задействован один из способов организации отложенной обработки, который так и именовался: *обработчик нижней половины*, но сейчас он неприменим. А термин так и остался в качестве нарицательного, относящегося ко всем разным способам организации отложенной обработки, которые и рассматриваются далее.

Отложенные прерывания: *softirq*

Отложенные прерывания определяются статически *во время компиляции ядра*. Поэтому, забегая вперед, примем на заметку, что технику отложенных прерываний можно реализовать *в чистом виде*, только если произвести пересборку ядра Linux под свои требования.

Отложенные прерывания представлены с помощью структур `softirq_action`, определенных в файле <linux/interrupt.h> в следующем виде (ядро 5.4):

```
// структура, представляющая одно отложенное прерывание
struct softirq_action {
```

```
   void (*action)(struct softirq_action *);
};
```

В ядре 2.6.18 (и много где в литературе) вы увидите другое (более раннее) определение:

```
struct softirq_action {
   void (*action)(struct softirq_action *);
   void *data;
};
```

Для уточнения картины с `softirq_action` вам будет недостаточно заголовочных файлов (это один из редких таких случаев), и необходимо будет окунуться в рассмотрение исходных кодов *реализации* ядра (файл <kernel/softirq.c>). Если же у вас не установлены исходные тексты ядра, то нужно либо сделать это, либо, что более разумно, обратиться к следующим ресурсам: **http://lxr.free-electrons.com/** или **http://lxr.linux.no/**:

```
enum {          /* задействованные номера */
   HI_SOFTIRQ=0,
   TIMER_SOFTIRQ,
   NET_TX_SOFTIRQ,
   NET_RX_SOFTIRQ,
   BLOCK_SOFTIRQ,
   BLOCK_IOPOLL_SOFTIRQ,
   TASKLET_SOFTIRQ,
   SCHED_SOFTIRQ,
   HRTIMER_SOFTIRQ,
   RCU_SOFTIRQ, /* Preferable RCU should always be the last softirq */
   NR_SOFTIRQS  /* число задействованных номеров */
};
static struct softirq_action softirq_vec[NR_SOFTIRQS]
char *softirq_to_name[NR_SOFTIRQS] = {
   "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "BLOCK_IOPOLL",
   "TASKLET", "SCHED", "HRTIMER",<>"RCU"
};
```

В 2.6.18 (то, что кочует из одного литературного источника в другой) аналогичные описания были заметно проще и статичнее:

```
enum {
   HI_SOFTIRQ=0,
   TIMER_SOFTIRQ,
   NET_TX_SOFTIRQ,
   NET_RX_SOFTIRQ,
   BLOCK_SOFTIRQ,
   TASKLET_SOFTIRQ
};
static struct softirq_action softirq_vec[32]
```

Здесь имеется возможность создать 32 обработчика softirq, и это количество *фиксировано*. В этой версии ядра (2.6.18) их было 32, из которых задействованных 6. Их определения из предыдущей версии помогают лучше понять то, что имеет место в настоящее время.

Динамическая диагностика использования softirq в работающей системе может производиться так:

```
$ cat /proc/softirqs
                CPU0       CPU1       CPU2       CPU3
      HI:          0          0          0          0
   TIMER:     764858     806727     615475     617660
  NET_TX:          0          0       2520          0
  NET_RX:        777        700     471255        521
   BLOCK:      48435     112893         12      24613
BLOCK_IOPOLL:       0          0          0          0
 TASKLET:        108         81          5          1
   SCHED:     405389     430031     327195     345751
 HRTIMER:        985       1356       1211       1518
     RCU:     884440     850717     837750     672217
```

В любом случае (независимо от версии) добавить новый уровень обработчика (назовем его XXX_SOFT_IRQ) без перекомпиляции ядра мы не сможем. Максимальное число используемых обработчиков softirq не может быть динамически изменено. Отложенные прерывания с меньшим номером выполняются раньше отложенных прерываний с большим номером (приоритетность). Обработчик одного отложенного прерывания никогда не вытесняет другой обработчик softirq. Единственное событие, которое может вытеснить обработчик softirq, — это аппаратное прерывание. Однако на другом процессоре одновременно с обработчиком отложенного прерывания может выполняться другой (*и даже этот же*) обработчик отложенного прерывания. Отложенное прерывание выполняется *в контексте прерывания*, а значит, для него недопустимы блокирующие операции.

Если вы решились на пересборку ядра (в чем нет ничего страшного) и создание нового уровня softirq, то для этого необходимо:

1. Определить новый индекс (уровень) отложенного прерывания, вписав (файл <linux/interrupt.h>) свою константу вида XXX_SOFT_IRQ в перечисление — где-то, очевидно, на одну позицию выше TASKLET_SOFTIRQ (иначе зачем переопределять новый уровень и не использовать тасклет?).
2. Во время инициализации модуля должен быть зарегистрирован (объявлен) обработчик отложенного прерывания с помощью вызова open_softirq(), который принимает три параметра: этот индекс отложенного прерывания, функцию-обработчик и значение поля data :

```
/* The bottom half */
void xxx_analyze(void *data) {
  /* Analyze and do ..... */
}
```

```
void __init roller_init() {
   /* ... */
   request_irq(irq, xxx_interrupt, 0, "xxx", NULL);
   open_softirq(XXX_SOFT_IRQ, xxx_analyze, NULL);
}
```

3. Функция-обработчик отложенного прерывания (так же как и рассматриваемого далее тасклета) должна в точности соответствовать правильному прототипу:

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

4. Зарегистрированное отложенное прерывание — чтобы оно было поставлено в очередь на выполнение — должно быть отмечено (генерировано, возбуждено: `rise softirq`). Это называется *генерацией отложенного прерывания*. Обычно обработчик аппаратного прерывания (ISR, верхней половины) перед возвратом возбуждает свои обработчики отложенных прерываний:

```
/* The interrupt handler */
static irqreturn_t xxx_interrupt(int irq, void *dev_id) {
   /* ... */
   /* Mark softirq as pending */
   raise_softirq(XXX_SOFT_IRQ);
   return IRQ_HANDLED;
}
```

5. Затем в подходящий (не для вас, для системы) момент времени отложенное прерывание начнет выполняться. Обработчик отложенного прерывания выполняется при *разрешенных* прерываниях процессора (особенность нижней половины). Во время выполнения обработчика отложенного прерывания новые отложенные прерывания *на задействованном* процессоре запрещаются. Однако на другом процессоре обработчики отложенных прерываний могут выполняться. На самом деле, если вдруг генерируется отложенное прерывание в тот момент, когда еще 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 задействованы в обработчике отложенного прерывания, и даже глобальные данные, которые используются только внутри самого обработчика, должны соответствующим образом ограждаться примитивами синхронизации.

С одной стороны, все достаточно просто — можете поверить, а еще лучше проверить, что набросанной ранее схемы вполне достаточно для реализации кода. С другой стороны, всё как-то тяжеловесно и усложнено, особенно в сравнении с альтернативными методами, рассматриваемыми далее. Вопрос: зачем? Ответ прост: главная причина использования отложенных прерываний — превосходная *масштабируемость на многие процессоры*. Это становится актуальным сегодня, когда число ядер ординарного персонального компьютера может составлять 8 и способно серьезно увеличиться в ближайшее время. Но если нет необходимости масштабирования на многие процессоры, то лучшим выбором будет механизм тасклетов.

Тасклеты

> ***Пояснение***
>
> Не ищите перевода термину «tasklet», это жаргон: если `task` — это отдельная задача в терминах ядра, то `tasklet` мы воспринимаем как «маленькую задачку», маленький отдельный поток.

Предыдущая схема достаточно тяжеловесная, и в большинстве случаем ее подменяют тасклетами — механизмом на базе тех же `softirq` с двумя фиксированными индексами `HI_SOFTIRQ` или `TASKLET_SOFTIRQ`. Тасклеты — это не 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) может 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 применена 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;
}
```

Когда 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 кеша 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 текущий момент уже начал выполнение, то функция не возвратит управление, пока тасклет не закончит свое действие. В качестве альтернативы можно использовать функцию `tasklet_disable_nosync()`, которая запрещает указанный тасклет, но возвращается сразу, не ожидая, пока тасклет завершит выполнение (что обычно небезопасно, т. к. в этом 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 одном из уровней которых он построен), тасклет не может переходить в блокированное состояние.

Демон *ksoftirqd*

Обработка отложенных прерываний (softirq) и соответственно тасклетов осуществляется с помощью набора потоков пространства ядра (по одному потоку на каждый процессор). Потоки пространства ядра помогают обрабатывать отложенные прерывания, когда система перегружена большим количеством отложенных прерываний:

```
$ ps -ALf | head -n12
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root         1     0     1  0    1 08:55 ?        00:00:01 /sbin/init
...
root         4     2     4  0    1 08:55 ?        00:00:00 [ksoftirqd/0]
...
root         7     2     7  0    1 08:55 ?        00:00:00 [ksoftirqd/1]
...
```

Для каждого *процессора* существует свой поток. Каждый поток имеет имя в виде ksoftirqd/n, где n — номер процессора. Например, в двухпроцессорной системе будут запущены два потока с именами ksoftirqd/0 и ksoftirqd/1. То, что на каждом процессоре выполняется свой поток, гарантирует, что если в системе есть свободный процессор, то он всегда будет в состоянии выполнять отложенные прерывания. После того как потоки запущены, они выполняют замкнутый цикл.

Здесь же попутно уместно напомнить, что в современном ядре Linux, даже в самых ординарных конфигурациях, может выполняться совсем немалое число автономных потоков ядра, и в разных дистрибутивах и архитектурах процессора это число может очень сильно различаться:

```
$ ps -ALf | grep ' \[' | wc -l
410
$ ps -ALf | grep ' \[' | wc -l
84
```

Очереди отложенных действий: *workqueue*

Очереди отложенных действий (workqueue) — это еще один, но совершенно иной 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 перехо-

дить в блокированные состояния. Рабочие потоки, которые выполняются по умолчанию, называются `events/n`, где `n` — номер процессора, например:

```
$ ps -ALf | grep '\['events
root        15     2    15  0    1 17:08 ?        00:00:00 [events/0]
root        16     2    16  0    1 17:08 ?        00:00:00 [events/1]
root        17     2    17  0    1 17:08 ?        00:00:00 [events/2]
root        18     2    18  0    1 17:08 ?        00:00:00 [events/3]
```

Когда какие-либо действия ставятся в очередь, поток ядра возвращается к выполнению и выполняет эти действия. Когда в очереди не остается работы, которую нужно выполнять, поток снова возвращается в состояние ожидания. Каждое действие представлено с помощью `struct work_struct` (определяется в файле <linux/workqueue.h>, причем 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-обработчик 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` выполним последовательность действий примерно следующего содержания:

1. При инициализации модуля создаем 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);
}
```

2. 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 задействованном 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 незавершенных отложенных действий с задержками используется функция:

     ```
     int cancel_delayed_work(struct work_struct *work);
     ```

Но мы совсем необязательно должны рассчитывать на общую очередь (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 ранее `schedule_*()`, но работают с созданной очередью, указанной первым 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 ее применения (в отличие, например, от тасклетов) для других целей.

Сравнение и примеры

Начнем со сравнений. Оставив в стороне рассмотрение `softirq` — это механизм тяжелый и уже достаточно обсужденный, в том смысле, что его использование оправданно при требовании масштабирования высокоскоростных процессов на большое число обслуживающих процессоров в SMP. Две другие рассмотренные схемы: это тасклеты и очереди отложенных действий. Они представляют собой две различные схемы реализации отложенных работ в современном Linux, которые переносят работы из верхних половин драйверов в их нижние половины. 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 различие: 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

мной занятий неоднократно задавался вопрос: «Можно ли тасклеты использовать автономно, вне процесса обработки прерываний?». Вот так мы и построим иллюстрирующие модули — сама функция инициализации модуля будет активировать отложенную обработку. Вот пример для тасклетов:

mod_tasklet.c

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

MODULE_AUTHOR("Oleg Tsiliuric <olej@front.ru>");
MODULE_LICENSE("GPL v2");

static cycles_t cycles1, cycles2;
static u32 j1, j2;
static int context;

char tasklet_data[] = "tasklet function was called";

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

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

int init_module(void) {
   context = in_atomic();
   j1 = jiffies;
   cycles1 = get_cycles();
   tasklet_schedule(&my_tasklet);   /* Schedule the Bottom Half */
   printk("%010lld [%05d] : tasklet was scheduled in contxt %d\n",
           (long long unsigned)cycles1, j1, context);
   return 0;
}

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

Вот как выглядит его исполнение:

```
$ uname -a
Linux R420 5.4.0-124-generic #140-Ubuntu SMP Thu Aug 4 02:23:37 UTC 2022 x86_64 x86_64 x86_64
GNU/Linux
$ sudo insmod mod_tasklet.ko
$ lsmod | head -n3
Module                  Size  Used by
mod_tasklet            16384  0
vboxnetadp             28672  0
$ dmesg | tail -n2 | grep " : "
[41665.550350] 100615043133471 [10341295] : tasklet was scheduled in contxt 0
[41665.550357] 100615043164872 [10341295] : tasklet function was called in contxt 1
$ sudo rmmod mod_tasklet
$ ../time/clock
00005BC0A00EFD1C
00005BC0A01327F9
00005BC0A0133D2C
2399990329
```

По временным меткам видно, что выполнение функции тасклета происходит позже планирования тасклета на выполнение, но латентность очень низкая (системный счетчик `jiffies` не успевает изменить значение — все происходит в пределах одного системного тика), отсрочка выполнения составляет порядка 30 000 процессорных тактов частоты 2,5 Гц (показан уже обсуждавшийся тест `clock` из раздела о службе времени, нас интересует только последняя строка его вывода). Специально показана диагностика контекста в функции инициализации модуля (контекст процесса) и в функции тасклета (атомарный контекст).

> ***Замечание вдогонку***
>
> Начиная с версий ядра 5.9–5.10, в коде ядра появились замечания, объявляющие многолетний механизм тасклетов deprecated, а API тасклетов существенно изменен (начиная с замены *определения* `struct tasklet_struct`). Это слишком свежие изменения, чтобы их пытаться комментировать, но я приведу вариант предыдущего теста в таком виде, при котором он *компилируется* и точно так же *выполняется*, как ранее:
>
> ```
> $ inxi -S
> System:    Host: lmde32 Kernel: 5.10.0-16-686 i686 bits: 32 Desktop: Cinnamon 5.4.10
> Distro: LMDE 5 Elsie
> ```

mod_tasklet.c

```
#include <linux/module.h>
#include <linux/interrupt.h>
#include <linux/version.h>      /* LINUX_VERSION_CODE */

MODULE_AUTHOR("Oleg Tsiliuric <olej@front.ru>");
MODULE_LICENSE("GPL v2");
```

```
static cycles_t cycles1, cycles2;
static u32 j1, j2;
static int context;

char tasklet_data[] = "tasklet function was called";

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

#if LINUX_VERSION_CODE < KERNEL_VERSION(5,9,0)
DECLARE_TASKLET(my_tasklet, tasklet_function,
                (unsigned long)&tasklet_data);
#else
DECLARE_TASKLET_OLD(my_tasklet, tasklet_function);
#endif

int init_module(void) {
        context = in_atomic();
        j1 = jiffies;
        cycles1 = get_cycles();
        my_tasklet.data = (unsigned long)&tasklet_data; //(char*)data;
        tasklet_schedule(&my_tasklet);           /* Schedule the Bottom Half */
        printk("%010lld [%05d] : tasklet was scheduled in contxt %d\n",
             (long long unsigned)cycles1, j1, context);
        return 0;
}

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

А в следующем примере мы проделаем практически то же самое (близкие эксперименты для возможности сравнения), но относительно очередей отложенных действий:

mod_workqueue.c

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

```
MODULE_AUTHOR("Oleg Tsiliuric <olej@front.ru>");
MODULE_LICENSE("GPL v2");

static int works = 2; // число отложенных работ
module_param(works, int, S_IRUGO);

static struct workqueue_struct *my_wq;

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

static void my_wq_function(struct work_struct *work) { /* Bottom Half Function */
   u32 j = jiffies;
   cycles_t cycles = get_cycles();
   my_work_t *wrk = (my_work_t *)work;
   printk("#%d : %010lld [%05d] => %010lld [%05d] = %06lu : context %d\n",
           wrk->id, (long long unsigned)wrk->cycles, wrk->j,
           (long long unsigned)cycles, j,
           (long unsigned)(cycles - wrk->cycles), in_atomic()
         );
  kfree((void*)wrk);
  return;
}

int init_module(void) {
   int n, ret;
   if((my_wq = create_workqueue("my_queue")))
      for(n = 0; n < works; n++) {
         /* One more additional work */
         my_work_t *work = (my_work_t*)kmalloc(sizeof(my_work_t), GFP_KERNEL);
         if(work) {
            INIT_WORK((struct work_struct *)work, my_wq_function);
            work->id = n;
            work->j = jiffies;
            work->cycles = get_cycles();
            ret = queue_work(my_wq, (struct work_struct*)work);
            if(!ret) return -EPERM;
         }
         else return -ENOMEM;
      }
   else return -EBADRQC;
   return 0;
}
```

```
void cleanup_module(void) {
  flush_workqueue(my_wq);
  destroy_workqueue(my_wq);
  return;
}
```

Вот как исполнение проходит на этот раз на том же самом компьютере:

```
$ sudo insmod mod_workqueue.ko works=5
$ lsmod | head -n3
Module                  Size  Used by
mod_workqueue          16384  0
vboxnetadp             28672  0
$ ps -ef | grep root | grep my_
root       41699       2  0 22:39 ?        00:00:00 [my_queue]
$ dmesg | tail -n5
[42052.166168] #0 : 101542920708030 [10437950] => 101542920719181 [10437950] = 011151 : context 0
[42052.166174] #1 : 101542920734316 [10437950] => 101542920737709 [10437950] = 003393 : context 0
[42052.166177] #2 : 101542920743535 [10437950] => 101542920745674 [10437950] = 002139 : context 0
[42052.166179] #3 : 101542920750258 [10437950] => 101542920752409 [10437950] = 002151 : context 0
[42052.166182] #4 : 101542920756741 [10437950] => 101542920759477 [10437950] = 002736 : context 0
$ sudo rmmod mod_workqueue
```

Здесь мы видим, что появился новый обрабатывающий поток ядра с заданным нами именем. Теперь латентность реакции несколько больше случая тасклетов, что и соответствует утверждениям в литературе. Снова показана диагностика контекста в функциях отложенных работ, и теперь она — контекст процесса.

Обсуждение

При рассмотрении техники обработки прерываний возникает ряд тонких вопросов, на которые меня натолкнули участники проводимых мною тренингов. Одна из таких интересных групп вопросов (потому что здесь, собственно, два вопроса) выглядит так:

- при регистрации нескольких обработчиков прерываний, разделяющих одну линию IRQ, какой будет порядок срабатывания по времени этих обработчиков (связанных в последовательный список): от позже зарегистрированных к более ранним (что было бы целесообразно) или же наоборот?
- при регистрации нескольких обработчиков прерываний, разделяющих одну линию IRQ, есть ли способы изменения последовательности срабатывания этих нескольких обработчиков?

На второй вопрос я (пока) не знаю ответа, а вот относительно первого рассмотрим еще вот такой тест:

mod_ser.c

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

```
MODULE_LICENSE("GPL v2");
#define SHARED_IRQ 1
#define MAX_SHARED 9
#define NAME_SUFFIX "serial_"
#define NAME_LEN   10
static int irq = SHARED_IRQ, num = 2;
module_param(irq, int, 0);
module_param(num, int, 0);

static irqreturn_t handler(int irq, void *id) {
   cycles_t cycles = get_cycles();
   printk(KERN_INFO "%010lld : irq=%d - handler #%ld\n", cycles, irq, (long)id);
   return IRQ_NONE;
}

static char dev[MAX_SHARED][NAME_LEN];

int init_module(void) {
   long i;
   if(num > MAX_SHARED) num = MAX_SHARED;
   for(i = 0; i < num; i++) {
      sprintf(dev[i], "serial_%02ld", i + 1);
      if(request_irq(irq, handler, IRQF_SHARED, dev[i], (void*)(i + 1))) return -1;
   }
   return 0;
}

void cleanup_module(void) {
   long i;
   for(i = 0; i < num; i++) {
      synchronize_irq(irq);
      free_irq(irq, (void*)(i + 1));
   }
}
```

Здесь на одну (любую) линию IRQ (параметр модуля `irq`) устанавливается `num` последовательных обработчиков прерывания (по умолчанию параметр `num` равен 2), которые фиксируют время своего срабатывания. Испытываем этот модуль *на реальном* (не виртуальном) оборудовании (тонкий клиент T520 от HP):

```
$ inxi -MSxxx
System:
  Host: antix21 Kernel: 4.9.0-279-antix.1-amd64-smp arch: x86_64 bits: 64
    compiler: gcc v: 10.2.1 Desktop: IceWM v: 2.9.8 vt: 7 dm: slimski v: 1.5.0
    Distro: antiX-21_x64-base Grup Yorum 31 October 2021
    base: Debian GNU/Linux 11 (bullseye)
```

```
Machine:
  Type: Desktop System: Hewlett-Packard product: HP t520 Flexible Series TC
    v: N/A serial: <superuser required> Chassis: type: 4
    serial: <superuser required>
  Mobo: Hewlett-Packard model: 21EF v: 00~ serial: <superuser required>
    UEFI: AMI v: L41 v01.09 date: 10/19/2016
$ uname -r
4.9.0-279-antix.1-amd64-smp
```

Исследуем интересные для нас IRQ устройства на шине USB (USB-устройства я выбрал только из-за большей простоты — вы можете выбрать любой класс устройств):

```
$ lsusb
Bus 002 Device 002: ID 0438:7900 Advanced Micro Devices, Inc. Root Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 003: ID 2101:020f ActionStar
Bus 001 Device 002: ID 0438:7900 Advanced Micro Devices, Inc. Root Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
$ lsusb -t
/:  Bus 04.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/2p, 5000M
/:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/2p, 480M
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=ehci-pci/2p, 480M
    |__ Port 1: Dev 2, If 0, Class=Hub, Driver=hub/4p, 480M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci-pci/2p, 480M
    |__ Port 1: Dev 2, If 0, Class=Hub, Driver=hub/4p, 480M
        |__ Port 2: Dev 3, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
        |__ Port 2: Dev 3, If 1, Class=Human Interface Device, Driver=usbhid, 1.5M
```

Здесь только одно внешнее (подключенное) устройство на USB (против ожидания), но только потому, что и клавиатура, и мышь у меня здесь подключены через единый порт USB с помощью коммутатора D-Link DKMV-U. И вот линии прерываний USB:

```
$ cat /proc/interrupts | grep ehci
  18:       5374       5412   IO-APIC   18-fasteoi   ehci_hcd:usb1, ehci_hcd:usb2
$ cat /proc/interrupts | grep ahci
  19:       9519       9524   IO-APIC   19-fasteoi   ahci[0000:00:11.0]
```

Подвесим тестируемый модуль на такой IRQ:

```
$ sudo insmod mod_ser.ko irq=18
$ lsmod | head -n3
Module                  Size  Used by
mod_ser                 1508  0
8021q                  20506  0
```

Вот фрагмент системного журнала при перемещении мыши:

```
$ dmesg | grep 'irq=18' | head -n6
[47712.102929] 57157472571836 : irq=18 - handler #2
```

```
[47712.142921] 57157520469724 : irq=18 - handler #1
[47712.142923] 57157520474041 : irq=18 - handler #2
[47712.166916] 57157549208617 : irq=18 - handler #1
[47712.166918] 57157549213276 : irq=18 - handler #2
[47712.174914] 57157558788144 : irq=18 - handler #1
```

Подвигаем мышью:

```
$ dmesg | grep 'irq=18' | head -n6
[47738.220175] 57188753720372 : irq=18 - handler #2
[47738.244184] 57188782470106 : irq=18 - handler #1
[47738.244188] 57188782481019 : irq=18 - handler #2
[47738.252183] 57188792047577 : irq=18 - handler #1
[47738.252187] 57188792061274 : irq=18 - handler #2
[47738.260183] 57188801629465 : irq=18 - handler #1
```

Хорошо видно (по меткам времени) как побежали прерывания:

```
$ sudo rmmod mod_ser
$ lsmod | head -n3
Module                  Size  Used by
8021q                  20506  0
garp                    5898  1 8021q
```

После выгрузки подсчитаем, сколько мы нащелкали прерываний:

```
$ dmesg | grep 'irq=18' | wc -l
2519
```

Здесь уже по меткам счетчика процессорных тактов (`rdtsc`) мы можем предположить, что ранее зарегистрированный обработчик и срабатывает раньше, т. е. новые обработчики прерывания устанавливаются в хвост очереди разделяемых прерываний.

Далее уместно рассмотреть то же, но в инсталляции *виртуальной* — для определенности под VirtualBox. Для интереса к таким экспериментам есть несколько оснований: а) подтвердить, что и в виртуальной установке доступны к использованию и изучению *аппаратные* ресурсы; б) убедиться, что в одном и том же гипервизоре виртуальных машин разные дистрибутивы получают *разное* отображение аппаратных ресурсов и их нужно искать, и в) наконец, подойти к вопросу: «А как мы все это будем отлаживать?», который нам вскоре предстоит рассматривать во всей полноте... Итак:

```
$ sudo inxi -MS
System:
  Host: antix21 Kernel: 4.9.0-279-antix.1-486-smp arch: i686 bits: 32
    Desktop: IceWM v: 2.9.7
    Distro: antiX-21_386-base Grup Yorum 31 October 2021
Machine:
  Type: Virtualbox System: innotek GmbH product: VirtualBox v: 1.2
    serial: N/A
```

```
  Mobo: Oracle model: VirtualBox v: 1.2 serial: N/A BIOS: innotek GmbH
    v: VirtualBox date: 12/01/2006
$ cat /proc/interrupts | grep 'hci_'
 19:          0     126308   IO-APIC  19-fasteoi   ehci_hcd:usb1, eth0
 22:          0        107   IO-APIC  22-fasteoi   ohci_hcd:usb2
$ lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 004: ID 80ee:0021 VirtualBox USB Tablet
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
$ lsusb -t
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=ohci-pci/12p, 12M
    |__ Port 1: Dev 4, If 0, Class=Human Interface Device, Driver=usbhid, 12M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci-pci/12p, 480M
$ cat /sys/class/input/mouse1/device/name
VirtualBox USB Tablet
```

Интересующее нас устройство — это IRQ 19:

```
$ sudo insmod mod_ser.ko irq=19
$ lsmod | head -n3
Module                  Size  Used by
mod_ser                16384  0
nls_utf8               16384  1
$ dmesg | grep 'irq=19' | tail -n6
[105715.542283] 259122846589477 : irq=19 - handler #1
[105715.542286] 259122846619329 : irq=19 - handler #2
[105716.264454] 259124616609721 : irq=19 - handler #1
[105716.264458] 259124616675683 : irq=19 - handler #2
[105718.269738] 259129531642027 : irq=19 - handler #1
[105718.269741] 259129531660762 : irq=19 - handler #2
```

Подвигаем мышью, периодически контролируя прерывания, зафиксированные *нашим модулем*:

```
$ dmesg | grep 'irq=19' | wc -l
1970
$ dmesg | grep 'irq=19' | wc -l
1982
$ dmesg | grep 'irq=19' | wc -l
2004
$ dmesg | grep 'irq=19' | wc -l
2094
$ sudo rmmod mod_ser
```

И еще одна, коротко, *виртуальная* инсталляция — для подтверждения того, насколько по-разному можно ожидать отображения аппаратных ресурсов (не только в виртуальных, но и реальных конфигурациях), и того, насколько каждый экземпляр оборудования требует индивидуального подхода:

```
$ inxi -S
System:    Host: lmde32 Kernel: 5.10.0-16-686 i686 bits: 32 Desktop: Cinnamon 5.4.10 Distro:
LMDE 5 Elsie
$ lsusb -t
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=ohci-pci/12p, 12M
    |__ Port 1: Dev 2, If 0, Class=Human Interface Device, Driver=usbhid, 12M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci-pci/12p, 480M
$ cat /proc/interrupts | grep hci_
  9:      25098    XT-PIC      acpi, ehci_hcd:usb1, enp0s3
 10:      10260    XT-PIC      ohci_hcd:usb2, vmwgfx
$ sudo insmod mod_ser.ko irq=10 num=3
$ dmesg | grep 'irq=10' | tail -n6
[ 8347.134222] 20046582976683 : irq=10 - handler #1
[ 8347.134225] 20046582997098 : irq=10 - handler #2
[ 8347.134226] 20046583000350 : irq=10 - handler #3
[ 8347.221947] 20046793516431 : irq=10 - handler #1
[ 8347.221951] 20046793537367 : irq=10 - handler #2
[ 8347.221952] 20046793540983 : irq=10 - handler #3
$ dmesg | grep 'irq=10' | wc -l
741
$ dmesg | grep 'irq=10' | wc -l
825
$ dmesg | grep 'irq=10' | wc -l
885
$ sudo rmmod mod_ser
```

- Глава 6 -
Периферийные устройства в модулях ядра

Если знаешь, где искать, то найдешь скелет в любом шкафу.
Чак Паланик, «Бойцовский клуб»

Обслуживание проприетарных (которые вы создаете под свои цели) аппаратных расширений (для самых разнообразных целей) невозможно описать в общем виде — здесь вам предстоит работать в непосредственном и плотном контакте с разработчиком «железа», в постоянных консультациях, по каким портам ввода/вывода выполнять операции и с какой целью. Поэтому задачи непосредственно *организации обмена* данными в этой главе затрагиваются минимально (да их и невозможно рассмотреть в описании обозримого объема).

Мы рассмотрим только основные принципы учета и связывания периферийных устройств в системе, те вопросы, которые позволяют непосредственно выйти на порты и адреса, по которым уже далее нужно читать/писать для обеспечения функционирования устройства по его собственной алгоритмике. Другими словами, нас здесь интересует вопрос «как зацепиться за устройство на шине», а последующая организация работы по обмену с этим устройством — это уже на откуп вам совместно с вашим консультантом или разработчиком аппаратуры устройства.

Кроме того, работа с оборудованием в *коде модуля* ядра очень часто и сильно завязана с общими принципами, стандартами и тенденциями в работе собственно «железа» (стандарты шин) и инструментами *пространства пользователя* (такими как sysfs, udev, libusb, ...), степень задействования которых в последних версиях ядра все увеличивается. Это те вопросы, которые в публикациях по ядру не рассматриваются (из-за их «непринадлежности»), а посвященные им целевые публикации либо слишком перегружены ненужными деталями (по «железу»), либо никак не увязываются с процессами в ядре (для пользовательских проектов). Созданию минимальной связной картины взаимодействия этих компонентов здесь тоже будет специально уделено некоторое внимание.

Поддержка шинных устройств в модуле

Меня часто спрашивают в обсуждениях: «а устройство само создает имя в /dev?», «есть какие-то особенности в выборе номера `major` для USB-устройств?», «а как выражаются в коде обменные операции для USB-устройства?». Правильные ответы на

эти вопросы должны звучать соответственно так: «не создает», «для шинного устройства PCI или USB просто не создается номер `major`» и «никак не выражаются». И вот почему...

Дело в том, что *физические* устройства (в нашем представлении) на PCI или USB (платы, адаптеры, встроенные чипы, внешние устройства на USB и др.) в представлении Linux *логическими* устройствами не являются. Linux знает только три типа *устройств*: символьные устройства, блочные устройств и сетевые интерфейсы. А любое устройство PCI или USB может быть и первым (адаптеры синхронной связи E1/T1/J1), и вторым (флеш-диски, внешние HDD на USB), и третьим (все адаптеры Ethernet). На начальном этапе модуль не может знать, какое перед ним *логическое* устройство, и на этой *первой ступени* инициализации модуля ядро должно:

- соотнести идентификацию *физического* устройства (VID:PID) с поддерживаемым этим модулем набором устройств;
- извлечь непосредственно из устройства (например, из конфигурационной области устройства PCI) характерные его параметры: линию IRQ, адреса портов ввода/вывода для DMA (Direct Memory Access), характерные режимы обмена с концевыми точками (EP) устройства USB, ...;
- произвести *регистрацию* устройства, главным действием которой будет запись двух адресов функций обратного вызова для инициализации устройства, для активации устройства и его деактивации (функции `probe` и `remove` в структуре `struct pci_driver` для PCI-устройства, вызываемые при *загрузке и выгрузке модуля*, и функции `probe` и `disconnect` в структуре `struct usb_driver` для USB-устройства, вызываемые при «горячем» *подключении и отключении* USB-устройства).

На этом, собственно, первая ступень работы с устройством завершается. Следующая, *вторая ступень* инициализации производится уже изнутри кода функции `probe`, которая теперь будет вызвана обязательно в нужный момент. На этом этапе модуль:

- проделывает все необходимые инициализации: подключает устройство к линии IRQ, определяет функцию (функции) обработки прерываний и т. п.;
- но главное действие на этой ступени состоит в том, что модуль (и его автор), зная функциональное предназначение *физического* устройства (из техдокументации устройства), регистрирует соответствующее ему символьное или блочное устройство или сетевой интерфейс;
- это происходит в точности так же, как и с драйверами устройств, которые мы рассматривали ранее. Вот на этой ступени и выполняется создание имени устройства в /dev, если это необходимо, и назначение пары номеров `major` и `minor`;
- главным итогом этой ступени является определение всех операций ввода/вывода, требуемых таким устройством, — например, `read()`, `write()` и `ioctl()` для символьного устройства.

На этом фактически заканчивается вторая ступень инициализации устройства в драйвере. Все остальные действия описываются и реализуются на *третьей ступени — внутри обменных функций* (определенных на предыдущей ступени). Только

здесь вступают в игру технические спецификации самого устройства, алгоритмы его обмена, записи/чтения портов, организация работы DMA, обменные операции с концевыми точками (EP) USB-устройства. Это уже совершенно конкретные операции, полностью определяемые спецификациями устройства.

В любом случае работа над драйвером устройства начинается с детального анализа оборудования и формирования плана реализации последовательности операций.

Анализ оборудования

На первом шаге всякой разработки, касающейся непосредственно образца оборудования, производится уточнение того, как это оборудование видится со стороны системы, и соответствует ли это видение тому, как мы понимаем это оборудование. Целью анализа, обычно производимого предварительно, перед написанием модуля драйвера является уточнение специфических *численных параметров* тех или иных образцов оборудования и подготовка исходных данных. Общеизвестные команды для таких целей, это, например, `lspci` и `lsusb`, о которых мы будем говорить далее подробно и обстоятельно. Тем не менее, пока мы не подошли к их плотному использованию, они стоят хотя бы минимального отдельного упоминания...

Команда `lspci` перечисляет все устройства, распознанные на внутренней шине обмена PCI:

```
$ lspci
00:00.0 Host bridge: Intel Corporation Mobile 945GM/PM/GMS, 943/940GML and 945GT Express
Memory Controller Hub (rev 03)
00:02.0 VGA compatible controller: Intel Corporation Mobile 945GM/GMS, 943/940GML Express
Integrated Graphics Controller (rev 03)
00:02.1 Display controller: Intel Corporation Mobile 945GM/GMS/GME, 943/940GML Express
Integrated Graphics Controller (rev 03)
...
02:06.0 CardBus bridge: Texas Instruments PCIxx12 Cardbus Controller
02:06.2 Mass storage controller: Texas Instruments 5-in-1 Multimedia Card Reader (SD/MMC/MS/MS
PRO/xD)
02:06.3 SD Host controller: Texas Instruments PCIxx12 SDA Standard Compliant SD Host
Controller
02:06.4 Communication controller: Texas Instruments PCIxx12 GemCore based SmartCard controller
02:0e.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5788 Gigabit Ethernet (rev 03)
08:00.0 Network controller: Intel Corporation PRO/Wireless 3945ABG [Golan] Network Connection
(rev 02)
```

По каждому устройству указывается его производитель (например, Broadcom Corporation) и модель этого устройства в терминологии самого производителя (например, NetXtreme BCM5788 Gigabit Ethernet). Зачастую, при работе с драйвером нас будут интересовать не текстуальные описания производителя и модели, а их численные коды, что *для того же набора устройств* выглядит так:

```
$ lspci -n
00:00.0 0600: 8086:27a0 (rev 03)
```

```
00:02.0 0300: 8086:27a2 (rev 03)
00:02.1 0380: 8086:27a6 (rev 03)
...
02:06.0 0607: 104c:8039
02:06.2 0180: 104c:803b
02:06.3 0805: 104c:803c
02:06.4 0780: 104c:803d
02:0e.0 0200: 14e4:169c (rev 03)
08:00.0 0280: 8086:4222 (rev 02)
```

Первый из этих параметров (численный индекс производителя) называют VID (Vendor ID), а второй — PID (Product ID), или иногда DID (Device ID). Мы о них будем еще неоднократно говорить. Достаточно часто нужна диагностика устройств PCI по иерархии их подключения (что мы тоже детально будем обсуждать вскоре):

```
$ lspci -t
-[0000:00]-+-00.0
           +-02.0
           +-02.1
           +-1b.0
           +-1c.0-[08]----00.0
           +-1d.0
           +-1d.1
           +-1d.2
           +-1d.3
           +-1d.7
           +-1e.0-[02-06]--+-06.0
           |               +-06.2
           |               +-06.3
           |               +-06.4
           |               \-0e.0
           +-1f.0
           +-1f.1
           \-1f.2
```

Возможности утилиты `lspci` весьма обширны, и посмотреть их можно, выполнив команду в недозволенном синтаксисе, — например, так:

```
$ lspci --help
lspci: invalid option -- '-'
Usage: lspci [<switches>]
...
Resolving of device ID's to names:
-n              Show numeric ID's
-nn             Show both textual and numeric ID's (names & numbers)
-q              Query the PCI ID database for unknown ID's via DNS
-qq             As above, but re-query locally cached entries
-Q              Query the PCI ID database for all ID's via DNS
...
```

Здесь есть чрезвычайно замысловатые (и полезные!) опции — как, например, те, которые оставлены в части приведенного фрагмента вывода. По любому устройству мы можем запросить диагностику высокой (-v) или самой высокой (-vv) степени детализации:

```
$ lspci -d14e4:169c -v
02:0e.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5788 Gigabit Ethernet (rev 03)
        Subsystem: Hewlett-Packard Company Device 30aa
        Flags: bus master, 66MHz, medium devsel, latency 64, IRQ 16
        Memory at e8110000 (32-bit, non-prefetchable) [size=64K]
        Expansion ROM at <ignored> [disabled]
        Capabilities: <access denied>
        Kernel driver in use: tg3
```

Другая незаменимая команда, к которой мы будем неоднократно обращаться при анализе сменных устройств на линиях USB, — это lsusb:

```
$ lsusb
Bus 001 Device 002: ID 0424:2503 Standard Microsystems Corp. USB 2.0 Hub
Bus 001 Device 003: ID 1a40:0101 TERMINUS TECHNOLOGY INC. USB-2.0 4-Port HUB
Bus 001 Device 005: ID 046d:080f Logitech, Inc. Webcam C120
Bus 004 Device 002: ID 046d:c517 Logitech, Inc. LX710 Cordless Desktop Laser
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 006: ID 03f0:171d Hewlett-Packard Wireless (Bluetooth + WLAN) Interface
                                                                    [Integrated Module]
Bus 001 Device 007: ID 08ff:2580 AuthenTec, Inc. AES2501 Fingerprint Sensor
Bus 001 Device 009: ID 16d5:6502 AnyDATA Corporation CDMA/UMTS/GPRS modem
```

Команда lsusb во многом похожа по своей функциональности на упомянутую ранее lspci:

```
$ lsusb -t
/:  Bus 05.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
/:  Bus 04.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
    |__ Port 1: Dev 2, If 0, Class=HID, Driver=usbhid, 1.5M
    |__ Port 1: Dev 2, If 1, Class=HID, Driver=usbhid, 1.5M
/:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci_hcd/8p, 480M
    |__ Port 1: Dev 2, If 0, Class=hub, Driver=hub/3p, 480M
        |__ Port 1: Dev 6, If 0, Class='bInterfaceClass 0xe0 not yet handled', Driver=btusb, 12M
        |__ Port 1: Dev 6, If 1, Class='bInterfaceClass 0xe0 not yet handled', Driver=btusb, 12M
        |__ Port 1: Dev 6, If 2, Class=vend., Driver=, 12M
        |__ Port 1: Dev 6, If 3, Class=app., Driver=, 12M
        |__ Port 2: Dev 7, If 0, Class=vend., Driver=, 12M
```

```
    |__ Port 4: Dev 3, If 0, Class=hub, Driver=hub/4p, 480M
        |__ Port 4: Dev 9, If 0, Class=vend., Driver=option, 12M
        |__ Port 4: Dev 9, If 1, Class=vend., Driver=option, 12M
        |__ Port 4: Dev 9, If 2, Class=vend., Driver=option, 12M
        |__ Port 4: Dev 9, If 3, Class=stor., Driver=usb-storage, 12M
    |__ Port 6: Dev 5, If 0, Class='bInterfaceClass 0x0e not yet handled', Driver=uvcvideo, 480M
    |__ Port 6: Dev 5, If 1, Class='bInterfaceClass 0x0e not yet handled', Driver=uvcvideo, 480M
```

Разобраться в деталях показанных здесь выводов `lspci` и `lsusb` не так-то просто, но это и не входит в наши планы — эти выводы являются основным подспорьем при разработке аппаратных драйверов, и читать их детали в *конкретном окружении* становится естественно и просто...

Но для анализа всего установленного в системе оборудования (начиная с анализа изготовителя и состава BIOS) существует достаточно много команд «редкого применения», которые помнят только заматерелые системные администраторы и которые не всегда попадают в справочные руководства. Все такие команды в большинстве требуют прав root, а кроме того, некоторые из них могут присутствовать в ряде дистрибутивов Linux, но отсутствовать в других (и тогда их просто нужно доустановить с помощью менеджера программных пакетов). Информация от этих команд в какой-то мере дублирует друг друга, но только частично. Все такие команды в результате своего выполнения производят очень обширный объем вывода, поэтому его бессмысленно анализировать с экрана, а надо перенаправлять этот поток вывода в текстовый файл и использовать его в качестве журнала работы команды при последующем изучении.

Сбор информации об оборудовании может стать *ключевой позицией* при работе над драйверами периферийных устройств. Далее приводится только краткое описание (в порядке справки-напоминания) некоторых подобных команд (и несколько начальных строк их вывода — для идентификации того, что это именно та команда) — более детальное обсуждение увело бы нас слишком далеко от наших целей. Вот некоторые такие команды:

```
$ time sudo lshw > lshw.lst
real    0m5.545s
user    0m5.029s
sys     0m0.193s
$ cat > lshw.lst
notebook.localdomain
    description: Notebook
    product: HP Compaq nc6320 (ES527EA#ACB)
    vendor: Hewlett-Packard
...
```

> ***Примечание***
>
> Учтите, что показанная команда выполняется достаточно долго, и это не должно вас смущать.

Еще несколько полезных команд из той же группы:

```
$ lshal
Dumping 162 device(s) from the Global Device List:
-------------------------------------------------
udi = '/org/freedesktop/Hal/devices/computer'
  info.addons = {'hald-addon-acpi'} (string list)
...
$ sudo dmidecode
# dmidecode 2.10
SMBIOS 2.4 present.
23 structures occupying 1029 bytes.
Table at 0x000F38EB.
...
```

Последняя команда, например, в том числе дает детальную информацию о банках памяти и о том, какие модули оперативной памяти куда установлены. Интерфейсом к ней служит удобная команда `inxi`:

```
$ sudo inxi -MCDxxx -f
Machine:   Type: ARM Device System: Xunlong Orange Pi One details: Allwinner sun8i Family
           rev: N/A serial: 02c000815fd5e717
CPU:       Topology: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32 type: MCP arch: v7l
           rev: 5 bogomips: 0
           Speed: 1008 MHz min/max: 480/1008 MHz Core speeds (MHz): 1: 1008 2: 1008 3: 1008
           4: 1008
           Features: edsp evtstrm fastmult half idiva idivt lpae neon thumb tls vfp vfpd32 vfpv3
           vfpv4
Drives:    Local Storage: total: 14.84 GiB used: 6.32 GiB (42.6%)
           ID-1: /dev/mmcblk0 model: SB16G size: 14.84 GiB serial: 0x5baaa201 scheme: MBR
```

Для разработчиков *блочных* устройств представляет интерес пакет `smartctl` (предустановлен почти в любом дистрибутиве), выдающий детальную информацию по дисковому накопителю:

```
$ sudo smartctl -A /dev/sda
smartctl 5.39.1 2010-01-28 r3054 [i386-redhat-linux-gnu] (local build)
Copyright (C) 2002-10 by Bruce Allen, http://smartmontools.sourceforge.net

=== START OF READ SMART DATA SECTION ===
SMART Attributes Data Structure revision number: 16
Vendor Specific SMART Attributes with Thresholds:
ID# ATTRIBUTE_NAME          FLAG     VALUE WORST THRESH TYPE      UPDATED  WHEN_FAILED RAW_VALUE
  1 Raw_Read_Error_Rate     0x000f   100   100   046    Pre-fail  Always       -       49961
  2 Throughput_Performance  0x0005   100   100   030    Pre-fail  Offline      -       15335665
  3 Spin_Up_Time            0x0003   100   100   025    Pre-fail  Always       -       1
  4 Start_Stop_Count        0x0032   098   098   000    Old_age   Always       -       7320
...
```

Еще одна утилита, широко используемая при отработке блочных устройств (любых: будь то диск, USB-накопитель или RAM-диск) — `hdparm`:

```
$ sudo hdparm -i /dev/sda
/dev/sda:
 Model=WDC WD2500AAKX-001CA0, FwRev=15.01H15, SerialNo=WD-WMAYU0425651
 Config={ HardSect NotMFM HdSw>15uSec SpinMotCtl Fixed DTR>5Mbs FmtGapReq }
 RawCHS=16383/16/63, TrkSize=0, SectSize=0, ECCbytes=50
 BuffType=unknown, BuffSize=16384kB, MaxMultSect=16, MultSect=16
 CurCHS=16383/16/63, CurSects=16514064, LBA=yes, LBAsects=488397168
 IORDY=on/off, tPIO={min:120,w/IORDY:120}, tDMA={min:120,rec:120}
 PIO modes:  pio0 pio3 pio4
 DMA modes:  mdma0 mdma1 mdma2
 UDMA modes: udma0 udma1 udma2 udma3 udma4 udma5 *udma6
 AdvancedPM=no WriteCache=enabled
 Drive conforms to: Unspecified:  ATA/ATAPI-1,2,3,4,5,6,7
```

Подсистема *udev*

Подсистема udev — это подсистема создания именованных устройств, которая пришла на смену devfs (или даже статическому отображению устройств в каталоге /dev, доставшемуся раннему Linux от общей философии UNIX-систем). Более того, udev создает в /dev имена *динамически* (по мере подключения) и только для тех устройств, которые реально присутствуют в системе в текущий момент. Подсистема udev представляет собой надстройку пространства пользователя над файловой системой ядра /sys (которую мы достаточно детально рассмотрели ранее). Задачи ядра — определять изменения в аппаратной конфигурации системы (например, для устройств «горячего» подключения и USB), регистрировать эти изменения и вносить изменения в каталог /sys. Задача подсистемы udev — выполнить дальнейшую интеграцию и настройку такого устройства в системе (отобразить его в каталоге /dev) и предоставить пользователю уже готовое к работе устройство.

Подсистема udev, таким образом, это обширная надстройка пользовательского уровня, инструмент из области администрирования системы Linux, и *детальное* ее рассмотрение увело бы нас очень далеко от наших намерений. Но не упоминание udev — в контексте настройки устройств, используемых системой, — создало бы для работающего над драйвером специалиста ложную картину в еще большей мере. Поэтому мы совершим беглый экскурс в структуру udev в мере, достаточной программисту для настройки драйверов.

Асинхронные уведомления от ядра о любых изменениях в /sys посылаются ядром через дейтаграммный сокет `netlink`, спроектированный *специально* для этого случая вида обмена:

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

Получив такое уведомление от ядра, демон `udevd` (но это может делать и *любой*, хоть ваш собственный, процесс пространства пользователя) имеет возможность проанализировать дерево /sys (на основе *параметров*, полученных из содержимого дейтаграммного уведомления), а далее *динамически* (по событию) создать соответствующее имя в /dev. Всё достаточно просто. Более того, поскольку всё это (после

получения уведомления) происходит в пространстве пользователя (не в ядре), то при создании имени в /dev можно применить достаточно развитую систему *правил*, формируемых пользователем (в текстовых файлах) и определяющих характер создаваемого имени устройства.

Подсистема udev настраивает устройства в соответствии с заданными правилами. Правила эти содержатся в файлах каталогов /etc/udev/rules.d/ и /usr/lib/udev/rules.d, отдельные устанавливаемые проекты могут размещать файлы правил еще и в другие отдельные места. Все файлы правил просматриваются (отрабатываются) в *алфавитном* порядке (лексикографическом — вот почему традицией стало, для однозначности последовательности их применения, именовать такие файлы, начиная с численного префикса). Приведем возможное содержимое упомянутых каталогов:

```
$ ls -w100 /usr/lib/udev/rules.d
39-usbmuxd.rules                    73-special-net-names.rules
40-usb-media-players.rules          75-net-description.rules
40-usb_modeswitch.rules             75-probe_mtd.rules
40-vm-hotadd.rules                  77-mm-broadmobi-port-types.rules
50-firmware.rules                   77-mm-cinterion-port-types.rules
50-udev-default.rules               77-mm-dell-port-types.rules
55-dm.rules                         77-mm-dlink-port-types.rules
56-dm-parts.rules                   77-mm-ericsson-mbm.rules
56-hpmud.rules                      77-mm-fibocom-port-types.rules
56-lvm.rules                        77-mm-foxconn-port-types.rules
60-autosuspend-chromiumos.rules     77-mm-gosuncn-port-types.rules
...
$ ls -l /usr/lib/udev/rules.d | wc -l
126
```

Содержимое файлов правил текстовое и может быть достаточно разнообразным, но оно записывается исходя из весьма ограниченного набора синтаксических *правил действий для событий* в /sys:

- каждое правило записывается отдельной строкой (даже очень длинной), причем способы переноса строки не предусмотрены;
- если выполняются *все условия* (ключи соответствия, match key, записываются со знаком ==) возникшего события, то предпринимаются действия;
- действия, описанные в *правиле* (ключи назначения, assignment key, записываются со знаком =) предписывают выполнение некоторых операций: изменение флагов доступа (MODE="0600"), выполнение программы *до* создания имени устройства (PROGRAM="..."), запуск программы *после* создания имени устройства (RUN="...") и другие;
- все параметры (значения) в условиях и правилах заключаются в кавычки: ACTION=="remove";
- в записи условий и правил допускаются шаблоны: KERNEL=="controlD[0-9]*", NAME="dri/%k", ...

Все, что касается синтаксиса правил udev, исчерпывающе описано на справочной странице:

```
$ man 7 udev
UDEV(7)                                udev                                UDEV(7)
NAME
       udev - dynamic device management
...
```

Вот примеры нескольких реальных файлов правил (из разных систем):

```
$ cat 91-drm-modeset.rules
KERNEL=="controlD[0-9]*", NAME="dri/%k", MODE="0600"
$ cat 60-raw.rules
...
# An example would be:
ACTION=="add", KERNEL=="sda", RUN+="/bin/raw /dev/raw/raw1 %N"
...
```

Основной объем потребностей разработчика в информации по работе с udev обеспечивает не очень широко известная команда udevadm с огромным множеством параметров и опций:

```
$ udevadm info -q path -n sda
/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda
$ udevadm info -a -p $(udevadm info -q path -n sda)
...
looking at device '/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda':
    KERNEL=="sda"
    SUBSYSTEM=="block"
...
$ udevadm info -h
Usage: udevadm info OPTIONS
  --query=<type>             query device information:
      name                     name of device node
      symlink                  pointing to node
      path                     sys device path
      property                 the device properties
      all                      all values
  --path=<syspath>           sys device path used for query or attribute walk
  --name=<name>              node or symlink name used for query or attribute walk
...
```

Ключевой вопрос: как определить значения параметров, используемых в записи правил подключения и отключения конкретного устройства, для которого отрабатывается драйвер? Их можно увидеть при выполнении физического подключения/выключения этого устройства *при работающей программе* мониторинга уведомлений:

```
$ udevadm monitor --property --kernel
monitor will print the received events for:
KERNEL - the kernel uevent
```

```
KERNEL[27478.580340] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-4/1-4.4 (usb)
ACTION=add
BUSNUM=001
DEVNAME=/dev/bus/usb/001/035
DEVNUM=035
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-4/1-4.4
DEVTYPE=usb_device
MAJOR=189
MINOR=34
PRODUCT=1307/163/100
SEQNUM=3186
SUBSYSTEM=usb
TYPE=0/0/0

KERNEL[27478.580711] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-4/1-4.4/1-4.4:1.0 (usb)
ACTION=add
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-4/1-4.4/1-4.4:1.0
DEVTYPE=usb_interface
INTERFACE=8/6/80
MODALIAS=usb:v1307p0163d0100dc00dsc00dp00ic08isc06ip50
PRODUCT=1307/163/100
SEQNUM=3187
SUBSYSTEM=usb
TYPE=0/0/0
...
```

Здесь показан лишь начальный вывод, охватывающий только *два* асинхронных уведомления от ядра (для реальных устройств — например, USB — их может следовать значительно больше: 5, 10 и более — согласно фазам установки устройства). Представленные строки и отображают имена и значения параметров, посылаемых в уведомлении, например: `ACTION=add` — происходит *подключение* устройства (могло бы быть отключение), `MAJOR=189`, `MINOR=34` — старший и младший номера устройства в /dev, о которых мы уже так много знаем, `PRODUCT=1307/163/100` — индексы VID:PID для подключаемого устройства, о которых упоминалось ранее. Именно из этих *именованных* параметров и их значений конструируются те текстовые файлы параметров, которые вносятся в каталог /etc/udev/rules.d/ и которые *предписывают* выполнить те или иные действия при поступления уведомления от ядра с совпадающими параметрами.

В папке udev сопровождающего книгу файлового архива предлагается пример (мы не будем его подробно комментировать), не имеющий прямого отношения к коду модуля, но демонстрирующий, как *любой произвольный* пользовательский процесс может получать асинхронные уведомления о событиях ядра (приложение mondev в этой папке регистрирует уведомления о «горячих» подключениях/выключениях устройств, а приложение mondev — уведомления об изменениях в состояниях сетевых интерфейсов). Разбор таких приложений очень проясняет то, как драйверы устройств взаимодействуют с ядром.

Идентификация модуля

При установке в системе новых устройств часто бывает нужно идентифицировать то, каким модулем ядра будет поддерживаться устройство с заданной парой индексов VID:PID (это относится и к устройствам на шине PCI, и к USB-устройствам). Воспользуемся для демонстрации внешним USB Wi-Fi-адаптером (Wi-Fi-«свистком») модели Tenda W311M (рис. 6.1).

Рис. 6.1. Внешний USB Wi-Fi-адаптер модели Tenda W311M

Команда выполняется с опцией `-c`:

```
$ lsusb | grep Wireless
Bus 001 Device 012: ID 148f:5370 Ralink Technology, Corp. RT5370 Wireless Adapter
$ modprobe -c | grep usb: | grep -i 148f | grep -i 5370
alias usb:v148Fp5370d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
```

Здесь мы (в последнем поле строки) получили имя *модуля*, который будет поддерживать устройство `148f:5370`. Если для некоторого устройства неизвестен модуль поддержки, то вывод последней строки будет пустой. Естественно, подобным образом может быть получена информация по всем устройствам этого (или любого другого) производителя:

```
$ modprobe -c | grep usb: | grep -i 148f
alias usb:v148Fp1706d*dc*dsc*dp*ic*isc*ip*in* rt2500usb
alias usb:v148Fp2070d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp2570d*dc*dsc*dp*ic*isc*ip*in* rt2500usb
alias usb:v148Fp2573d*dc*dsc*dp*ic*isc*ip*in* rt73usb
alias usb:v148Fp2671d*dc*dsc*dp*ic*isc*ip*in* rt73usb
alias usb:v148Fp2770d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp2870d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3070d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3071d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3072d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3370d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3572d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp3573d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp5370d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp5372d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp5572d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp8070d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148Fp9020d*dc*dsc*dp*ic*isc*ip*in* rt2500usb
```

```
alias usb:v148Fp9021d*dc*dsc*dp*ic*isc*ip*in* rt73usb
alias usb:v148FpF101d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
alias usb:v148FpF301d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
```

Эту же информацию о модуле поддержки можно получить и по-другому:

```
# cat /lib/modules/`uname -r`/modules.alias | grep usb: | grep -i 148f | grep -i 5370
alias usb:v148Fp5370d*dc*dsc*dp*ic*isc*ip*in* rt2800usb
```

И самое интересное — то, как и откуда эта информация попадает в файл modules.alias:

```
$ modinfo rt2800usb
filename:       /lib/modules/3.13.6-
200.fc20.i686/kernel/drivers/net/wireless/rt2x00/rt2800usb.ko
license:        GPL
firmware:       rt2870.bin
description:    Ralink RT2800 USB Wireless LAN driver.
version:        2.3.0
author:         http://rt2x00.serialmonkey.com
srcversion:     D6F814DAF78F2BEA3DA12CB
alias:          usb:vF201p5370d*dc*dsc*dp*ic*isc*ip*in*
alias:          usb:v177Fp0254d*dc*dsc*dp*ic*isc*ip*in*
alias:          usb:v083ApF511d*dc*dsc*dp*ic*isc*ip*in*
...
alias:          usb:v148Fp5370d*dc*dsc*dp*ic*isc*ip*in*
...
alias:          usb:v07B8p2870d*dc*dsc*dp*ic*isc*ip*in*
depends:        rt2x00lib,rt2800lib,rt2x00usb
intree:         Y
vermagic:       3.13.6-200.fc20.i686 SMP mod_unload 686
signer:         Fedora kernel signing key
sig_key:        72:23:0F:69:91:87:54:91:2A:09:46:5F:53:D5:ED:EE:0C:C0:71:99
sig_hashalgo:   sha256
parm:           nohwcrypt:Disable hardware encryption. (bool)
```

Здесь примечательно то, что нам представлен драйвер чипсета Ralink 5370, а не какой-то конкретной модели изделия, и этот драйвер поддерживает весьма много (несколько сотен) самых различных моделей устройств от разных производителей, общее у которых то, что они собраны именно на таком чипсете:

```
$ modinfo rt2800usb | wc -l
338
```

Если ваше устройство идентифицировано модулем, то дальше такой модуль будет загружен (со всеми требуемыми зависимостями), и, например — в обсуждаемом случае, у вас появится новый сетевой интерфейс:

```
$ lsmod | grep rt2
rt2800usb              26581  0
rt2x00usb              19262  1 rt2800usb
```

```
rt2800lib              77341  1 rt2800usb
rt2x00lib              56555  3 rt2x00usb,rt2800lib,rt2800usb
crc_ccitt              12549  1 rt2800lib
mac80211              510326  5 iwl3945,iwlegacy,rt2x00lib,rt2x00usb,rt2800lib
cfg80211              400375  4 iwl3945,iwlegacy,mac80211,rt2x00lib
$ iwconfig
wlp8s0    IEEE 802.11bg  ESSID:off/any
          Mode:Managed  Access Point: Not-Associated   Tx-Power=off
          Retry  long limit:7   RTS thr:off   Fragment thr:off
          Power Management:off
lo        no wireless extensions.
enp2s14   no wireless extensions.
wlp0s29f7u4  IEEE 802.11bgn  ESSID:off/any
          Mode:Managed  Access Point: Not-Associated   Tx-Power=0 dBm
          Retry  long limit:7   RTS thr:off   Fragment thr:off
          Power Management:on
```

Последний из показанных и есть обсуждаемый сетевой интерфейс.

Ошибки идентификации модуля

Показанный в предыдущем разделе процесс идентификации весьма продуктивен, особенно для интеграции новых, недавно появившихся моделей устройств. Но номенклатура доступных устройств постоянно расширяется, и не исключено, что ваше новое устройство получит поддержку *неадекватного* модуля, что не обеспечит его работоспособность. Образцом сказанного может быть сетевой адаптер Broadcom BCM43228, который активно используется в ноутбуках HP (и порождает много недоумения и вопросов в Интернете):

```
$ lspci | grep Broad
24:00.0 Network controller: Broadcom Corporation BCM43228 802.11a/b/g/n
$ lspci -n | grep 24:00.0
24:00.0 0280: 14e4:4359
```

После инсталляции системы (Fedora) для такого сетевого адаптера иногда устанавливается поддержка модулем `bcma` — общим модулем поддержки адаптеров 43*xxx* серии:

```
$ modprobe -c | grep -i 14e4 | grep 4359
alias pci:v000014E4d00004359sv*sd*bc*sc*i* bcma
$ lsmod | grep bcma
bcma 46142 0
```

Но отправившись на сайт производителя (Broadcom), мы узнаем, что модуль `bcma` поддерживает адаптеры 43*xxx* серии вплоть до 43227, а для поддержки 43228 нужно использовать *закрытый* модуль `wl`. К счастью, в большинстве случаев модули новых устройств очень быстро попадают в репозитории дистрибутивов, откуда их можно и устанавливать:

```
$ sudo yum install kmod-wl
...
Установлено:
 kmod-wl.x86_64 0:6.30.223.141-5.fc20.14.....................................
Установлены зависимости:
 broadcom-wl.noarch 0:6.30.223.141-2.fc20.....................................
 kmod-wl-3.13.6-200.fc20.x86_64.x86_64 0:6.30.223.141-5.fc20.14..............
Выполнено!
$ sudo rmmod bcma
```

Записываем выгруженный модуль в «черный список» и проверяем использование его альтернативы:

```
# echo "blacklist bcma" >> /etc/modprobe.d/blacklist.conf
# modprobe wl
```

И после перезагрузки получаем совершенно иную конфигурацию поддержки Wi-Fi (работоспособный вариант):

```
$ lsmod | grep wl
wl                   4207671  0
cfg80211              513095  1 wl
lib80211               13968  2 wl,lib80211_crypt_tkip
$ iwconfig
lo        no wireless extensions.
em1       no wireless extensions.
wlo1      IEEE 802.11abg  ESSID:"ZTE"
          Mode:Managed  Frequency:2.462 GHz  Access Point: C8:64:C7:8A:50:16
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Power Management:off
```

Устройства на шине PCI

Архитектура шины PCI (Peripheral Component Interconnect) была разработана в качестве замены предыдущему стандарту ISA/EISA (Industry Standard Architecture) с тремя основными целями: а) получить лучшую производительность при передаче данных между компьютером и его периферией; б) быть независимой от платформы, насколько это возможно, и в) упростить добавление и удаление периферийных устройств в системе. Первоначальный стандарт PCI описывал параллельный обмен 32-битовыми данными на частоте 33 или 66 МГц, обеспечивая пиковую производительность 266 MB/s (мегабит в секунду). Следующее расширение, известное как PCI Extended (PCI-X), определяло шину до 64-битной, частоту до 133 МГц и производительность в 1 GB/s. Стандарт PCI Express (PCIe, или PCI-E) представляет семейство нового поколения. В отличие от PCI, PCIe использует последовательный протокол передачи данных. При этом PCIe поддерживает максимально 32 последовательные линии (links), каждая из которых (в стандарте версии 1.1) поддерживает поток 250 MB/s в каждом направлении передачи, обеспечивая таким образом производительность до 8 GB/s в каждом направлении. Стандарт PCIe 2.0 предусматривает еще большие скорости передачи.

Примечание

Последовательные каналы передачи, в отличие от параллельных шин, что предполагалось на ранних периодах развития компьютерных технологий, обеспечивают более высокие скорости и устойчивость обмена за счет отсутствия эффекта интерференции сигнала (рассинхронизации). Поэтому переход к последовательным протоколам обмена стал общей тенденцией стандартизации, примеры чему: PCIe, SATA, USB, FireWire...

Стандарты PCI, помимо отмеченных вариантов, имеют еще варианты, связанные с мобильными применениями: CardBus, Mini PCI, PCI Express Mini Card, Express Card (которые существенно принципиально от базовых вариантов не отличаются). В настоящее время PCI широко используется на самых разных процессорных платформах: IA-32/IA-64, Alpha, PowerPC, SPARC64 и пр.

Для разработчика драйверной поддержки PCI-устройств все это разнообразие стандартов не имеет особого значения (может, разве что, изменяться размер конфигурационной области, о чем будет сказано далее). Поэтому, мы в контексте нашего обсуждения больше не будем учитывать различия PCI-шин.

Самой актуальной для автора драйвера является поддержка автоопределения интерфейса PCI-плат: PCI-устройства настраиваются автоматически во время загрузки — это делается программами BSP (Board Support Programm, поддержки аппаратной платформы). В случае *x*86-компьютера универсального назначения функцию такой поддержки выполняют программы BIOS. Затем драйвер устройства получает доступ к информации о конфигурации устройства и производит инициализацию. Это происходит без необходимости совершать какое-либо тестирование периода выполнения (как, например, в стандарте PnP для устройств ISA).

Каждое периферийное устройство PCI адресуется по подключению следующими физическими параметрами: номером шины, номером устройства и номером 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-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 эквивалентных (по своим возможностям) функций (хорошим примером являются 2, 4 или 8 независимых интерфейсов E1/T1/J1 на PCI-платах основных мировых производителей: Digium, Sangoma и других).

Адресацию PCI-устройств в своей Linux-системе смотрим так:

```
$ lspci
00:00.0 Host bridge: Intel Corporation Mobile 945GM/PM/GMS, 943/940GML and 945GT Express
Memory Controller Hub (rev 03)
```

```
00:02.0 VGA compatible controller: Intel Corporation Mobile 945GM/GMS, 943/940GML Express
Integrated Graphics Controller (rev 03)
00:02.1 Display controller: Intel Corporation Mobile 945GM/GMS/GME, 943/940GML Express
Integrated Graphics Controller (rev 03)
00:1b.0 Audio device: Intel Corporation 82801G (ICH7 Family) High Definition Audio Controller
(rev 01)
00:1c.0 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 1 (rev 01)
00:1c.2 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 3 (rev 01)
00:1c.3 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 4 (rev 01)
00:1d.0 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #1 (rev 01)
00:1d.1 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #2 (rev 01)
00:1d.2 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #3 (rev 01)
00:1d.3 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #4 (rev 01)
00:1d.7 USB Controller: Intel Corporation 82801G (ICH7 Family) USB2 EHCI Controller (rev 01)
00:1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev e1)
00:1f.0 ISA bridge: Intel Corporation 82801GBM (ICH7-M) LPC Interface Bridge (rev 01)
00:1f.2 IDE interface: Intel Corporation 82801GBM/GHM (ICH7 Family) SATA IDE Controller (rev 01)
02:06.0 CardBus bridge: Texas Instruments PCIxx12 Cardbus Controller
02:06.1 FireWire (IEEE 1394): Texas Instruments PCIxx12 OHCI Compliant IEEE 1394 Host Controller
02:06.2 Mass storage controller: Texas Instruments 5-in-1 Multimedia Card Reader (SD/MMC/MS/
                                                                                MS PRO/xD)
02:06.3 SD Host controller: Texas Instruments PCIxx12 SDA Standard Compliant SD Host Controller
02:06.4 Communication controller: Texas Instruments PCIxx12 GemCore based SmartCard controller
02:0e.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5788 Gigabit Ethernet (rev 03)
08:00.0 Network controller: Intel Corporation PRO/Wireless 3945ABG [Golan] Network Connection
(rev 02)
```

Особенно полезным может оказаться использование уточняющих (расширяющих) опций команды `lspci`, например, такой:

```
$ lspci -vk
...
00:1b.0 Audio device: Intel Corporation 82801G (ICH7 Family) High Definition Audio Controller
(rev 01)
        Subsystem: Hewlett-Packard Company Device 30aa
        Flags: bus master, fast devsel, latency 0, IRQ 21
        Memory at e8580000 (64-bit, non-prefetchable) [size=16K]
        Capabilities: <access denied>
        Kernel driver in use: HDA Intel
        Kernel modules: snd-hda-intel
...
00:1f.2 IDE interface: Intel Corporation 82801GBM/GHM (ICH7 Family) SATA IDE Controller
(rev 01) (prog-if 80 [Master])
        Subsystem: Hewlett-Packard Company Device 30aa
        Flags: bus master, 66MHz, medium devsel, latency 0, IRQ 17
        I/O ports at 01f0 [size=8]
        I/O ports at 03f4 [size=1]
        I/O ports at 0170 [size=8]
        I/O ports at 0374 [size=1]
```

```
        I/O ports at 60a0 [size=16]
        Capabilities: <access denied>
        Kernel driver in use: ata_piix

02:06.0 CardBus bridge: Texas Instruments PCIxx12 Cardbus Controller
        Subsystem: Hewlett-Packard Company Device 30aa
        Flags: bus master, medium devsel, latency 168, IRQ 18
        Memory at e8100000 (32-bit, non-prefetchable) [size=4K]
        Bus: primary=02, secondary=03, subordinate=06, sec-latency=176
        Memory window 0: 80000000-83fff000 (prefetchable)
        Memory window 1: 88000000-8bfff000
        I/O window 0: 00003000-000030ff
        I/O window 1: 00003400-000034ff
        16-bit legacy interface ports at 0001
        Kernel driver in use: yenta_cardbus
        Kernel modules: yenta_socket
...
```

Здесь мы информацию по тем же устройствам получаем в развернутом виде (а поэтому ее часто избыточно много), здесь же представлены и технические параметры устройств (порты ввода/вывода, линии IRQ), и имена поддерживающих устройства *модулей ядра* (драйверов).

Различие между понятиями *устройства* и *функции* PCI хорошо заметно на только что приведенном примере многофункционального устройства производителя Texas Instruments с номером устройства 6 на шине 2, которое представляет собой объединение пяти функций: 0...4. Например, функция `02:06.3` представляет собой оборудование чтения SD-карт и поддерживается отдельным модулем ядра, создающим соответствующие имена устройств вида:

```
$ ls /dev/mm*
/dev/mmcblk0  /dev/mmcblk0p1
```

Другое представление той же адресной информации (тот же хост, та же конфигурация) можем получить так:

```
$ tree /sys/bus/pci/devices/
/sys/bus/pci/devices/
├── 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
├── 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:02.0
├── 0000:00:02.1 -> ../../../devices/pci0000:00/0000:00:02.1
├── 0000:00:1b.0 -> ../../../devices/pci0000:00/0000:00:1b.0
├── 0000:00:1c.0 -> ../../../devices/pci0000:00/0000:00:1c.0
├── 0000:00:1c.2 -> ../../../devices/pci0000:00/0000:00:1c.2
├── 0000:00:1c.3 -> ../../../devices/pci0000:00/0000:00:1c.3
├── 0000:00:1d.0 -> ../../../devices/pci0000:00/0000:00:1d.0
├── 0000:00:1d.1 -> ../../../devices/pci0000:00/0000:00:1d.1
├── 0000:00:1d.2 -> ../../../devices/pci0000:00/0000:00:1d.2
├── 0000:00:1d.3 -> ../../../devices/pci0000:00/0000:00:1d.3
├── 0000:00:1d.7 -> ../../../devices/pci0000:00/0000:00:1d.7
```

```
├── 0000:00:1e.0 -> ../../../devices/pci0000:00/0000:00:1e.0
├── 0000:00:1f.0 -> ../../../devices/pci0000:00/0000:00:1f.0
├── 0000:00:1f.2 -> ../../../devices/pci0000:00/0000:00:1f.2
├── 0000:02:06.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:06.0
├── 0000:02:06.1 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:06.1
├── 0000:02:06.2 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:06.2
├── 0000:02:06.3 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:06.3
├── 0000:02:06.4 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:06.4
├── 0000:02:0e.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:0e.0
└── 0000:08:00.0 -> ../../../devices/pci0000:00/0000:00:1c.0/0000:08:00.0
```

Здесь отчетливо видны (слева) поля — например, для контроллера VGA это `0000:00:02.0` — выделены домен (16 битов), шина (8 битов), устройство (5 битов) и функция (3 бита). Поэтому, когда мы говорим о конкретном устройстве, поддерживаемом модулем (см. далее), мы часто имеем в виду полный набор: номер домена + номер шины + номер устройства + номер функции.

С другой стороны[1], каждое устройство по типу идентифицируется двумя индексами: индексом производителя (Vendor ID) и индексом типа устройства (Device ID). Эта пара однозначно идентифицирует тип устройства. Использование двух 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 Интернета, — например, по адресу **http://pciids.sourceforge.net/** («The PCI ID Repository»). Эти два параметра являются уникальным (среди всех устройств в мире) ключом поиска устройств, установленных на шине PCI. Идентификация устройства парой Vendor ID : Device ID — это константа, неизменно закрепленная за устройством. Адресная же идентификация устройства (шина : устройство : функция) — величина, изменяющаяся в зависимости от того, в какой конфигурации компьютера (в каком экземпляре компьютера) используется устройство, в какой PCI-слот установлено устройство, и даже от того, какие другие устройства, помимо интересующего нас, устанавливаются или извлекаются из компьютера. Однозначно связать идентификацию VID:DID с адресной идентификацией устройства — это одна из первейших задач модуля-драйвера.

Поиск устройств, установленных на шине PCI, в программном коде модуля делается *циклическим перебором* (перечислением, enumeration) всех установленных устройств по определенным критериям поиска. Для поиска (перебора устройств, установленных на шине PCI) в программном коде модуля в цикле используется итератор:

```
struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device, struct pci_dev *from);
```

Здесь *from* — это `NULL` при начале поиска (или возобновлении поиска с начала) или указатель устройства, найденного на предыдущем шаге поиска. Если в качестве

[1] Нужно четко различать *адресацию* и *идентификацию* PCI-устройства. При перестановке устройства в другой разъем PCI его адресация изменится, но идентификация является константным параметром этого устройства.

Vendor ID и/или Device ID указана константа с символьным именем `PCI_ANY_ID=-1`, то предполагается перебор всех доступных устройств с таким идентификатором. Если искомое устройство не найдено (или больше таких устройств не находятся в цикле), то очередной вызов возвратит `NULL`. Если возвращаемое значение не `NULL`, то возвращается указатель структуры, описывающей устройство, и счетчик использования для устройства инкрементируется. Когда устройство удаляется (модуль выгружается) для декремента этого счетчика использования необходимо вызвать:

```
void pci_dev_put(struct pci_dev *dev);
```

> ***Примечание***
>
> Эта процедура весьма напоминает использование POSIX API в пространстве пользователя для работы с именами, входящими в тот или иной каталог: все имена перебираются последовательно, пока результат очередного перебора не станет `NULL`.

После нахождения устройства, но прежде начала его использования, необходимо разрешить использование устройства вызовом: `pci_enable_device(struct pci_dev *dev)` — часто это выполняется в функции инициализации устройства: поле `probe` структуры `struct pci_driver` (см. далее), но может выполняться и автономно в коде драйвера.

Каждое найденное устройство имеет свое пространство конфигурации, значения которого заполнены программами BIOS (или PnP OS, или программами BSP) — важно, что на момент загрузки модуля это конфигурационное пространство всегда заполнено и может только читаться (не записываться). Пространство конфигурации PCI-устройства состоит из 256 байтов для каждой функции устройства (для устройств PCI Express расширено до 4 Кбайт конфигурационного пространства для каждой функции) и стандартизированной схемы 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 свое устройство. Вот — для сравнения — начальные строки вывода команды для того же хоста (видим здесь — 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 байта конфигурационной области стандартизованы, остальные зависят от устройства. Самыми актуальными для нас являются (кроме ID, описанных ранее) поля по смещению:

```
0x10 — Base Address 0
0x14 — Base Address 1
0x18 — Base Address 2
```

```
0x1C – Base Address 3
0x20 – Base Address 4
0x24 – Base Address 5
0x3C – IRQ Line
0x3D – IRQ Pin
```

Вся регистрация устройства PCI и связывание его параметров с кодом модуля должны происходить *исключительно* через значения, считанные из конфигурационного пространства устройства. Обработку конфигурационной информации показывает модуль (см. папку pci в сопровождающем книгу файловом архиве) lab2_pci.ko (заимствовано из [6]):

lab2_pci.c

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

static int __init my_init(void) {
   u16 dval;
   char byte;
   int j = 0;
   struct pci_dev *pdev = NULL;
   printk(KERN_INFO "LOADING THE PCI_DEVICE_FINDER\n");
   /* either of the following looping constructs will work */
   for_each_pci_dev(pdev) {
      /*    while ((pdev = pci_get_device
                    (PCI_ANY_ID, PCI_ANY_ID, pdev))) { */
      printk(KERN_INFO "\nFOUND PCI DEVICE # j = %d, ", j++);
      printk(KERN_INFO "READING CONFIGURATION REGISTER:\n");
      printk(KERN_INFO "Bus,Device,Function=%s", pci_name(pdev));
      pci_read_config_word(pdev, PCI_VENDOR_ID, &dval);
      printk(KERN_INFO " PCI_VENDOR_ID=%x", dval);
      pci_read_config_word(pdev, PCI_DEVICE_ID, &dval);
      printk(KERN_INFO " PCI_DEVICE_ID=%x", dval);
      pci_read_config_byte(pdev, PCI_REVISION_ID, &byte);
      printk(KERN_INFO " PCI_REVISION_ID=%d", byte);
      pci_read_config_byte(pdev, PCI_INTERRUPT_LINE, &byte);
      printk(KERN_INFO " PCI_INTERRUPT_LINE=%d", byte);
      pci_read_config_byte(pdev, PCI_LATENCY_TIMER, &byte);
      printk(KERN_INFO " PCI_LATENCY_TIMER=%d", byte);
      pci_read_config_word(pdev, PCI_COMMAND, &dval);
      printk(KERN_INFO " PCI_COMMAND=%d\n", dval);
      /* decrement the reference count and release */
      pci_dev_put(pdev);
   }
```

```
   return 0;
}

static void __exit my_exit(void) {
   printk(KERN_INFO "UNLOADING THE PCI DEVICE FINDER\n");
}

module_init(my_init);
module_exit(my_exit);

MODULE_AUTHOR("Jerry Cooperstein");
MODULE_DESCRIPTION("LDD:1.0 s_22/lab2_pci.c");
MODULE_LICENSE("GPL v2");
```

Рассмотрение кода этого примера позволяет сформулировать ряд полезных утверждений:

- поскольку операция перечисления устройств PCI производится часто, то для ее записи сконструирован специальный макрос `for_each_pci_dev()`. Следом за ним в коде комментарием показано его раскрытие в виде явного цикла;
- конфигурационные параметры никогда не читаются напрямую, по их смещениям — для этого существуют (определены в <linux/pci.h>) инлайново заданные вызовы вида `pci_read_config_byte()`, `pci_read_config_word()` и подобные им;
- конкретный вид считываемого конфигурационного параметра задается символьными константами вида `PCI_*` (определены там же), указываемыми как второй параметр макроса;
- результат (значение конфигурационного параметра) возвращается в виде побочного эффекта в третий параметр макроса.

И теперь мы можем рассмотреть небольшой начальный фрагмент результата выполнения приведенного ранее модуля (весь результат может быть очень объемным):

```
$ inxi -MCxxx
Machine:
  Type: Desktop System: Hewlett-Packard product: HP t520 Flexible Series TC
    v: N/A serial: <superuser required> Chassis: type: 4
    serial: <superuser required>
  Mobo: Hewlett-Packard model: 21EF v: 00~ serial: <superuser required>
    UEFI: AMI v: L41 v01.09 date: 10/19/2016
CPU:
  Info: dual core model: AMD GX-212JC SOC with Radeon R2E Graphics bits: 64
    type: MCP smt: <unsupported> arch: Puma rev: 1 cache: L1: 128 KiB
    L2: 1024 KiB
  Speed (MHz): avg: 1198 min/max: N/A cores: 1: 1198 2: 1198 bogomips: 4790
  Flags: avx ht lm nx pae sse sse2 sse3 sse4_1 sse4_2 sse4a ssse3 svm
```

```
$ sudo insmod lab2_pci.ko
$ lspci
00:00.0 Host bridge: Advanced Micro Devices, Inc. [AMD] Family 16h (Models 30h-3fh) Processor
Root Complex
00:01.0 VGA compatible controller: Advanced Micro Devices, Inc. [AMD/ATI] Mullins [Radeon
R1E/R2E Graphics] (rev 01)
00:01.1 Audio device: Advanced Micro Devices, Inc. [AMD/ATI] Kabini HDMI/DP Audio
00:02.0 Host bridge: Advanced Micro Devices, Inc. [AMD] Family 16h (Models 30h-3fh) Host Bridge
00:02.3 PCI bridge: Advanced Micro Devices, Inc. [AMD] Family 16h Processor Functions 5:1
00:08.0 Encryption controller: Advanced Micro Devices, Inc. [AMD] Kabini/Mullins PSP-Platform
Security Processor
00:10.0 USB controller: Advanced Micro Devices, Inc. [AMD] FCH USB XHCI Controller (rev 11)
00:11.0 SATA controller: Advanced Micro Devices, Inc. [AMD] FCH SATA Controller [IDE mode]
(rev 40)
00:12.0 USB controller: Advanced Micro Devices, Inc. [AMD] FCH USB EHCI Controller (rev 39)
00:13.0 USB controller: Advanced Micro Devices, Inc. [AMD] FCH USB EHCI Controller (rev 39)
00:14.0 SMBus: Advanced Micro Devices, Inc. [AMD] FCH SMBus Controller (rev 42)
00:14.2 Audio device: Advanced Micro Devices, Inc. [AMD] FCH Azalia Controller (rev 02)
00:14.3 ISA bridge: Advanced Micro Devices, Inc. [AMD] FCH LPC Bridge (rev 11)
00:18.0 Host bridge: Advanced Micro Devices, Inc. [AMD] Family 16h (Models 30h-3fh) Processor
Function 0
00:18.1 Host bridge: Advanced Micro Devices, Inc. [AMD] Family 16h (Models 30h-3fh) Processor
Function 1
00:18.2 Host bridge: Advanced Micro Devices, Inc. [AMD] Family 16h (Models 30h-3fh) Processor
Function 2
00:18.3 Host bridge: Advanced Micro Devices, Inc. [AMD] Family 16h (Models 30h-3fh) Processor
Function 3
00:18.4 Host bridge: Advanced Micro Devices, Inc. [AMD] Family 16h (Models 30h-3fh) Processor
Function 4
00:18.5 Host bridge: Advanced Micro Devices, Inc. [AMD] Family 16h (Models 30h-3fh) Processor
Function 5
01:00.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL8111/8168/8411 PCI Express
Gigabit Ethernet Controller (rev 0c)
$ dmesg | tail -n202
[16658.443324] lab2_pci: loading out-of-tree module taints kernel.
[16658.443547] LOADING THE PCI_DEVICE_FINDER
[16658.443554]
               FOUND PCI DEVICE # j = 0,
[16658.443555] READING CONFIGURATION REGISTER:
[16658.443557] Bus,Device,Function=0000:00:00.0
[16658.443562]  PCI_VENDOR_ID=1022
[16658.443564]  PCI_DEVICE_ID=1566
[16658.443566]  PCI_REVISION_ID=0
[16658.443568]  PCI_INTERRUPT_LINE=0
[16658.443570]  PCI_LATENCY_TIMER=0
[16658.443572]  PCI_COMMAND=4
...
$ sudo rmmod lab2_pci
```

К этому моменту рассмотрения мы разобрались, хотелось бы надеяться, с тем, как перечисляются PCI-устройства в системе и как извлекаются их параметры из области конфигурации. Теперь нас должен интересовать вопрос: как это использовать в коде своего собственного модуля? Общий скелет любого модуля, реализующего драйвер PCI-устройства, всегда практически однотипен...

Для использования некоторой группы устройств PCI код модуля определяет массив (таблицу) описания устройств, обслуживаемых этим модулем. Каждому новому устройству в этом списке соответствует новый элемент. Последний элемент массива всегда нулевой — это и есть признак завершения списка устройств. Строки такого массива заполняются макросом `PCI_DEVICE`:

```
static struct pci_device_id i810_ids[] = {
   { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG1) },
   { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810_IG3) },
   { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82810E_IG) },
   { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82815_CGC) },
   { PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82845G_IG) },
   { 0, },
};
```

Очень часто такой массив будет содержать всего два элемента: элемент, описывающий единичное устройство-функцию, поддерживаемую модулем, и завершающий нулевой терминатор.

Созданная структура `struct pci_device_id` должна быть экспортирована в пользовательское пространство, чтобы позволить системам «горячего» подключения и загрузки модулей (sysfs, udev и т. д.) знать, с какими устройствами работает наш модуль. Эту задачу решает макрос `MODULE_DEVICE_TABLE`:

```
MODULE_DEVICE_TABLE(pci, i810_ids);
```

Кроме доступа к области конфигурационных параметров, программный код должен получить доступ к областям ввода/вывода и регионов памяти, ассоциированных с PCI-устройством. Таких областей ввода/вывода может быть до шести (см. ранее формат области конфигурационных параметров) — 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 и есть индекс региона: 0...5.
- первые два вызова возвращают начальный и конечный адреса 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 неко-

торых архитектур — *x*86 из числа таких — 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 кешироваться процессором, и доступ к ним не может быть оптимизирован. Доступ к памяти таких областей должен быть отмечен как «без упреждающей выборки». Все, что относится к отображению памяти, будет рассмотрено отдельно далее, в следующем разделе.

- флаги PCI региона (`pci_resource_flags()`) определены в <linux/ioport.h>, вот некоторые из них:
  - `IORESOURCE_IO`, `IORESOURCE_MEM` — только один из этих флагов может быть установлен. Он указывает, относятся ли адреса к пространству ввода/вывода или к пространству памяти (в архитектурах, отображающих ввод/вывод на память);
  - `IORESOURCE_PREFETCH` — определяет, допустима ли для региона упреждающая выборка;
  - `IORESOURCE_READONLY` — определяет, является ли регион памяти защищенным от записи.

Назначение любого из шести регионов ввода/вывода и их число — совершенно специфично для каждого типа устройства. Например, согласно спецификации на сетевой адаптер RTL8139C, его MAC-адрес занимает первые 6 байтов в пространстве портов нулевого региона ввода/вывода, отведенного адаптеру.

Основной структурой, которую должны создать все драйверы PCI, 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 и 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 (области ввода/вывода или прерыванию) соответствующего PCI-устройства, драйвер должен как минимум вызвать функцию:

  ```
  int pci_enable_device(struct pci_dev *dev);
  ```

- `remove` — функция обратного вызова при удалении устройства;
- `suspend` — функция менеджера энергосохранения, вызываемая, когда устройство уходит в пассивное состояние (засыпает);
- `resume` — функция менеджера энергосохранения, вызываемая когда устройство пробуждается;
- ... и другие функции обратного вызова.

Обычно для создания правильной структуры `struct pci_driver` достаточно бывает определить как минимум поля:

```
static struct pci_driver own_driver = {
   .name = "mod_skel",
   .id_table = i810_ids,
   .probe = probe,
   .remove = remove,
};
```

Теперь устройство может быть зарегистрировано в ядре:

```
int pci_register_driver(struct pci_driver *dev);
```

Этот вызов возвращает 0, если регистрация устройства прошла успешно. При завершении (выгрузке) модуля выполняется обратная операция:

```
void pci_unregister_driver(struct pci_driver *dev);
```

К этому моменту рассмотрения у нас есть вся информация, для того чтобы начинать запись специфических операций ввода/вывода для нашего устройства. Некоторых дополнительных замечаний заслуживает регистрация обработчика прерываний от устройства, но это уже будет, скорее, повторение того материала, который мы рассматривали ранее...

Подключение к линии прерывания

Установка обработчиков прерываний и их написание, как только что было отмечено, рассматривались ранее. Здесь мы останавливаемся только на той детали этого процесса, согласно которой при установке обработчика прерывания для устройства необходимо указывать используемую им линию IRQ :

```
typedef irqreturn_t (*irq_handler_t)(int, void*);
int request_irq(unsigned int irq, irq_handler_t handler, ...);
```

В устройствах шины ISA в поле первого параметра указывалось фиксированное значение (номер линии IRQ), устанавливаемое механически на плате устройства

(переключателями/джамперами) или записываемое конфигурационными программами в EPROM устройства. В устройствах PnP ISA предпринимались попытки проб и тестирования различных линий IRQ на принадлежность к устройству. В нынешних PCI-устройствах это значение извлекается из области конфигурационных параметров устройства (смещение `0x3C`), но делается это не непосредственно, а через API ядра из структуры `struct pci_dev`. И тогда весь процесс регистрации, который очень часто записывается в теле функции `probe`, о которой говорилось ранее, выглядит, 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 ранее.

Отображение памяти

Ранее уже были показаны адреса из адресных регионов устройства PCI, возвращаемые вызовами PCI API:

```
unsigned long pci_resource_start(struct pci_dev *dev, int bar);
unsigned long pci_resource_end(struct pci_dev *dev, int bar);
```

Результаты функции — это адреса шины, которые (в зависимости от архитектуры) необходимо преобразовать в виртуальные (логические) адреса, с которыми оперирует код адресных пространств и ядра, и пользователя:

```
#include <asm/io.h>
unsigned long virt_to_bus(volatile void *address);
void *bus_to_virt(unsigned long address);
unsigned long virt_to_phys(volatile void *address);
void *phys_to_virt(unsigned long address);
```

> ***Пояснение***
>
> Для архитектуры x86 физический адрес (`phys`) и адрес шины (`bus`) — это одно и то же, однако не факт, что это имеет место и для других архитектур.

Большинство PCI-устройств отображают свои управляющие регистры на адреса памяти, и высокопроизводительные приложения предпочитают иметь прямой доступ к таким регистрам — вместо того, чтобы постоянно вызывать `ioctl()` для выполнения этой работы. Отображение устройства означает связывание диапазона адресов пользовательского пространства с памятью устройства. Всякий раз, когда программа читает или записывает в заданном диапазоне адресов, она на самом деле обращается к устройству. Существенным ограничением отображения памяти (`mmap()`) является то, что ядро может управлять виртуальными адресами только на

уровне таблиц страниц, — таким образом, отображаемая область должна быть кратной размеру страницы RAM (`PAGE_SIZE`) и должна находиться в физической памяти, начиная с адреса, который кратен `PAGE_SIZE`. Если рассмотреть адрес памяти (виртуальный или физический), можно увидеть, что он делится на номер страницы и смещение внутри этой страницы. Например, если используются страницы по 4096 байтов, 12 младших значащих битов 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 распространенной операцией — существующий макрос `PAGE_SHIFT` сообщает, на сколько битов в текущей архитектуре должно быть выполнено смещение адреса для выполнения преобразования в PFN.

DMA

Работа PCI-устройства может быть предусмотрена как по прямому чтению адресов ввода/вывода, так и (что бывает гораздо чаще) с использованием механизмов DMA (Direct Memory Access). Только простые и низкоскоростные устройства используют программный ввод/вывод. Передача данных по DMA организуется на аппаратном уровне и выполняется (к примеру, когда программа запрашивает данные через такую функцию, например, как `read()`) в таком порядке:

1. Когда процесс вызывает `read()`, метод драйвера выделяет буфер DMA (или указывает адрес в ранее выделенном буфере) и выдает команду оборудованию передавать свои данные в этот буфер (указывая в этой команде адрес начала буфера и объем передачи).
2. Инициировавший операцию *процессор* после этого или блокируется по доступу к шине памяти (или может выполнять некоторые иные действия в пределах разгрузки конвейера команд процессора, поскольку шина доступа к памяти блокирована).
3. Периферийное устройство аппаратно захватывает шину обмена и записывает данные последовательно в буфер DMA с указанного адреса, после чего вызывает *прерывание*, когда весь заказанный объем передан.
4. Обработчик прерывания подтверждает прерывание и переводит процесс в активное состояние — процесс теперь получает входные данные и имеет возможность читать данные.

С операцией `write()` история симметричная: процессор загружает в устройство начальный адрес и размер передаваемой области, инициирует DMA и уходит с шины памяти. Периферийное устройство, когда оно само полностью автономно считает подготовленный для него обменный блок памяти, уведомит (разбудит) процессор по прерыванию.

Организация обмена по DMA — это основной способ взаимодействия со всеми высокопроизводительными устройствами. С другой стороны, обмен по DMA полностью зависим от деталей аппаратной реализации, поэтому в общем виде может

быть рассмотрен только достаточно поверхностно. Уже из приведенного здесь схематичного описания понятно, что одно из ключевых действий, которые должен выполнить код модуля, — это предоставить устройству буфер для выполнения операций DMA (понятие «предоставить буфер» предполагает указание двух его параметров: начального адреса и размера). Буферы DMA могут выделяться только в строго определенных областях памяти:

- эта память должна распределяться в физически непрерывной области памяти, поэтому выделение посредством `vmalloc()` неприменимо, — память под буферы должна выделяться `kmalloc()` или `__get_free_pages()`;
- для многих архитектур выделение памяти должно быть специфицировано с флагом `GFP_DMA`. Для PCI-устройств *x*86 это будет выделение ниже адреса `MAX_DMA_ADDRESS=16MB`;
- память должна выделяться, начиная с *границы страницы* физической памяти и в объеме *целых* страниц физической памяти.

Для *распределения памяти* под буферы DMA предоставляется несколько альтернативных групп API (в зависимости от того, что мы хотим получить), их реализации полностью архитектурно зависимы, но вызовы создают уровень абстракций:

1. Coherent DMA mapping:

```
void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *dma_handle, gfp_t flag);
void dma_free_coherent(struct device *dev, size_t size, void *vaddr, dma_addr_t dma_handle);
```

   Здесь не требуется распределять предварительно буфер DMA — этот способ применяется для устойчивых распределений многократно (повторно) используемых буферов.

2. Streaming DMA mapping:

```
dma_addr_t dma_map_single(struct device *dev, void *ptr, size_t size,
                          enum dma_data_direction direction);
void dma_unmap_single(struct device *dev, dma_addr_t dma_handle, size_t size,
                          enum dma_data_direction direction);
```

   где `direction` — это направление передачи данных: `PCI_DMA_TODEVICE`, `PCI_DMA_FROMDEVICE`, `PCI_DMA_BIDIRECTIONAL`, `PCI_DMA_NONE`.

   Этот способ применяется для выделения под однократные операции.

3. DMA pool:

```
#include <linux/dmapool.h>
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
                 size_t size, size_t align, size_t allocation);
void dma_pool_destroy(struct dma_pool *pool);
void *dma_pool_alloc(struct dma_pool *pool, gfp_t mem_flags, dma_addr_t *handle);
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t handle);
```

   Нередко бывает необходимо частое выделение *малых* областей для DMA-обмена, но `dma_alloc_coherent()` допускает минимальное выделение только в одну физическую страницу (`PAGE_SIZE`: 4, 8, 64, ... Kb — в зависимости от архитектуры).

В таком случае оптимальным становится `dma_pool_alloc()` (как можно видеть из прототипов, пул `struct dma_pool` должен быть сначала создан `dma_pool_create()`, и только затем из него производятся выделения `dma_pool_alloc()`).

4. Старый (перешедший еще из ядра 2.4) API — PCI-специфический интерфейс: два (две пары вызовов) метода, аналогичных соответственно п. 1 и 2. Утверждается, что новый, описанный ранее интерфейс, независим от вида аппаратных шин — в перспективе на новые витки развития. Этот же (старый) API разрабатывался исключительно в ориентации на PCI-шину:

```
void *pci_alloc_consistent(struct device *dev, size_t size, dma_addr_t *dma_handle);
void pci_free_consistent(struct device *dev, size_t size, void *vaddr, dma_addr_t
dma_handle);
dma_addr_t pci_map_single(struct device *dev, void *ptr, size_t size, int direction);
void pci_unmap_single(struct device *dev, dma_addr_t dma_handle, size_t size, int
direction);
```

Выделив любым подходящим способом блок памяти для обмена по DMA, драйвер выполняет последовательность операций (обычно это проделывается в цикле, в чем и состоит работа драйвера):

1. Адрес начала блока записывается в соответствующий регистр одной из шести областей ввода/вывода PCI-устройства, как обсуждалось ранее: конкретные адреса таких регистров здесь и далее определяются исключительно спецификацией устройства.
2. Еще в один специфический регистр заносится длина блока для обмена.
3. Наконец, в регистр команды заносится значение (чаще это выделенный бит) команды начала операции по DMA.
4. Когда внешнее PCI-устройство сочтет, что оно готово приступить к выполнению этой операции, оно аппаратно захватывает шину PCI и под собственным управлением записывает (считывает) указанный блок данных.
5. По завершении выполнения операции устройство освобождает шину PCI под управление процессора и извещает систему прерыванием по выделенной устройству линии IRQ о завершении операции.

Из сказанного здесь легко понять, что принципиальной операцией при организации DMA-обмена в модуле является только создание буфера DMA — все остальное должно исполнять периферийное устройство. Примеры различного выделения буферов DMA приведены в папке dma сопровождающего книгу файлового архива (идея тестов заимствована из [6], а результаты выполнения показаны там же в файле dma.hist). Вот как это происходит при использовании нового API:

lab1_dma.c

```
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/slab.h>
#include <linux/dma-mapping.h>
#include <linux/dmapool.h>
```

```
#include "out.c"
#define pool_size 1024
#define pool_align 8

// int direction = PCI_DMA_TODEVICE ;
// int direction = PCI_DMA_FROMDEVICE ;
static int direction = PCI_DMA_BIDIRECTIONAL;
//int direction = PCI_DMA_NONE;

static int __init my_init(void) {
   char *kbuf;
   dma_addr_t handle;
   size_t size = (10 * PAGE_SIZE);
   struct dma_pool *mypool;
   /* dma_alloc_coherent method */
   kbuf = dma_alloc_coherent(NULL, size, &handle, GFP_KERNEL);
   output(kbuf, handle, size, "This is the dma_alloc_coherent() string");
   dma_free_coherent(NULL, size, kbuf, handle);
   /* dma_map/unmap_single */
   kbuf = kmalloc(size, GFP_KERNEL);
   handle = dma_map_single(NULL, kbuf, size, direction);
   output(kbuf, handle, size, "This is the dma_map_single() string");
   dma_unmap_single(NULL, handle, size, direction);
   kfree(kbuf);
   /* dma_pool method */
   mypool = dma_pool_create("mypool", NULL, pool_size, pool_align, 0);
   kbuf = dma_pool_alloc(mypool, GFP_KERNEL, &handle);
   output(kbuf, handle, size, "This is the dma_pool_alloc() string");
   dma_pool_free(mypool, kbuf, handle);
   dma_pool_destroy(mypool);
   return -1;
}
```

Тот же код, но использующий специфичный для PCI API:

lab1_dma_PCI_API.c

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

#include "out.c"

// int direction = PCI_DMA_TODEVICE ;
// int direction = PCI_DMA_FROMDEVICE ;
static int direction = PCI_DMA_BIDIRECTIONAL;
//int direction = PCI_DMA_NONE;
```

```
static int __init my_init(void) {
   char *kbuf;
   dma_addr_t handle;
   size_t size = (10 * PAGE_SIZE);
   /* pci_alloc_consistent method */
   kbuf = pci_alloc_consistent(NULL, size, &handle);
   output(kbuf, handle, size, "This is the pci_alloc_consistent() string");
   pci_free_consistent(NULL, size, kbuf, handle);
   /* pci_map/unmap_single */
   kbuf = kmalloc(size, GFP_KERNEL);
   handle = pci_map_single(NULL, kbuf, size, direction);
   output(kbuf, handle, size, "This is the pci_map_single() string");
   pci_unmap_single(NULL, handle, size, direction);
   kfree(kbuf);
   /* let it fail all the time! */
   return -1;
}
```

Каждый из методов (в обоих тестах) последовательно создает буфер для DMA-операций, записывает туда строку, именующую метод создания, вызывает диагностику и удаляет этот буфер. Вот общая часть двух модулей, в частности, содержащая функцию диагностики:

out.c

```
static int __init my_init(void);
module_init(my_init);

MODULE_AUTHOR("Jerry Cooperstein");
MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_DESCRIPTION("LDD:1.0 s_23/lab1_dma.c");
MODULE_LICENSE("GPL v2");

#define MARK "=> "
 static void output(char *kbuf, dma_addr_t handle, size_t size, char *string) {
   unsigned long diff;
   diff = (unsigned long)kbuf - handle;
   printk(KERN_INFO MARK "kbuf=%12p, handle=%12p, size = %d\n",
           kbuf, (void*)(unsigned long)handle, (int)size);
   printk(KERN_INFO MARK "(kbuf-handle)= %12p, %12lu, PAGE_OFFSET=%12lu, compare=%lu\n",
           (void*)diff, diff, PAGE_OFFSET, diff - PAGE_OFFSET);
   strcpy(kbuf, string);
   printk(KERN_INFO MARK "string written was, %s\n", kbuf);
}
```

Вот как выглядит выполнение этих примеров:

```
$ sudo insmod lab1_dma.ko
insmod: error inserting 'lab1_dma.ko': -1 Operation not permitted
```

```
$ dmesg | tail -n200 | grep '=>'
=> kbuf=    c0c10000, handle=      c10000, size = 40960
=> (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
=> string written was, This is the dma_alloc_coherent() string
=> kbuf=    d4370000, handle=    14370000, size = 40960
=> (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
=> string written was, This is the dma_map_single() string
=> kbuf=    c0c02000, handle=      c02000, size = 40960
=> (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
=> string written was, This is the dma_pool_alloc() string
$ sudo insmod lab1_dma_PCI_API.ko
insmod: error inserting 'lab1_dma.ko': -1 Operation not permitted
$ dmesg | tail -n50 | grep '=>'
=> kbuf=    c0c10000, handle=      c10000, size = 40960
=> (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
=> string written was, This is the pci_alloc_consistent() string
=> kbuf=    d4370000, handle=    14370000, size = 40960
=> (kbuf-handle)=     c0000000,   3221225472, PAGE_OFFSET=  3221225472, compare=0
=> string written was, This is the pci_map_single() string
```

Эти примеры интересны не столько своими результатами, сколько тем, что фрагменты этого кода могут быть использованы в качестве стартовых шаблонов для написания реальных DMA-обменов.

Предупреждение

Показанные в этом разделе образцы кода работают непосредственно с оборудованием компьютера, поэтому они экстремально к нему чувствительны, и какие-то из методов могут аварийно завершаться в зависимости от конкретики оборудования (чипсета) и использующейся версии ядра. Аварийное завершение здесь не приводит к краху системы, но аварийно завершает загрузку модуля и делает его остатки невыгружаемыми, при этом можно ожидать увидеть что-то типа такого:

```
$ sudo insmod lab1_dma.ko
Убито
$ lsmod | head -n3
Module                  Size  Used by
lab1_dma               1412  1
8021q                 20506  0
$ dmesg | tail -n50
...
[ 1485.891268] lab1_dma: loading out-of-tree module taints kernel.
[ 1485.891505] => kbuf=ffff880035c50000, handle=    35c50000, size = 40960
[ 1485.891509] => (kbuf-handle)= ffff880000000000, 18446612132314218496,
PAGE_OFFSET=18446612132314218496, compare=0
[ 1485.891510] => string written was, This is the dma_alloc_coherent() string
[ 1485.891533] BUG: unable to handle kernel NULL pointer dereference at 00000000000001f8
[ 1485.891897] IP: [<ffffffff813d8e22>] swiotlb_map_page+0x32/0xd0
[ 1485.892163] PGD 0
[ 1485.892319] Oops: 0000 [#1] PREEMPT SMP
...
```

Устройства USB

Если разрабатывать драйверы для PCI-устройств вряд ли придется многим из читающих этот текст программистам (автору пришлось, и это врагу не пожелаешь), то необходимость обеспечивать поддержку USB-устройств возникает в десятки раз чаще. Сразу же здесь отметим не совсем очевидную вещь: поддержка PCI актуальна главным образом для платформ Intel *x*86 (и некоторых других платформ, поддерживающих аппаратную спецификацию PCI). А вот поддержка USB реализуется и требуется во всех архитектурах, поддерживаемых Linux, и в первую очередь в бурно развивающейся линии устройств на ARM-процессорах.

Некоторые технические детали

USB-терминология охватывает три *поколения* стандартов (включающих как минимум 8 модификаций) — дифференциация происходит по скорости обмена и конструктивным особенностям кабелей и оконечных разъемов (и не всегда и не во всем совместимых снизу вверх):

- первоначальный стандарт 1.0 — сейчас это уже практически архаика, которую даже на старых устройствах почти не встретишь. В стандарте 1.0 определялась скорость до 1,5 Мбит/с, стандарт 1.1 определял скорость до 12 Мбит/с. Еще 20 лет назад на смену первопроходцу USB 1.0 пришел улучшенный USB 2.0;
- USB версии 2.0... Как и первая версия, эта спецификация использует два вида проводов. По витой паре идет передача данных, а по второму типу провода — питание устройства, от которого и производится передача информации. Такой тип подключения годится только для устройств с малым потреблением тока. Для принтеров и другой такого рода техники приходится использовать собственные блоки питания. В стандарте оговаривалось три подстандарта (режима работы):
  - Low-speed, 10–1500 Кбит/с (клавиатуры, геймпады, мыши);
  - Full-speed, 0,5–12 Мбит/с (аудио- и видеоустройства);
  - High-speed, 25–480 Мбит/с (видеоустройства, устройства для хранения данных);
- в 2008 году появился стандарт USB 3.0. Скорость передачи данных выросла с 480 Мбит/с до 5 Гбит/с. Помимо скорости передачи данных, USB 3.0 отличается от версии 2.0 и силой тока: вместо 500 мА USB 3.0 способен отдавать до 900 мА. Конструктивно (разъемы и кабели) версии 3.0 совместимы с предыдущими. Но это не значит, что кабели предыдущих версий станут работать на новых скоростях: в версии USB 2.0 используются 4 провода, а у USB 3.0 их 8. Для однозначной идентификации разъемы USB 3.0 принято изготавливать из пластика синего цвета, разъемы же предыдущих версий черные;
- в 2013 году появляется версия USB 3.1 с максимальной заявленной скорость передачи данных до 10 Гбит/с и выходной мощностью до 100 Вт (20 В, 5 А). Одновременно с обновленным стандартом появился и принципиально новый

разъем — USB type-C. Он навсегда решил проблему неправильного подключения кабеля, т. к. стал симметричным и универсальным, и теперь все равно, какой стороной подключать провод к устройству;

- в 2017 году появилась информация о новой версии — USB 3.2. Она получила сразу два канала (больше проводов) по 10 Гбит/с в каждую сторону и суммарную скорость в итоге 20 Гбит/с. Стандарт USB 3.2 также обратно совместим с режимами USB 3.1, 3.0 и ниже. Поддерживается типом подключения USB-C на более современных устройствах (пока в стандартных компьютерных конфигурациях, работающих под Linux, мы его вряд ли встретим).

Распиновка контактов USB-устройств и вид их оконечных разъемов показаны на рис. 6.2.

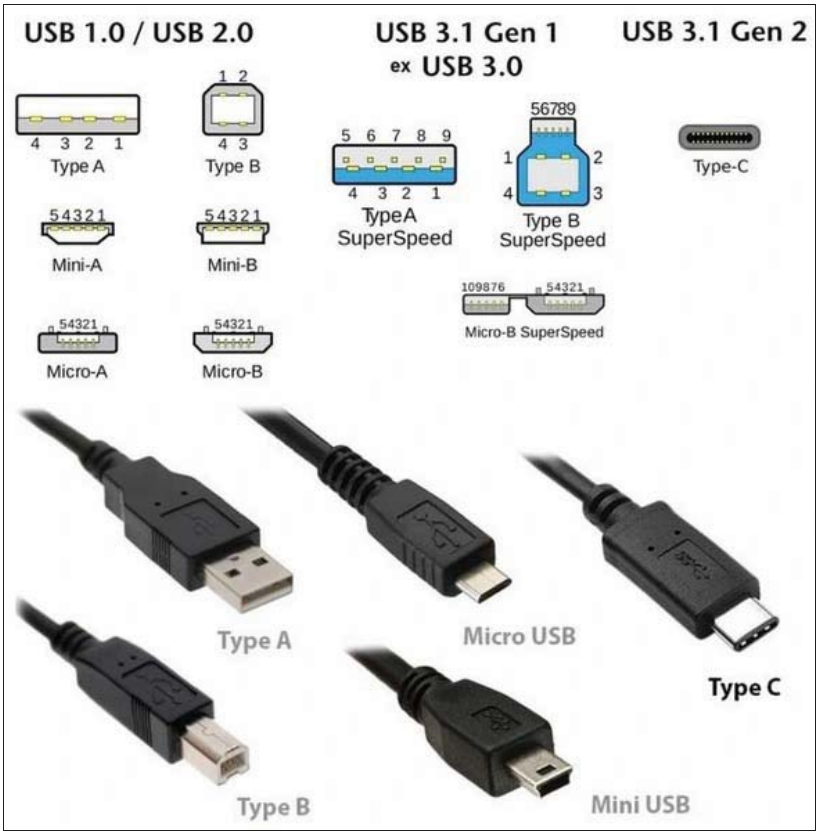


Рис. 6.2. USB-стандарты и их оконечные разъемы

Обмен во всех стандартах происходит по одной или нескольким дифференциальным последовательным линиям (контакты D+ и D–). Стандарты оговаривают оконечные разъемы, которые, кроме дифференциальной линии, содержат две линии питания оконечного устройства.

Поддержка в Linux

Стандарт USB описывает протокол ведущий/ведомый (master/slave), где ведущим *всегда* является USB-хост, а ведомым — периферийное устройство. Контроллер хоста, в свою очередь, является одним из устройств на PCI-шине, как это уже упоминалось ранее, например:

```
$ lspci | grep USB
00:10.0 USB controller: Advanced Micro Devices, Inc. [AMD] FCH USB XHCI Controller (rev 11)
00:12.0 USB controller: Advanced Micro Devices, Inc. [AMD] FCH USB EHCI Controller (rev 39)
00:13.0 USB controller: Advanced Micro Devices, Inc. [AMD] FCH USB EHCI Controller (rev 39)
```

В качестве хоста в системе могут присутствовать контроллеры *разных* стандартов (*несовместимых* на нижнем уровне интерфейса работы с хостом):

- UHCI (Universal Host Controller Interface) — спецификация, инициализированная Intel;
- OHCI (Open Host Controller Interface) — спецификация, созданная компаниями Compaq и Microsoft;
- EHCI (Enhanced Host Controller Interface) — спецификация для поддержки стандарта USB 2.0;
- XHCI (eXtensible Host Controller Interface) — спецификация для поддержки стандарта USB 3.0;
- WHCI (Wireless USB) — поддержка протоколов Wireless USB;
- USB OTG — спецификация, популярная во встраиваемых и мобильных устройствах, обеспечивающая, в частности, поддержку dual-role (DRD) устройств, которые могут выступать либо как хост, либо как устройство в зависимости от ситуации (спецификация USB OTG не относится к кругу наших интересов, но должна быть упомянута для полноты картины).

Спецификации UHCI и OHCI появились еще во времена USB 1.0. Для USB 2.0 был разработан EHCI. Для USB 3.0 используется универсальный интерфейс XHCI. Множество спецификаций, которые могут поддерживаться вашим компьютером, определяется совокупностью нескольких факторов: а) аппаратной поддержкой со стороны примененного чипа USB; б) поддержкой, которая реализована в вашем дистрибутиве Linux; в) конфигурационными параметрами, с которыми собрано ядро вашей системы; г) кабелями USB, которые вы используете (как уже вскользь упоминалось).

К счастью для разработчика, низкоуровневый слой поддержки USB в ядре Linux в значительной мере нивелирует различия спецификаций контроллера для уровня API, используемого при написании модулей поддержки устройств. Поддержка разных *типов* контроллеров и другие опции USB низкого уровня должны быть разрешены и скомпилированы в ядре (обычно это имеет место), а проверить, какие опции доступны, можно так:

```
$ cat /boot/config-`uname -r` | grep _USB_ | grep HCI_HCD=
CONFIG_USB_XHCI_HCD=y
CONFIG_USB_EHCI_HCD=y
```

```
CONFIG_USB_OHCI_HCD=y
CONFIG_USB_UHCI_HCD=y
CONFIG_USB_WHCI_HCD=m
```

Схема *идентификации* конкретных устройств парой VID:PID (Vendor ID : Product ID — производитель : изделие), оказавшаяся плодотворной ранее для устройств PCI, была перенесена и на устройства USB (пара значений после ID в следующем выводе):

```
$ lsusb
Bus 001 Device 002: ID 0424:2503 Standard Microsystems Corp. USB 2.0 Hub
Bus 001 Device 003: ID 1a40:0101 TERMINUS TECHNOLOGY INC. USB-2.0 4-Port HUB
Bus 001 Device 005: ID 046d:080f Logitech, Inc. Webcam C120
Bus 004 Device 002: ID 046d:c517 Logitech, Inc. LX710 Cordless Desktop Laser
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 006: ID 03f0:171d Hewlett-Packard Wireless (Bluetooth + WLAN) Interface
[Integrated Module]
Bus 001 Device 007: ID 08ff:2580 AuthenTec, Inc. AES2501 Fingerprint Sensor
Bus 001 Device 009: ID 16d5:6502 AnyDATA Corporation CDMA/UMTS/GPRS modem
```

На один контроллер USB могут быть последовательно подключены до 127 устройств. Но непосредственно подключать одно устройство к другому нельзя, поскольку питание таких устройств осуществляется по той же шине. Поэтому для подключения дополнительных устройств используются специальные хабы, обеспечивающие снабжение устройств энергией. В результате USB-устройства образуют дерево, каждая нетерминальная вершина которого является хабом:

```
$ lsusb -t
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=ehci-pci/2p, 480M
    |__ Port 1: Dev 2, If 0, Class=Hub, Driver=hub/8p, 480M
        |__ Port 2: Dev 3, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
        |__ Port 2: Dev 3, If 1, Class=Human Interface Device, Driver=usbhid, 1.5M
        |__ Port 8: Dev 4, If 3, Class=Human Interface Device, Driver=usbhid, 12M
        |__ Port 8: Dev 4, If 1, Class=Audio, Driver=snd-usb-audio, 12M
        |__ Port 8: Dev 4, If 2, Class=Audio, Driver=snd-usb-audio, 12M
        |__ Port 8: Dev 4, If 0, Class=Audio, Driver=snd-usb-audio, 12M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci-pci/2p, 480M
    |__ Port 1: Dev 2, If 0, Class=Hub, Driver=hub/6p, 480M
        |__ Port 6: Dev 3, If 0, Class=Hub, Driver=hub/6p, 480M
            |__ Port 1: Dev 4, If 0, Class=Human Interface Device, Driver=usbhid, 480M
            |__ Port 1: Dev 4, If 1, Class=Human Interface Device, Driver=usbhid, 480M
            |__ Port 1: Dev 4, If 2, Class=Human Interface Device, Driver=usbhid, 480M
```

Каждое устройство USB характеризуется и своей адресацией (размещением на шине), которая имеет формат: <шина>:<устройство> (2 первых числа в каждой стро-

ке). Устройство с номером 1 на каждой шине (из числа 127 возможных) — это *всегда* очередной корневой разветвитель USB (хаб) для этой шины. Адресная информация может существенно меняться в зависимости от того, в какой разъем USB включается устройство, или даже просто при последовательном выключении/включении, тогда как идентификация (VID:PID) остается навсегда закрепленной за устройством.

Список идентификаторов USB (VID:PID) централизованно в сети поддерживается в файле с именем usb.ids — в некоторых дистрибутивах он может присутствовать в системе, в других — нет, но в любом случае лучше воспользоваться самой свежей копией этого файла, например, отсюда: **http://www.linux-usb.org/usb.ids** (образец такого файла по состоянию на 17.08.2022, последний на дату подготовки книги, находится в папке usb сопровождающего книгу файлового архива). Как можно видеть, в идентификации устройств USB присутствуют многие прямые аналогии с идентификацией устройств PCI, но... как утверждается: «дьявол кроется в деталях»:

```
$ cat usb.ids | wc -l
25244
```

А детали состоят в том, что число общеизвестных USB-устройств исчисляется уже *десятками тысяч*, а с учетом «серых» производителей и в несколько раз больше, и ежедневно это число увеличивается на сотни. В таких условиях *одним* и тем же модулем ядра обычно реализуется поддержка нескольких *десятков* или даже *сотен* типов функционально подобных устройств. Выяснить, каким модулем поддерживается интересующее нас USB-устройство (по VID:PID), можно такой формой команды:

```
$ modprobe -c | grep -i 16d5 | grep -i 6502
alias usb:v16D5p6502d*dc*dsc*dp*ic*isc*ip* option
```

которая (команда) черпает свою информацию из файла modules.alias:

```
$ cat /lib/modules/`uname -r`/modules.alias | grep -i 16d5 | grep -i 6502
alias usb:v16D5p6502d*dc*dsc*dp*ic*isc*ip* option
```

Пара слов о USB-модемах

Речь идет о USB-модемах 2G/3G/4G/LTE... — очень скоро мы начнем разбираться, наверное, с 5G. Это совершенно частный вопрос... но уж очень часто о нем спрашивают, и очень часто приходится разбираться с очередной моделью модема. Проблема состоит в том, что производители таких модемов *практически никогда* в спешке выхода на рынок не отрабатывают поддержку своих изделий в Linux. И поддержку тысяч специфических проприетарных устройств вынуждены обеспечивать участники команды развития ядра Linux... что никак не является их профильной деятельностью.

В качестве тестового устройствах в некоторых примерах будет использоваться старенький EUDO модем ADU-510A как представитель класса мобильных WAN-модемов (но конкретные модели не играют роли). Как мы увидим, названное

устройство (16d5:6502) поддерживается модулем option. Вообще, как показывает экспериментальный опыт (на достаточно широком наборе таких устройств), из-за огромного разнообразия моделей именно беспроводных модемов WAN поддержка конкретного экземпляра (это и вызывает множество вопросов) может осуществляться одним из *mpex* модулей ядра (о которых мы можно запросить дополнительную информацию)[2]:

- модуль usb_storage — поддержка более старых моделей. Устройство представляется как три последовательные линии с именами: /dev/ttyACM0, /dev/ttyACM1, /dev/ttyACM2:

```
$ modinfo usb_storage | head
filename:       /lib/modules/5.4.0-124-generic/kernel/drivers/usb/storage/usb-storage.ko
license:        GPL
description:    USB Mass Storage driver for Linux
author:         Matthew Dharm <mdharm-usb@one-eyed-alien.net>
srcversion:     C62DE01B04F29D62503A44A
alias:          usb:v*p*d*dc*dsc*dp*ic08isc06ip50in*
alias:          usb:v*p*d*dc*dsc*dp*ic08isc05ip50in*
alias:          usb:v*p*d*dc*dsc*dp*ic08isc04ip50in*
alias:          usb:v*p*d*dc*dsc*dp*ic08isc03ip50in*
alias:          usb:v*p*d*dc*dsc*dp*ic08isc02ip50in*
$ modinfo usb_storage | grep alias | wc -l
404
$ modprobe -c | grep -w usb_storage | wc -l
429
```

  Как ни считай, а поддерживается (с учетом семейственности) этим модулем порядка 400–430 разнообразных моделей USB-модемов;

- модуль option — поддержка наиболее распространенных моделей. Устройство здесь представляется как несколько (чаще всего 3, но бывает и 1, и до 5) последовательных линий с именами: /dev/ttyUSB0, /dev/ttyUSB1, /dev/ttyUSB2, ...:

```
$ modinfo option | head
filename:       /lib/modules/5.4.0-124-generic/kernel/drivers/usb/serial/option.ko
license:        GPL v2
description:    USB Driver for GSM modems
author:         Matthias Urlichs <smurf@smurf.noris.de>
srcversion:     2B9C94EE65CA5A7235E5DBF
alias:          usb:v305Ap1406d*dc*dsc*dp*icFFisc*ip*in*
alias:          usb:v305Ap1405d*dc*dsc*dp*icFFisc*ip*in*
alias:          usb:v305Ap1404d*dc*dsc*dp*icFFisc*ip*in*
alias:          usb:v2DF3p9D03d*dc*dsc*dp*icFFisc*ip*in*
alias:          usb:v2CB7p01A4d*dc*dsc*dp*icFFisc*ip*in*
```

[2] Возможно, позже появился еще модуль или модули от нового разработчика… но я уже далее не занимался изучением этого вопроса.

```
$ modinfo option | grep alias | wc -l
1329
$ modprobe -c | grep -w option | head -n10
alias usb:v03F0p421Dd*dc*dsc*dp*icFFiscFFipFFin* option
alias usb:v03F0pA31Dd*dc*dsc*dp*icFFisc06ip10in* option
alias usb:v03F0pA31Dd*dc*dsc*dp*icFFisc06ip12in* option
alias usb:v03F0pA31Dd*dc*dsc*dp*icFFisc06ip13in* option
alias usb:v03F0pA31Dd*dc*dsc*dp*icFFisc06ip14in* option
alias usb:v03F0pA31Dd*dc*dsc*dp*icFFisc06ip1Bin* option
alias usb:v0408pEA02d*dc*dsc*dp*ic*isc*ip*in* option
alias usb:v0408pEA03d*dc*dsc*dp*ic*isc*ip*in* option
alias usb:v0408pEA04d*dc*dsc*dp*ic*isc*ip*in* option
alias usb:v0408pEA05d*dc*dsc*dp*ic*isc*ip*in* option
$ modprobe -c | grep -w option | wc -l
1329
```

Это самая многочисленная группа поддержки достаточно новых устройств.

- модуль qcaux — это (пока?) малочисленная группа поддержки новых многостандартных (CDMA, GPRS, EDGE, HSUPA) моделей (например, Pantech/Verizon UMW190). Здесь устройство в /dev представляется как смесь имен разного образца — например: /dev/ttyUSB0, /dev/ttyUSB1, /dev/ttyACM0:

```
$ modinfo qcaux | head
filename:       /lib/modules/5.4.0-124-generic/kernel/drivers/usb/serial/qcaux.ko
license:        GPL v2
srcversion:     988EF28AB8DCCDD11BA956C
alias:          usb:v1FACp0151d*dc*dsc*dp*icFFiscFFipFFin*
alias:          usb:v106Cp*d*dc*dsc*dp*icFFiscFFipFFin*
alias:          usb:v106Cp*d*dc*dsc*dp*icFFiscFEipFFin*
alias:          usb:v106Cp*d*dc*dsc*dp*icFFiscFDipFFin*
alias:          usb:v04E8p6640d*dc*dsc*dp*icFFisc00ip00in*
alias:          usb:v0474p0754d*dc*dsc*dp*icFFiscFFip00in*
alias:          usb:v1004p6000d*dc*dsc*dp*icFFiscFFip00in*
$ modinfo qcaux | grep alias | wc -l
15
$ modprobe -c | grep -w qcaux | head -n10
alias usb:v0474p0754d*dc*dsc*dp*icFFiscFFip00in* qcaux
alias usb:v04E8p6640d*dc*dsc*dp*icFFisc00ip00in* qcaux
alias usb:v1004p6000d*dc*dsc*dp*icFFiscFFip00in* qcaux
alias usb:v106Cp*d*dc*dsc*dp*icFFiscFDipFFin* qcaux
alias usb:v106Cp*d*dc*dsc*dp*icFFiscFEipFFin* qcaux
alias usb:v106Cp*d*dc*dsc*dp*icFFiscFFipFFin* qcaux
alias usb:v106Cp3701d*dc*dsc*dp*icFFisc00ip00in* qcaux
alias usb:v106Cp3702d*dc*dsc*dp*icFFisc00ip00in* qcaux
alias usb:v106Cp3711d*dc*dsc*dp*icFFisc00ip00in* qcaux
alias usb:v106Cp3712d*dc*dsc*dp*icFFisc00ip00in* qcaux
$ modprobe -c | grep -w qcaux | wc -l
15
```

Прежде всего ищите VID:PID своего устройства среди этих свыше 2000 представленных устройств. Еще более детальную (но очень излишне объемную — обращаем внимание на права root) информацию по любому конкретному устройству (крайне нужную разработчику драйвера) изучаем так:

```
# lsusb -vv -d 16d5:6502
Bus 001 Device 009: ID 16d5:6502 AnyDATA Corporation CDMA/UMTS/GPRS modem
Device Descriptor:
...
  idVendor           0x16d5 AnyDATA Corporation
  idProduct          0x6502 CDMA/UMTS/GPRS modem
...
# lsusb -vv -d 16d5:6502 | wc -l
159
```

Устройства USB в коде модуля

Первейшую информацию при разработке модуля об устройстве (даже попавшем нам случайно в руки, и о котором мы ничего не знаем) черпаем, подключая его в USB-разъем и извлекая из него, из вывода команд `lsusb` (VID:PID) и `dmesg` (протокол *подключения/отключения* устройства подсистемой udev/sysfs). Начинаем с этого:

```
$ lsusb
Bus 002 Device 002: ID 0438:7900 Advanced Micro Devices, Inc. Root Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 003: ID 2101:020f ActionStar
Bus 001 Device 002: ID 0438:7900 Advanced Micro Devices, Inc. Root Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 007: ID 0fe6:9700 ICS Advent DM9601 Fast Ethernet Adapter
Bus 003 Device 008: ID 1908:2310 GEMBIRD USB2.0 PC CAMERA
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
$ lsusb -t
/:  Bus 04.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/2p, 5000M
/:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/2p, 480M
    |__ Port 1: Dev 8, If 2, Class=Audio, Driver=snd-usb-audio, 480M
    |__ Port 1: Dev 8, If 0, Class=Video, Driver=uvcvideo, 480M
    |__ Port 1: Dev 8, If 3, Class=Audio, Driver=snd-usb-audio, 480M
    |__ Port 1: Dev 8, If 1, Class=Video, Driver=uvcvideo, 480M
    |__ Port 2: Dev 7, If 0, Class=, Driver=dm9601, 12M
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=ehci-pci/2p, 480M
    |__ Port 1: Dev 2, If 0, Class=Hub, Driver=hub/4p, 480M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci-pci/2p, 480M
    |__ Port 1: Dev 2, If 0, Class=Hub, Driver=hub/4p, 480M
        |__ Port 2: Dev 3, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
        |__ Port 2: Dev 3, If 1, Class=Human Interface Device, Driver=usbhid, 1.5M
```

Попутно обратим внимание, как хорошо в этой системе работает балансировка обработки прерываний на его двух процессорах:

```
$ inxi -C
CPU:
  Info: dual core model: AMD GX-212JC SOC with Radeon R2E Graphics bits: 64 type: MCP cache:
    L2: 1024 KiB
  Speed (MHz): avg: 1198 min/max: N/A cores: 1: 1198 2: 1198
$ cat /proc/interrupts | grep hci
  18:      59283      59348   IO-APIC   18-fasteoi   ehci_hcd:usb1, ehci_hcd:usb2
  19:      10940      10966   IO-APIC   19-fasteoi   ahci[0000:00:11.0]
  26:     940715     940939   PCI-MSI 262144-edge      xhci_hcd
  27:          0          0   PCI-MSI 262145-edge      xhci_hcd
  28:          0          0   PCI-MSI 262146-edge      xhci_hcd
```

У меня на компьютере подключены к USB два интересующих меня устройства (рис. 6.3 и 6.4) — вы же можете использовать любые, имеющиеся под рукой (с соответствующей коррекцией VID:PID).

Рис. 6.3. Внешний Ethernet-адаптер ICS Advent DM9601 Fast Ethernet Adapter, 0fe6:9700

Рис. 6.4. Веб-камера GEMBIRD модель CAM62U, 1908:2310 (с микрофоном, откуда и появился аудиоканал)

Свои же устройства я и буду использовать для подготовки скелета драйвера... При *подключении* устройств мы наблюдаем в системном журнале:

```
$ dmesg
...
[25531.535036] usb 3-2: new full-speed USB device number 7 using xhci_hcd
[25531.669353] usb 3-2: New USB device found, idVendor=0fe6, idProduct=9700
[25531.669364] usb 3-2: New USB device strings: Mfr=0, Product=2, SerialNumber=0
[25531.669369] usb 3-2: Product: USB 2.0 10/100M Ethernet Adaptor
[25531.756774] dm9601 3-2:1.0 eth1: register 'dm9601' at usb-0000:00:10.0-2, Davicom DM96xx
USB 10/100 Ethernet, 00:e0:4c:53:44:58
[25531.845392] dm9601 3-2:1.0 eth1: link up, 100Mbps, full-duplex, lpa 0xFFFF
...
```

```
[25584.713232] usb 3-1: new high-speed USB device number 8 using xhci_hcd
[25584.832347] usb 3-1: New USB device found, idVendor=1908, idProduct=2310
[25584.832354] usb 3-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[25584.832359] usb 3-1: Product: USB2.0 PC CAMERA
[25584.832363] usb 3-1: Manufacturer: Generic
[25584.833667] uvcvideo: Found UVC 1.00 device USB2.0 PC CAMERA (1908:2310)
[25584.835234] uvcvideo 3-1:1.0: Entity type for entity Processing 2 was not initialized!
[25584.835242] uvcvideo 3-1:1.0: Entity type for entity Camera 1 was not initialized!
[25584.835421] input: USB2.0 PC CAMERA as /devices/pci0000:00/0000:00:10.0/usb3/3-1/
3-1:1.0/input/input12
```

Детали нас не интересуют (хотя они информативны и интересны при тонком анализе), но нам важно отчетливо зафиксировать, какими модулями ядра (из дерева исходных кодов ядра) поддерживаются задействованные нами устройства *стандартным* образом (вскоре мы увидим важность этого).

Показанное физическое подключение устройств вызывает целую бурю загрузки *всех* (по цепочке зависимостей) модулей, требуемых для *полной* поддержки устройств (теперь, к этому времени, мы знаем, что это работа правил udev, отрабатываемых в ответ на широковещательные сообщения sysfs через сокет `netlink` — `PF_NETLINK`):

```
$ lsmod | head -n4
Module                  Size  Used by
sr9700                  6295  0
dm9601                  7171  0
usbnet                 20211  2 sr9700,dm9601
$ lsmod | head -n12
Module                  Size  Used by
uvcvideo               77766  0
videobuf2_vmalloc       4987  1 uvcvideo
snd_usb_audio         135352  0
videobuf2_memops        1609  1 videobuf2_vmalloc
videobuf2_v4l2         11436  1 uvcvideo
snd_usbmidi_lib        19654  1 snd_usb_audio
videobuf2_core         26674  2 uvcvideo,videobuf2_v4l2
snd_rawmidi            18472  1 snd_usbmidi_lib
videodev              127185  3 uvcvideo,videobuf2_core,videobuf2_v4l2
snd_seq_device          3368  1 snd_rawmidi
media                  20419  2 uvcvideo,videodev
```

Здесь же, забегая вперед, заметим, что после *отключения* устройств из разъема USB драйверы, загруженные по требованию udev под их VID:PID, *остаются загруженными* — это сейчас окажется очень важным! (В принципе, по сообщениям sysfs на отключение устройства вполне можно *выгружать* подгруженные для него модули, но, наверное, это нецелесообразно).

Еще более детальную информацию (если этой недостаточно) можно получить из рассмотрения протокола широковещательных асинхронных сообщений ядра при

подключении и отключении устройств (выводимых командой udevadm, как это подробно обсуждалось ранее при рассмотрении udev).

В приводимых далее примерах модулей я буду использовать два своих упомянутых ранее устройства (см. рис. 6.3 и 6.4) — напомню при этом, что вы можете взять для экспериментов любое доступное устройство, скорректировав или добавив соответствующие ему VID:PID в код. Пример подключения (и отключения) и регистрации USB-устройства показан в демонстрационном модуле (см. папку usb в сопровождающем книгу файловом архиве) lab1_usb.ko (заимствован из [6] с некоторой корректировкой и заменой, естественно, VID:PID USB-устройства на используемые в эксперименте). Но прежде чем рассматривать пример, отметим, что регистрация USB-устройства в точности напоминает регистрацию PCI-устройства — здесь прямая аналогия. Основой для связывания является определяемая разработчиком большая структура (все описания в <linux/usb.h>):

```
struct usb_driver {
   const char *name;
   int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);
   void (*disconnect) (struct usb_interface *intf);
   int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf);
   ...
}
```

Главное отличие от PCI: функции обратного вызова probe() и disconnect() вызываются не при *загрузке* и *выгрузке* модуля, а каждый раз при физическом *подключении* и *отключении* USB-устройства. Код примера будет выглядеть так:

lab1_usb.c

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

struct my_usb_info {   // своя структура данных, неизвестная ядру
   int connect_count;
};

#define USB_INFO KERN_INFO "MY: "

static int my_usb_probe(struct usb_interface *intf, const struct usb_device_id *id) {
   struct my_usb_info *usb_info;
   struct usb_device *dev = interface_to_usbdev(intf);
   static int my_counter = 0;
   printk(USB_INFO "===== connect\n");
   printk(USB_INFO "devnum=%d, speed=%d\n", dev->devnum, (int)dev->speed);
   printk(USB_INFO "idVendor=0x%hX, idProduct=0x%hX, bcdDevice=0x%hX\n",
           dev->descriptor.idVendor,
           dev->descriptor.idProduct, dev->descriptor.bcdDevice);
```

```
   printk(USB_INFO "class=0x%hX, subclass=0x%hX\n",
           dev->descriptor.bDeviceClass, dev->descriptor.bDeviceSubClass);
   printk(USB_INFO "protocol=0x%hX, packetsize=%hu\n",
           dev->descriptor.bDeviceProtocol,
           dev->descriptor.bMaxPacketSize0);
   printk(USB_INFO "manufacturer=0x%hX, product=0x%hX, serial=%hu\n",
           dev->descriptor.iManufacturer, dev->descriptor.iProduct,
           dev->descriptor.iSerialNumber);
   usb_info = kmalloc(sizeof(struct my_usb_info), GFP_KERNEL);
   usb_info->connect_count = my_counter++;
   usb_set_intfdata(intf, usb_info);
   printk(USB_INFO "connect_count=%d\n\n", usb_info->connect_count);
   return 0;
}

static void my_usb_disconnect(struct usb_interface *intf) {
   struct my_usb_info *usb_info;
   usb_info = usb_get_intfdata(intf);
   printk(USB_INFO "===== disconnect\n");
   kfree(usb_info);
}

static struct usb_device_id my_usb_table[] = {
   { USB_DEVICE(0x046d, 0x080f) }, // Logitech Inc., Webcam C120
   { USB_DEVICE(0x1908, 0x2310) }, // GEMBIRD USB2.0 PC CAMERA
   { USB_DEVICE(0x0fe6, 0x9700) }, // ICS Advent DM9601 Fast Ethernet Adapter
   { }                             // Null terminator (required)
};

MODULE_DEVICE_TABLE(usb, my_usb_table);

static struct usb_driver my_usb_driver = {
   .name = "usb-my",
   .probe = my_usb_probe,
   .disconnect = my_usb_disconnect,
   .id_table = my_usb_table,
};

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

static void my_cleanup_module(void) {
   printk(USB_INFO "Goodbye USB\n");
```

```
    usb_deregister(&my_usb_driver);
}

module_init(my_init_module);
module_exit(my_cleanup_module);
MODULE_LICENSE("GPL v2");
```

Сложность наблюдения подобного модуля (для любого вашего устройства) состоит в том, что необходимо из системы удалить модуль, *ранее* поддерживавший это устройство и обрабатывавший его «горячие» подключения. Сделать это можно, отследив сообщения *такого* модуля в журнале системы при подключениях вашего USB-устройства, что мы и делали раньше командами `dmesg`:

```
$ dmesg
```

1. Прежде выгрузим вручную модули-драйверы, которые были загружены системой (udev) для подддержки этих устройств:

   ```
   $ sudo rmmod uvcvideo
   $ sudo rmmod dm9601
   ```

2. Вот теперь мы можем — при *отключенных* от USB тестируемых устройствах — загрузить свой новый драйвер:

   ```
   $ sudo insmod lab1_usb.ko
   $ dmesg | tail -n3
   [35749.639709] usbcore: deregistering interface driver dm9601
   [36201.963686] MY: Hello USB
   [36201.963767] usbcore: registered new interface driver usb-my
   ```

3. А подключив одно за другим устройства, видим:

   ```
   $ dmesg | tail -n40
   ...
   [36278.722142] usb 3-1: new high-speed USB device number 19 using xhci_hcd
   [36279.066238] usb 3-1: New USB device found, idVendor=1908, idProduct=2310
   [36279.066248] usb 3-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
   [36279.066254] usb 3-1: Product: USB2.0 PC CAMERA
   [36279.066258] usb 3-1: Manufacturer: Generic
   [36[41723.120971] MY: ===== connect
   [41723.120975] MY: devnum=19, speed=3
   [41723.120978] MY: idVendor=0x1908, idProduct=0x2310, bcdDevice=0x100
   [41723.120980] MY: class=0xEF, subclass=0x2
   [41723.120983] MY: protocol=0x1, packetsize=64
   [41723.120986] MY: manufacturer=0x1, product=0x2, serial=0
   [41723.120988] MY: connect_count=7
   ...
   [41762.447115] usb 3-2: new full-speed USB device number 20 using xhci_hcd
   [41762.581637] usb 3-2: New USB device found, idVendor=0fe6, idProduct=9700
   [41762.581647] usb 3-2: New USB device strings: Mfr=0, Product=2, SerialNumber=0
   [41762.581653] usb 3-2: Product: USB 2.0 10/100M Ethernet Adaptor
   ```

```
[41762.583801] MY: ===== connect
[41762.583810] MY: devnum=20, speed=2
[41762.583814] MY: idVendor=0xFE6, idProduct=0x9700, bcdDevice=0x101
[41762.583817] MY: class=0x0, subclass=0x0
[41762.583820] MY: protocol=0x0, packetsize=64
[41762.583823] MY: manufacturer=0x0, product=0x2, serial=0
[41762.583826] MY: connect_count=8
...
```

4. И так:

```
$ lsusb -t
/:  Bus 04.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/2p, 5000M
/:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/2p, 480M
    |__ Port 1: Dev 19, If 2, Class=Audio, Driver=usb-my, 480M
    |__ Port 1: Dev 19, If 0, Class=Video, Driver=usb-my, 480M
    |__ Port 1: Dev 19, If 3, Class=Audio, Driver=usb-my, 480M
    |__ Port 1: Dev 19, If 1, Class=Video, Driver=usb-my, 480M
    |__ Port 2: Dev 20, If 0, Class=, Driver=usb-my, 12M
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=ehci-pci/2p, 480M
    |__ Port 1: Dev 2, If 0, Class=Hub, Driver=hub/4p, 480M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=ehci-pci/2p, 480M
    |__ Port 1: Dev 2, If 0, Class=Hub, Driver=hub/4p, 480M
        |__ Port 2: Dev 3, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
        |__ Port 2: Dev 3, If 1, Class=Human Interface Device, Driver=usbhid, 1.5M
```

5. Ну и точно так же мы можем наблюдать отключение от разъема USB:

```
$ dmesg | tail -n7
[42137.280481] usb 3-2: USB disconnect, device number 20
[42137.280606] MY: ===== disconnect
[42153.281859] usb 3-1: USB disconnect, device number 19
[42153.281953] MY: ===== disconnect
[42153.282033] MY: ===== disconnect
[42153.282083] MY: ===== disconnect
[42153.282129] MY: ===== disconnect
```

То, что мы сделали, — это *главная часть* модуля USB-устройства: идентификация USB-устройства и увязывание его в код модуля. Но это никак не затрагивало обмен данными с устройством. Модель описания аппаратного устройства USB в общем случае включает в себя одну или более *конфигураций* для каждого устройства, но активной в любой момент времени может быть только одна из них. Конфигурации имеют один или более *интерфейсов*, каждый из которых может содержать различные параметры/настройки. Такие интерфейсы могут соответствовать стандарту USB, а могут быть специфичными лишь для определенного производителя/устройства. Интерфейсы имеют одну или более *конечных точек* (endpoints), каждая из которых поддерживает *только один* тип и направление передачи данных (например: «bulk out» или «interrupt in»). Полная конфигурация может иметь до шестнадцати конечных точек в каждом направлении. Передача данных по USB осуществля-

ется пакетами. Для каждой концевой точки хранится запись о максимальном размере пакета. Хост всегда является мастером в обмене, и независимо от направления именно он устанавливает флаг направления обмена: `out` — если хост отсылает данные устройству, `in` — если хост отправляет запрос на прием данных из устройства. Устройство никогда не инициирует передачу данных к хосту.

Любая обменная операция производится только с *конечной* (концевой) *точкой* устройства (endpoint, EP). Только EP может выступать как источник или приемник данных. Устройство может иметь до 32 EP: 16 — на прием и 16 — на передачу. Обращение к выбранному EP происходит по его адресу (*номеру*). Стандарты USB поддерживают 4 *типа* передачи данных:

- `bulk` — для пакетной передачи больших объемов некритичной по времени информации. Размер пакетов: 8, 16, 32, 64 — для USB 1.1 и 512 — для USB 2.0. Используется алгоритм подтверждения и повторной передачи (в случае возникновения ошибок), поэтому этот тип является достоверным. Поддерживаются оба направления: `in` и `out`;
- `control` — служит для передачи конфигурационной и управляющей информации. Используются алгоритмы подтверждения и повторной передачи. Направления: `in` (status) и `out` (setup, control);
- `interrupt` — для получения малых порций критичной по времени информации от устройства. Размер пакета: от 1 до 64 байтов — для USB 1.1 и до 1024 байтов — для USB 2.0. Этот тип предполагает, что устройство будет опрашиваться (со стороны хоста) с заданным интервалом. Направление — только `in`;
- `isochronous` — для передачи реалтайм-потоков на фиксированных скоростях передачи без управления потоком (без подтверждений). Область применения: аудиопотоки, видеопотоки, ... Размер пакета: до 1023 байтов — для USB 1.1 и до 1024 байтов — для USB 2.0. Предусмотрен контроль ошибок на приемной стороне по CRC16. Направления: `in` и `out`.

Из представленных описаний уже понятно то множество «степеней свободы», которое отдано на откуп фантазии разработчика USB-устройства. Все требуемые алгоритмы USB реализованы в ядре Linux, а программисту драйверов предоставляется удобный и простой интерфейс в виде набора функций, макросов, структур (которые ищем в <linux/usb.h>) — таких, например, как (перечень приводится для справки): `usb_register_dev()`, `interface_to_usbdev()`, `usb_control_msg()`, `usb_interrupt_msg()`, `usb_bulk_msg()`, `usb_set_interface()`, `usb_reset_endpoint()`, ... С помощью этого API достаточно несложно реализовать обмен для всякого конкретного случая.

Собственно реализация обмена с USB-устройством *после* его связывания с модулем ядра становится существенно зависимой от специфики самого устройства и протоколов его обмена. Это уже не относится непосредственно к общей функциональности модуля ядра, а поэтому обмен и не может быть рассмотрен в общем виде. Все это обсуждение представлено здесь только для того, чтобы показать картину на качественном уровне. Единственный совет, который может быть дан по этой части: всегда занимайтесь прописыванием обмена в тандеме с коллегой-аппаратчиком, ответственным за «внутренности» разрабатываемого устройства.

Многофункциональные USB-устройства

Ранее уже было сделано небольшое лирическое отступление относительно USB-модемов WAN (EVDO, LTE, WiMAX, ...). Но есть и еще одна особенность, относящаяся не только к ним, но и к разнообразным USB-устройствам (хотя к модемам в наибольшей мере). Это то, что производители начали неудачно называть «многофункциональные устройства»... Появилась эта неприятность в последние 7–8 лет.

Столь популярная у производителей USB-устройств идея многофункциональных USB-устройств состоит в следующем: путем записи по определенной концевой точке (EP) USB, заданной *производителем* байтовой (символьной) управляющей последовательности, VID:PID устройства изменяется, и устройство приобретает совершенно другую функциональность. То есть при «первичном» VID:PID это, как правило, блочное устройство, по функциям подобное CD-ROM, на которое производитель записывает свое проприетарное программное обеспечение, какую-то краткую документацию, инсталляционную инструкцию... А при переключении на «вторичный» VID:PID оно становится, к примеру, тем же модемом. Понятно, что без переключения на «вторичный» VID:PID собственно функции устройства недоступны, и оно бесполезно. Идея, по сути, чисто коммерческая и связанная с тем, что многие производители озабочены обеспечением поддержки своих поделок *только* под операционной системой Windows.

Позже эту технику производители распространили и на другие типы устройств. Логика состоит в том, что при повторных подключениях устройства к USB-шине *модуль*-драйвер (или какое-то *приложение* пользовательского пространства) должен засылать управляющую последовательность в устройство, чем *переключать* его в режим модема (сменив при этом VID:PID). В более общем случае: а) это могут быть устройства различной функциональности, а не только модемы, и б) может быть и более двух функциональностей, которые заложены в устройство. При написании модулей поддержки нужно быть готовым к такой архитектуре.

Поскольку о таком «сложном» поведении USB-устройств задают очень много вопросов, то проведем небольшое расследование по поводу того, какой последовательностью действий можно провести разборки с подобными устройствами (рис. 6.5), попади они вам в руки...

Рис. 6.5. Пример многофункционального USB-устройства: модем + роутер

Проследить процесс смены VID:PID многофункционального устройства можно, запустив монитор `udevadm` (обсуждался ранее) перед физическим подключением устройства к USB и наблюдая генерируемые при этом сообщения sysfs. В примерах я экспериментировал с 3G-модемом ADU-510A от AnyDATA (что было под рукой, и чего уже не жалко) — вы же можете проделать то же с любым, оказавшимся под рукой USB-модемом. Последовательность сообщений об изменениях состояний весьма объемная, но нет другого способа отследить происходящие процессы, кроме ее анализа. Далее показаны только некоторые, ключевые для понимания сообщения ядра, из числа 50 (!) сообщений, передаваемых через `netlink`-сокет в ходе отработки такого «горячего» подключения:

```
$ udevadm monitor --kernel --property
monitor will print the received events for:
KERNEL - the kernel uevent

KERNEL[9534.471672] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4 (usb)
ACTION=add
BUSNUM=001
DEVNAME=/dev/bus/usb/001/010
DEVNUM=010
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4
DEVTYPE=usb_device
MAJOR=189
MINOR=9
PRODUCT=5c6/1000/0
SEQNUM=2404
SUBSYSTEM=usb
TYPE=0/0/0
...
KERNEL[9534.478431] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/host8
(scsi)
ACTION=add
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/host8
DEVTYPE=scsi_host
SEQNUM=2406
SUBSYSTEM=scsi
...
KERNEL[9535.490388] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/host8/
target8:0:0/8:0:0:0/block/sr1 (block)
ACTION=add
DEVNAME=/dev/sr1
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/host8/target8:0:0/8:0:0:0/
block/sr1
DEVTYPE=disk
MAJOR=11
MINOR=1
SEQNUM=2411
SUBSYSTEM=block
...
```

```
KERNEL[9536.048197] remove   /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/host8/
target8:0:0/8:0:0:0/block/sr1 (block)
ACTION=remove
DEVNAME=/dev/sr1
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/host8/target8:0:0/8:0:0:0/
block/sr1
DEVTYPE=disk
MAJOR=11
MINOR=1
SEQNUM=2420
SUBSYSTEM=block
...
KERNEL[9537.336850] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4 (usb)
ACTION=add
BUSNUM=001
DEVNAME=/dev/bus/usb/001/011
DEVNUM=011
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4
DEVTYPE=usb_device
MAJOR=189
MINOR=10
PRODUCT=16d5/6502/0
SEQNUM=2427
SUBSYSTEM=usb
TYPE=0/0/0
...
KERNEL[9537.338995] add      /devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/
ttyUSB0/tty/ttyUSB0 (tty)
ACTION=add
DEVNAME=/dev/ttyUSB0
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb1/1-3/1-3.4/1-3.4:1.0/ttyUSB0/tty/ttyUSB0
MAJOR=188
MINOR=0
SEQNUM=2430
SUBSYSTEM=tty
...
```

Первым сообщением (`SEQNUM=2404`) фиксируется обнаружение на USB-шине устройства `5c6:1000` (что, как мы увидим далее, никак не соответствует VID:PID модема). Позже (`SEQNUM=2406`) это устройство квалифицируется как `SCSI` (`MAJOR=11`, `MINOR=1`) и для него устанавливается (`SEQNUM=2411`) модуль-драйвер блочного устройства /dev/sr1 (т. е. устройство CD-ROM). Но дальше (`SEQNUM=2420`) следует размонтирование и удаление устройства (`MAJOR=11`, `MINOR=1`), обнаружение (`SEQNUM=2427`) нового устройства `16d5:6502` (и это уже и есть 3G-модем) и делается вся дальнейшая инициализация этого устройства.

В результате такого установочного процесса мы получаем новое USB-устройство:

```
$ lsusb | grep AnyDATA
Bus 001 Device 011: ID 16d5:6502 AnyDATA Corporation CDMA/UMTS/GPRS modem
```

А в каталоге устройств — три дополнительных устройства, имитирующих поведение *последовательных* линий передачи, из которых /dev/ttyUSB0 является линией AT-совместимого модема, для которой мы можем устанавливать традиционное PPP-соединение любым из известных способов — например, с помощью какого-либо диалера (весьма популярным является `wvdial`, но можно это сделать и классическим способом ручной настройки PPP):

```
$ ls -l /dev/ttyUSB*
crw-rw---- 1 root dialout 188, 0 май 13 12:57 /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 1 май 13 12:57 /dev/ttyUSB1
crw-rw---- 1 root dialout 188, 2 май 13 12:57 /dev/ttyUSB2
```

Мы это сделаем с помощью известного инструмента (программы, GUI-апплета) `NetworkManager` (рис. 6.6):

```
$ which NetworkManager
/usr/sbin/NetworkManager
```

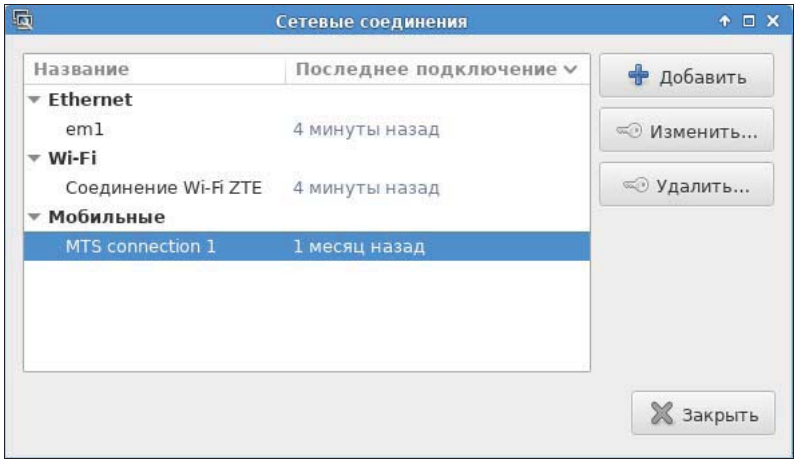


Рис. 6.6. Создание нового сетевого интерфейса

Убеждаемся, что все сказанное действительно имеет место (сетевой интерфейс `ppp0`):

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: em1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT
group default qlen 1000
    link/ether a0:1d:48:f4:93:5c brd ff:ff:ff:ff:ff:ff
3: wlo1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DORMANT
group default qlen 1000
    link/ether 34:23:87:d6:85:0d brd ff:ff:ff:ff:ff:ff
4: ppp0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN
mode DEFAULT group default qlen 3
    link/ppp
```

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    1024   0        0 em1
80.255.73.34    0.0.0.0         255.255.255.255 UH    0      0        0 ppp0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 em1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 wlo1
```

Остается невыясненным единственный вопрос: *кто* и с помощью *какого механизма* переключил устройство 5c6:1000 на устройство 16d5:6502 между SEQNUM=2411 и SEQNUM=2427? Это может делать, в принципе, непосредственно программный код *вашего модуля* после его установки, и такой подход имеет смысл проанализировать при подготовке проекта. Но в нашем случае это выполнил *прикладной* пакет usb_modeswitch, разработанный специально для таких ситуаций, потому что подобные потребности многократно повторяются для огромного спектра USB-устройств:

```
$ aptitude show usb-modeswitch
...
Описание: инструмент переключения режима для управления USB-устройствами "flip flop"
Некоторые современные устройства USB имеют на борту свои собственные патентованные драйверы
для Windows, особенно WAN-донглы. При первом подключении они работают как флеш-память
и начинают установку драйвера оттуда. Если драйвер уже установлен, запоминающее устройство
исчезает, и появляется новое устройство, например, USB-модем. Эта функция называется ZeroCD.

В системе Debian это не требуется, потому что драйвер включен в модуль usbserial ядра Linux.
Тем не менее устройство продолжает отображаться по умолчанию как usb-накопитель.
usb-modeswitch решает эту проблему посылкой команды, которая переключает устройство
из "usb-storage" в "usbserial".
```

Приложение usb_modeswitch позволяет в *ручном режиме* переключать (разнообразным образом) режимы многофункциональных устройств — например, так:

```
sudo usb_modeswitch -v 12d1 -p 1003 -V 12d1 -P 1001 -HR -M
"5553424312345678000000000000001106200000010000000000000000000000"
```

Здесь показано переключение устройств Huawei E220, E230, E270, E870 из 12d1:1003 в 12d1:1001. Опция -M и задает переключающую байтовую последовательность.

Но это годится для экспериментов, а для реального использования usb_modeswitch хранит конфигурационные файлы для большого количества известных устройств. В предыдущих версиях конфигурационные файлы устройств лежали в *каталоге* /etc/usb_modeswitch.d, сейчас же они находятся в *архиве* (разархивировать не надо) :

```
$ ls -l /usr/share/usb_modeswitch/configPack.tar.gz
-rw-r--r-- 1 root root 18024 фев 24  2020 /usr/share/usb_modeswitch/configPack.tar.gz
$ tar -tvf /usr/share/usb_modeswitch/configPack.tar.gz | head
-rw-r--r-- 0/0             130 2017-01-19 21:47 03f0:002a
-rw-r--r-- 0/0             131 2017-01-19 21:47 03f0:032a
-rw-r--r-- 0/0              38 2019-11-25 20:48 03f0:0857
-rw-r--r-- 0/0              28 2017-08-04 21:41 03f0:371d
-rw-r--r-- 0/0              28 2017-08-04 21:41 03f0:4b1d
-rw-r--r-- 0/0              28 2017-08-04 21:42 03f0:4e1d
```

```
-rw-r--r-- 0/0              28 2016-07-25 21:36 03f0:521d
-rw-r--r-- 0/0              28 2016-07-25 21:39 03f0:531d
-rw-r--r-- 0/0              28 2016-07-25 21:39 03f0:541d
-rw-r--r-- 0/0              28 2016-07-25 21:39 03f0:581d
$ tar -tvf /usr/share/usb_modeswitch/configPack.tar.gz | wc -l
513
```

Текущая версия знает об очень многих *многофункциональных* устройствах, учитывая, что каждый файл здесь (с именем вида «VID:PID») может относиться к двум, трем и более близким устройствам одного производителя. Любую конфигурацию, например для показанного мною ранее устройства AnyDATA, можно посмотреть так:

```
$ tar -f /usr/share/usb_modeswitch/configPack.tar.gz "05c6:1000:uMa=AnyDATA" -x --to-stdout
# AnyDATA devices, Bless UC165
TargetVendor= 0x16d5
TargetProduct=0x6502
MessageContent="55534243123456780000000000000061b000000020000000000000000000000"
StandardEject=1
```

А обеспечивает запуск `usb_modeswitch` при «горячем» подключении устройства, как мы это сейчас уже понимаем, демон `udevd` в соответствии с его правилами из файла 40-usb_modeswitch.rules, который ищите в каталогах /etc/udev/rules.d или /usr/lib/udev/rules.d: получив известный VID:PID, демон запускает программу `usb_modeswitch`:

```
$ ls -l /usr/lib/udev/rules.d | grep usb_
-rw-r--r-- 1 root root  42861 фев 24  2020 40-usb_modeswitch.rules
$ head -n 15 /usr/lib/udev/rules.d/40-usb_modeswitch.rules
# Part of usb-modeswitch-data, version 20191128
#
# Works with usb_modeswitch versions >= 2.4.0. Slash before %k parameter
# is for compatibility only. Versions >= 2.5.0 don't need it.
#
ACTION!="add|change", GOTO="modeswitch_rules_end"

# Adds a symlink "gsmmodem[n]" to the lowest ttyUSB port with interrupt
# transfer; checked against a list of known modems, or else no action
KERNEL=="ttyUSB*", ATTRS{bNumConfigurations}=="*", PROGRAM="usb_modeswitch --symlink-name
%p %s{idVendor} %s{idProduct} %E{PRODUCT}", SYMLINK+="%c"

SUBSYSTEM!="usb", ACTION!="add",, GOTO="modeswitch_rules_end"

# Generic entry for most Huawei devices, excluding Android phones
ATTRS{idVendor}=="12d1", ATTRS{manufacturer}!="Android", ATTR{bInterfaceNumber}=="00",
ATTR{bInterfaceClass}=="08", RUN+="usb_modeswitch '%b/%k'"
```

... и т. д. — как показано в последней строке — для всех Vendor ID, известных системе, или в более ранних версиях (немного понятнее):

```
ACTION=="add" SUBSYSTEM=="usb", SYSFS{idProduct}=="1446", SYSFS{idVendor}=="12d1",
RUN+="/usr/sbin/usb_modeswitch"
```

Программа usb_modeswitch является программой *пользовательского* пространства и, как и следовало ожидать, использует библиотеки проекта libusb — обмена с USB-устройствами из пользовательского пространства (об этом инструменте будет коротко расссказано далее). Сейчас же для нас важно, что все манипуляции с многофункциональными устройствами мы можем выполнить *самостоятельно*!

Синтаксис записи самих правил usb_modeswitch (/etc/usb_modeswitch.d) в меру замысловат, но зато хорошо описан, и его детальное рассмотрение выходит за рамки нашего изложения. Принципиально важно лишь то, что, используя пакет usb_modeswitch, вы сможете записать правила его переключений *для любого* многофункционального устройства (стороннего или под свой проект).

Устройства в пространстве пользователя

Ряд функций управления реальным устройством может быть вообще выполнен в пространстве пользователя, не прибегая к технике написания модулей и не входя в адресное пространство ядра. Например, случаем, в котором работа в пространстве пользователя может иметь большой смысл, является ситуация, когда мы начинаем взамодействие с новым и необычным оборудованием. Работая таким образом, мы можем научиться управлять этим оборудованием без риска подвешивания системы в целом. После того как мы сделаем это, выделение нашего программного обеспечения в модуль ядра становится формальной задачей.

Может, конечно, создаться впечатление, что, обеспечивая обмен из пространства пользователя, можно создать *лишь* простейшие учебные приложения. Но это мнение опровергается наличием известных публичных проектов, использующих операции только из пространства пользователя. Вот некоторые примеры:

- употребляемая во всех UNIX-подобных операционных системах графическая подсистема X11. В своем *базовом варианте* (не расширенном проприетарными видеодрайверами пространства ядра) этот пакет не нуждается в модулях ядра: работа с портами и видеопамятью адаптеров успешно осуществляется из пространства пользователя, а аппаратные прерывания, которые могут генерироваться оборудованием (обратный ход кадра), система X11 не использует. Именно эта архитектура базовой системы X11 создала возможность такого легкого ее переноса в самые экзотические системы UNIX (как, например, MINIX 3) и такое широкое ее применение в мире UNIX;
- проект libusb — предоставление API для поддержки всего спектра возможных USB-устройств из пространства пользователя. Здесь смысл и возможности реализации основаны на других основаниях: низкоуровневый обмен с USB-устройством (требующий обработки прерываний и других возможностей ядра) обеспечивает базовый слой поддержки USB, встроенный в ядро. Сам же проект libusb является уже *надстройкой* пользовательского пространства, обращающейся к базовым возможностям посредством системных вызовов. Показателем признания качества libusb может быть тот факт, что этот API используют, в свою очередь, такие крупнейшие проекты, как: CUPS (Common Unix Printing

System — основная на сегодня подсистема печати в Linux), SANE (обслуживание сканеров), `fprint` (биометрическая система идентификации отпечатков пальцев), `libgphoto2` (обслуживание фотографических камер) — и это еще далеко не все...;

- файловая система FUSE (Filesystem in Userspace) — файловая система в пространстве пользователя. Идея состоит в том, что сама корневая файловая система располагается в ядре, но все файловые операции определяются в таблице в пространстве пользователя (подобно тому, как мы делали это ранее в модулях), и эта таблица экспортируется в ядро посредством API FUSE. Позже, при системном вызове к такой файловой системе, запросы «выбрасываются» ядром в пространство пользователя и предоставляются для реализации пользователю. Ядерная часть FUSE включена в состав ядра еще с версии 2.6.14. Идеи FUSE оказались так притягательны для производителей, что на ней сегодня строится программная поддержка большинства малых мобильных гаджетов: диктофонов, фотоаппаратов и др. Через FUSE реализуется в Linux поддержка (для переносимости) файловых систем и Windows: NTFS, ExFAT;
- сетевая система в пользовательском пространстве — предложенная Intel (около 2015 г.) система DPDK (Data Plane Development Kit), обеспечивающая обход «узких мест» в самом ядре (сетевой подсистеме) Linux при переходе к сетевым картам 10 Gb/s, получающим все более широкое распространение;
- подсистема ядра GPIO (General-Purpose Input/Output) и библиотеки пространства пользователя GPIO API, которые создают легкий путь манипулирования *дискретными* входами/выходами для связи с разнообразными внешними исполнительными устройствами в системах реального управления.

Достаточно много проектов (упомянутых здесь и еще больше неназванных) концентрируются на том, чтобы работу с периферийными устройствами *вынести из ядра*. И в этом есть серьезный резон! Начиная новый проект, со всей тщательностью изучите — нельзя ли требуемую работу с устройствами *реализовать без программирования в ядре*!

И хотя эта тема (пользовательского пространства) только косвенно касается программирования и использования поддержки ядра и драйверов, ее стоит проиллюстрировать хотя бы для того, чтобы показать существующие альтернативы — как можно решать проблемы периферийных устройств, не влезая в сложности программирования в ядре.

Аппаратные порты

Взгляните на реально присутствующие в системе порты, которые могут использоваться для обмена:

```
$ sudo cat /proc/ioports
0000-03af : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
```

```
  0050-0053 : timer1
  0060-0060 : keyboard
  0061-0061 : PNP0800:00
  0064-0064 : keyboard
  0070-0071 : rtc0
  0080-008f : dma page reg
  00a0-00a1 : pic2
  00c0-00df : dma2
  00f0-00ff : PNP0C04:00
    00f0-00ff : fpu
...
0d00-ffff : PCI Bus 0000:00
  e000-efff : PCI Bus 0000:01
    e000-e0ff : 0000:01:00.0
      e000-e0ff : r8169
  f000-f0ff : 0000:00:01.0
  f100-f10f : 0000:00:11.0
    f100-f10f : ahci
  f110-f113 : 0000:00:11.0
...
```

Реализацию ввода/вывода из адресного пространства *пользователя* можно реализовать несколькими способами. Первый из них — это задействовать функции ввода/вывода. Для обмена с портами предоставляется (<sys/io.h>) набор функций:

```
unsigned char inb(unsigned short int port);
unsigned short int inw(unsigned short int port);
unsigned int inl(unsigned short int port);
void outb(unsigned char value, unsigned short int port);
void outw(unsigned short int value, unsigned short int port);
void outl(unsigned int value, unsigned short int port);
```

Сразу же акцентируем внимание на том (об этом говорилось и ранее), что при внешней полной идентичности с аналогичными функциями ядра эти вызовы представляют собой совершенно другие реализации. Функции ядра размещены в коде ядра, а пользовательские функции реализуются инлайновыми ассемблерными вставками GCC (это тоже мельком обсуждалось ранее):

```
static __inline unsigned char inb(unsigned short int __port) {
   unsigned char _v;
   __asm__ __volatile__ ("inb %w1,%0":"=a" (_v):"Nd" (__port));
   return _v;
}
```

Из-за инлайн-способа определений при компиляции GCC должны быть обязательно включены опции оптимизации (как минимум `-O` или `-O2`). Но операции обращения к портам недопустимы для *непривилегированных* процессов (выполняющихся в кольце защиты 3, а также не запущенных от имени root) — позже мы обсудим, как это выглядит. Для совершения обменных операций возможно (<sys/io.h>) ото-

бразить диапазон портов ввода/вывода в пространство задачи и разрешить их использовать:

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

Вызов: `ioperm()` устанавливает биты привилегий для доступа к области портов ввода/вывода. В роли параметров здесь выступают:

- `from` — начальный номер порта области;
- `num` — число портов в области;
- `turn_on` — разрешить (`1`) или запретить (`0`) привилегированные операции.

Таким способом можно изменить привилегии только для *первых* `0x3ff` портов ввода/вывода, а если нужно получить тот же результат для всех 65536 портов, надо воспользоваться системным вызовом `iopl()`.

Изменить уровень привилегированности пользовательского процесса *в отношении ввода/вывода* (это вовсе не означает перевод пользовательского приложения в другое кольцо защиты) можно так:

```
int iopl(int level);
```

Вызов `iopl()` меняет уровень *привилегий ввода/вывода*: всем процессам, уровень кольца защиты которых *ниже или равен* (более привилегированные) `level`, будут разрешены привилегированные операции (в дополнение к неограниченному доступу к портам ввода/вывода, работа на высоком уровне привилегий также позволяет процессу отключать прерывания. Скорее всего, это приведет к сбою системы, поэтому такое решение небезопасно). Эти права наследуются через `fork()` и `execve()`.

Естественно, что для получения привилегий ввода/вывода любым из способов процесс должен обладать правами root. После получения привилегий процесс может выполнять упомянутые ранее операции обмена: `out*()` и `in*()`.

Еще один способ основан на том, что Linux отображает пространство аппаратных портов в файл:

```
$ ls -l /dev/port
crw-r----- 1 root kmem 1, 4 нояб. 23 14:31 /dev/port
```

Поэтому возможность состоит в том, чтобы:

1. Открыть (`fd = open()`) указанный файл (естественно, с правами root).
2. Сдвинуть указатель позиции в файле на требуемый порт: `lseek(fd, port, SEEK_SET)`.
3. Считать (`read(fd, &data, 1)`) или записать (`write(fd, &data, 1)`) данные из порта или в порт.

Смысл проделываемого в этом случае состоит в том, что мы *поручаем* ядру Linux выполнить для нас операции ввода/вывода. Этот способ, конечно, будет в работе намного медленнее, но он не требует ни получения дополнительно привилегий уровня защиты процессора (режима супервизора), что небезопасно, ни оптимизации при компиляции (что иногда нежелательно).

Помимо возможности ввода/вывода для таких программ, как правило, нужно предотвратить выгрузку страниц этой 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:

- `MCL_CURRENT` — локировать все страницы, которые на текущий момент отображены в адресное пространство процесса;
- `MCL_FUTURE` — локировать все страницы, которые будут отображаться в будущем в адресное пространство процесса.

Демонстрируем все сказанное на примере кода (см. папку user_io в сопровождающем книгу файловом архиве):

ioports.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/io.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>

#define PARPORT_BASE 0x378

void do_io(unsigned long addr) {
   unsigned char zero = 0, readout = 0;
   printf("\twriting: 0x%02x to 0x%lx\n", zero, addr);
   outb(zero, addr);
   usleep(1000);
   readout = inb(addr + 1);
   printf("\treading: 0x%02x from 0x%lx\n", readout, addr + 1);
}

void do_read_devport(unsigned long addr) {
   unsigned char zero = 0, readout = 0;
   int fd;
   printf("/dev/port :\n");
   if((fd = open("/dev/port", O_RDWR)) < 0) {
      perror("reading /dev/port method failed"); return;
   }
   if(addr != lseek(fd, addr, SEEK_SET)) {
      perror("lseek failed"); close(fd); return;
   }
```

```
   printf("\twriting: 0x%02x to 0x%lx\n", zero, addr);
   write(fd, &zero, 1);
   usleep(1000);
   read(fd, &readout, 1);
   printf("\treading: 0x%02x from 0x%lx\n", readout, addr + 1);
   close(fd);
   return;
}

void do_ioperm(unsigned long addr) {
   printf("ioperm :\n");
   if(ioperm(addr, 2, 1)) {
      perror("ioperm failed"); return;
   }
   do_io(addr);
   if(ioperm(addr, 2, 0)) perror("ioperm failed");
   return;
}

int iopl_level = 3;
void do_iopl(unsigned long addr) {
   printf("iopl :\n");
   if(iopl(iopl_level)) {
      perror("iopl failed"); return;
   }
   do_io(addr);
   if(iopl(0)) perror("ioperm failed");
}

int main(int argc, char *argv[]) {
   unsigned long addr = PARPORT_BASE;
   if(argc > 1)
      if(!sscanf(argv[ 1 ],"%lx", &addr)) {
         printf("illegal address: %s\n", argv[ 1 ]);
         return EXIT_FAILURE;
      };
   if(argc > 2) iopl_level = atoi(argv[ 2 ]);
   do_read_devport(addr);
   do_ioperm(addr);
   do_iopl(addr);
   return errno;
}
```

Результаты его выполнения:

```
$ sudo ./ioports
/dev/port :
        writing: 0x00 to 0x378
        reading: 0x78 from 0x379
```

```
ioperm :
        writing: 0x00 to 0x378
        reading: 0x78 from 0x379
iopl :
        writing: 0x00 to 0x378
        reading: 0x78 from 0x379
```

Особенности доступа

Если попытаться использовать обменные операции (`ioperm()` или `iopl()`), не получив привилегий, то программа немедленно прервется на этом операторе. Странным образом различается при этом выполнение с правами root и без них:

```
$ ./ioperm
    writing: 0x00 to 0x378
Ошибка сегментирования (стек памяти сброшен на диск)
$ echo $?
139
$ sudo ./ioperm
[sudo] пароль для olej:
    writing: 0x00 to 0x378
Ошибка сегментирования
$ echo $?
139
```

Код завершения программы (`139`) довольно странный, но в любом случае он настолько серьезный, что попытка выполнения фиксируется в системном журнале:

```
$ dmesg | tail -n2
[20314.668678] traps: ioperm[16406] general protection fault ip:555a836d91da sp:7fff124dbac0
error:0 in ioperm[555a836d9000+1000]
[20327.584689] traps: ioperm[16435] general protection fault ip:55eda2b3e1da sp:7ffd31c52230
error:0 in ioperm[55eda2b3e000+1000]
```

Функции обмена (`in*()`/`out*()`) вообще никоим образом не возвращают и не устанавливают код ошибочности своего выполнения. Например, при обращении к несуществующим аппаратным портам:

```
$ sudo ./ioports 200
/dev/port :
        writing: 0x00 to 0x200
        reading: 0xff from 0x201
ioperm :
        writing: 0x00 to 0x200
        reading: 0xff from 0x201
iopl :
        writing: 0x00 to 0x200
        reading: 0xff from 0x201
```

Из несуществующих портов читается `0xff` — это не сильно надежный признак, и присутствие портов, наверное, нужно контролировать отдельно — например, по содержимому /proc/ioports, как это показывалось ранее.

Замечание относительно iopl() и уровней привилегий

Если функции указать параметр больше 3, то она возвратит ошибку:

```
$ sudo ./ioports 378 4
...
iopl failed: Invalid argument
$ echo $?
22
```

А поскольку в Linux из 4 колец защиты процессора *x*86 (0, 1, 2, 3) использованы только 2 (0 — для ядра Linux, 3 — для пользовательского пространства), то для вызова `iopl()` имеет смысл только:

- `iopl(3)` — разрешить пользовательскому процессу привилегированные операции;
- `iopl(0)` — возвратить запрет на выполнение пользовательским процессом привилегированных операций.

Еще один вопрос: а нельзя ли (например, при отработке прототипов, о чем говорилось в начале главы) получить доступ из пользовательского пространства к DMA и прерываниям? Нельзя — никаких средств для этого в пользовательском API нет. Но для отработки прототипов можно создать простейший модуль обработки аппаратного прерывания, который по получении прерывания должен будет всего лишь отослать *сигнал UNIX* процессу пользовательского пространства (эта возможность будет обсуждена далее). Таким же образом может решаться вопрос асинхронных событий аппаратуры.

Проект *libusb*

Проект `libusb` — это настолько мощный, настолько широко использующийся в различных сторонних проектах инструментарий и настолько давно прижившийся в Linux проект, что он заслуживает отдельного рассмотрения. Показателем признания и качества `libusb` может служить уже упоминавшийся ранее факт, что этот API используют такие крупнейшие проекты, как: CUPS (**C**ommon **U**nix **P**rinting **S**ystem — основная на сегодня подсистема печати в Linux), SANE (подсистема обслуживания сканеров), `fprint` (биометрическая система идентификации отпечатков пальцев), `libgphoto2` (обслуживание фотографических камер), и многие другие. Проект `libusb` — мультиплатформенный, и на сегодня этот инструментарий представлен (переносимо) в операционных системах: Linux, BSDs, Solaris, OS X, Windows, Android, Haiku (хотя переносимость и не является предметом нашего текущего рассмотрения).

Библиотека `libusb`, как уже было сказано ранее, позволяет осуществлять практически все операции диагностики и ввода/вывода с USB-устройствами из *пользовательских* процессов. Это позволяет перенести часть (большую или меньшую) функциональности проектов, работающих с USB-устройствами, из модульного

кода в пользовательский. Так сделано во многих известных проектах, и такой подход поощряется.

> ***Примечание***
>
> С середины 2012 г. существует альтернативная линия развития этого проекта, которая может фигурировать под именем `libusbx`.

Смысл проекта в том, что весь низкоуровневый обмен с USB-устройством (требующий обработки прерываний и других возможностей ядра) оставлен в ядерной части, а библиотека `libusb` предоставляет (libusb.h) обертки пользовательского пространства, обращающиеся к базовым возможностям в ядре посредством системных вызовов. И работает это *в любой* процессорной архитектуре, поддерживаемой Linux!

Для работы с `libusb` вам придется, вероятно, установить стандартным способом своего дистрибутива пакет `libusb-*-dev`, потому что обычно в дистрибутивах Linux по умолчанию установлен сам пакет библиотеки, но не установлены средства разработки (libusb.h) под него. В конечном итоге это должно выглядеть как-то так:

```
$ aptitude search libusb | grep ^i
i A libusb-1.0-0 - userspace USB programming library
i  libusb-1.0-0-dev - userspace USB programming library development files
```

После этого мы можем воспользоваться утилитой для точного определения опций для компиляции и сборки:

```
$ pkg-config libusb-1.0 --cflags
-I/usr/include/libusb-1.0
$ pkg-config libusb-1.0 --libs
-lusb-1.0
```

Теперь настало время рассмотреть пример (см. папку libusb в сопровождающем книгу файловом архиве) диагностики USB-оборудования (заимствовано с минимальными переделками с сайта проекта `libusb`):

ulist.c

```
#include <stdio.h>
#include <sys/types.h>
#include <libusb.h>

static void print_devs(libusb_device **devs) {
   libusb_device *dev;
   int i = 0;
   while ((dev = devs[i++]) != NULL) {
      struct libusb_device_descriptor desc;
      uint8_t path[8];
      int r = libusb_get_device_descriptor(dev, &desc);
      if (r < 0) {
         fprintf(stderr, "failed to get device descriptor");
```

```
            return;
        }
        printf("%04x:%04x (bus %d, device %d)",
               desc.idVendor, desc.idProduct,
               libusb_get_bus_number(dev), libusb_get_device_address(dev));
        r = libusb_get_port_numbers(dev, path, sizeof(path));
        if (r > 0) {
            int j;
            printf(" path: %d", path[0]);
            for (j = 1; j < r; j++)
               printf(".%d", path[j]);
        }
        printf("\n");
    }
}

int main(void) {
    libusb_device **devs;
    int r;
    ssize_t cnt;

    r = libusb_init(NULL);
    if (r < 0)
        return r;
    cnt = libusb_get_device_list(NULL, &devs);
    if (cnt < 0)
        return (int) cnt;
    print_devs(devs);
    libusb_free_device_list(devs, 1);
    libusb_exit(NULL);
    return 0;
}
```

Приложение диагностирует все найденные в системе USB-устройства, пару VID:PID, характеризующую устройство, и физическую топологию устройств по их подключениям (порты). Если у вас в системе используется несколько устройств с одинаковыми VID:PID, то идентифицировать их по этому основному показателю нельзя, и тогда на помощь приходит физическая топология подключения (рис. 6.7).

Сборку приложений (этого и следующего) осуществляем так (этот вывод позволяет воспроизвести содержимое файла Makefile, применяемого при сборке):

```
$ make
gcc -O2 -Wall `pkg-config libusb-1.0 --cflags`  utest.c `pkg-config libusb-1.0 --libs` -o utest
gcc -O2 -Wall `pkg-config libusb-1.0 --cflags`  ulist.c `pkg-config libusb-1.0 --libs` -o ulist
```

В итоге мы получили исполнимое приложение (выполнение от root обязательно):

```
$ sudo ./ulist
046d:0a45 (bus 2, device 4) path: 1.8
```

```
2101:020f (bus 2, device 3) path: 1.2
8087:0024 (bus 2, device 2) path: 1
1d6b:0002 (bus 2, device 1)
0624:0249 (bus 1, device 4) path: 1.6.1
0624:0248 (bus 1, device 3) path: 1.6
8087:0024 (bus 1, device 2) path: 1
1d6b:0002 (bus 1, device 1)
```

Это можно сравнить для проверки с тем, что дает нам команда `lsusb` на этом компьютере (и оно совпадет), но не будем терять на это время...

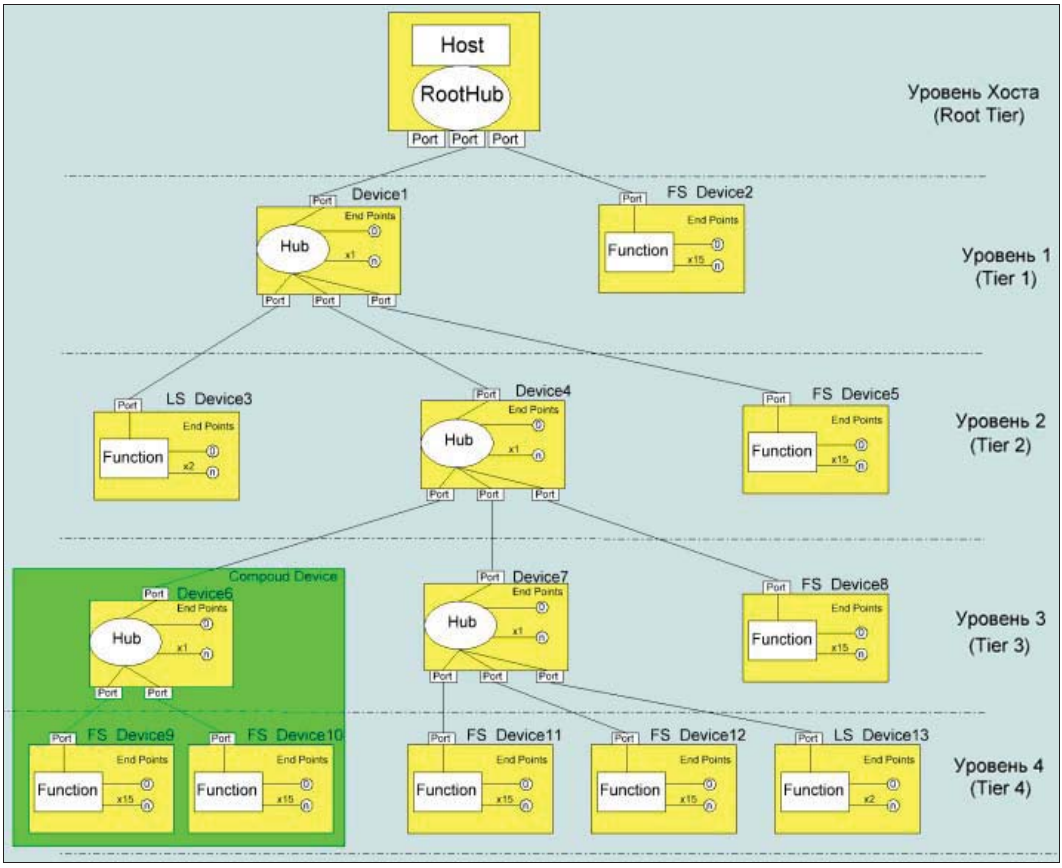


Рис. 6.7. Топология сети USB на физическом уровне

В следующем примере мы рассмотрим, как происходит обмен данными с USB-устройством, и здесь же покажем общую схему *любого* приложения, читающего из USB-устройства или пишущего в него. А поскольку нам нужно конкретное устройство для экспериментирования, выберем в этом качестве... мышь. Но на каждом компьютере у нас будет совершенно *конкретная и отличающаяся* конфигурация USB-устройств, даже таких простых, поэтому прежде мы соберем некоторую подготовительную информацию:

```
$ lsusb
Bus 004 Device 002: ID 0557:0204 ATEN International Co., Ltd
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 005 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

В показанной конфигурации у меня более сложный случай: и мышь, и клавиатура объединены на внешнем устройстве ATEN, и только затем через него подключены на единый USB-порт, — но это усложнение делает задачу только интереснее!

Приступая к задаче, детально каждый раз изучаем свое USB-устройство:

```
$ sudo lsusb -d 0557:0204 -v
Bus 004 Device 002: ID 0557:0204 ATEN International Co., Ltd
Device Descriptor:
  bLength                18
...
```

Но самые интересующие нас параметры, которые мы должны извлечь из этого вывода, — это номер интерфейса и адреса концевых точек обмена (EP, EndPoint):

```
$ lsusb -d 0557:0204 -v | grep bInterfaceNumber
Couldn't open device, some information will be missing
      bInterfaceNumber        0
      bInterfaceNumber        1
$ lsusb -d 0557:0204 -v | grep bEndpointAddress
Couldn't open device, some information will be missing
        bEndpointAddress     0x81  EP 1 IN
        bEndpointAddress     0x82  EP 2 IN
```

У вас их может оказаться больше (или даже меньше), но нам нужно найти те, которые в выводе отмечены как `Mouse` (нас в этой конфигурации будет интересовать пара `1`, `0x82`):

```
$ sudo lsusb -d 0557:0204 -v | grep -i mouse
      bInterfaceProtocol      2 Mouse
```

Теперь у нас есть вся информация для написания теста:

utest.c

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

int main (int argc, char *argv[]) {
   const uint16_t vendor_id = 0x0557;
   const uint16_t product_id = 0x0204;
```

```
#define SIZE 8
#define DEV_INTF 1 // bInterfaceNumber
#define EP 0x82    // bEndpointAddress
   struct libusb_device_handle *devh = NULL;
   unsigned char buf[SIZE];
   int ret, debug = 0, rep = 5;
   while ((ret = getopt(argc, argv, "v")) != -1) {
      switch (ret) {
         case 'v':
            debug++;
      }
   }
   if (argc > optind && atoi(argv[optind]) > 0)
      rep = atoi(argv[optind]);
   if ((ret = libusb_init(NULL)) != 0) {
      fprintf(stderr, "ошибка инициализации libusb: %d\n", ret);
      return 2;
   }
   if (debug > 0)  // уровень вывода отладочных сообщений
      libusb_set_option(NULL, LIBUSB_OPTION_LOG_LEVEL, LIBUSB_LOG_LEVEL_DEBUG);
   devh = libusb_open_device_with_vid_pid(NULL, vendor_id, product_id);
   if (devh == NULL) {
      fprintf(stderr, "такое устройство не подключено: %X:%X\n",
              vendor_id, product_id);
      libusb_exit(NULL);
      return 3;
   }
   if (libusb_kernel_driver_active(devh, DEV_INTF))
      libusb_detach_kernel_driver(devh, DEV_INTF);
   if (libusb_claim_interface(devh, DEV_INTF) < 0){
      fprintf(stderr, "ошибка интерфейса\n");
      return 4;
   }
   while (rep-- > 0) {
      int i,
         returned = libusb_interrupt_transfer(devh, EP, buf, SIZE, &ret, 5000);
      if (returned >= 0) {
         printf("read %d байт: ", ret);
         for (i = 0; i < ret; i++)
            printf("%d ", (int)buf[i]);
         printf("\n");
      }
      else {
                  printf("непонятно ... тайм-аут?\n");
           }
   }
```

```
    libusb_release_interface(devh, DEV_INTF);
    libusb_attach_kernel_driver(devh, DEV_INTF);
    libusb_close(devh);
    libusb_exit(NULL);
    return 0;
}
```

Запустим приложение **без** отладочной диагностики на то, чтобы «ловить» 25 считанных событий, и подвигаем мышью:

```
$ sudo ./utest 25
read = 4 байт: 1 0 0 0
read = 4 байт: 0 0 0 0
read = 4 байт: 2 0 0 0
read = 4 байт: 0 0 0 0
read = 4 байт: 0 1 0 0
read = 4 байт: 0 2 1 0
read = 4 байт: 0 3 2 0
read = 4 байт: 0 2 2 0
read = 4 байт: 0 2 1 0
read = 4 байт: 0 4 3 0
read = 4 байт: 0 5 4 0
read = 4 байт: 0 6 2 0
read = 4 байт: 0 8 3 0
read = 4 байт: 0 6 2 0
read = 4 байт: 0 6 3 0
read = 4 байт: 0 4 2 0
read = 4 байт: 0 2 0 0
read = 4 байт: 0 1 1 0
read = 4 байт: 0 254 0 0
read = 4 байт: 0 251 0 0
read = 4 байт: 0 248 0 0
read = 4 байт: 0 247 255 0
read = 4 байт: 0 245 254 0
read = 4 байт: 0 245 254 0
read = 4 байт: 0 248 254 0
```

Здесь есть и нажатия кнопок: первая колонка, и движения по вертикали и горизонтали...

Процессы, происходящие с USB, достаточно сложные, и для разбирательства с ними `libusb` предоставляет несколько уровней (`LIBUSB_LOG_LEVEL_DEBUG`, `LIBUSB_LOG_LEVEL_INFO` и др.) отладочного вывода (`LIBUSB_OPTION_LOG_LEVEL`). Вот как выглядит выполнение того же приложения с уровнем наивысшей детализации (`-v`) для пяти считанных событий:

```
$ sudo ./utest -v
[timestamp] [threadID] facility level [function call] <message>
--------------------------------------------------------------------------------
[ 0.026655] [00003195] libusb: debug [libusb_get_device_list]
[ 0.026808] [00003195] libusb: debug [libusb_get_device_descriptor]
```

```
[ 0.026832] [00003195] libusb: debug [libusb_open] open 4.2
[ 0.026953] [00003195] libusb: debug [usbi_add_pollfd] add fd 9 events 4
[ 0.026998] [00003195] libusb: debug [libusb_kernel_driver_active] interface 1
[ 0.027037] [00003195] libusb: debug [libusb_detach_kernel_driver] interface 1
[ 0.045896] [00003195] libusb: debug [libusb_claim_interface] interface 1
[ 0.046091] [00003195] libusb: debug [libusb_alloc_transfer] transfer 0x21dd438
[ 0.046157] [00003195] libusb: debug [libusb_submit_transfer] transfer 0x21dd438
[ 0.046231] [00003195] libusb: debug [add_to_flying_list] arm timerfd for timeout in 5000ms
(first in line)
[ 0.046307] [00003195] libusb: debug [submit_bulk_transfer] need 1 urbs for new transfer with
length 8
[ 0.046418] [00003195] libusb: debug [libusb_handle_events_timeout_completed] doing our own
event handling
[ 0.046466] [00003195] libusb: debug [handle_events] poll fds modified, reallocating
[ 0.046540] [00003195] libusb: debug [handle_events] poll() 3 fds with timeout in 60000ms
[ 1.996556] [00003195] libusb: debug [handle_events] poll() returned 1
[ 1.996669] [00003195] libusb: debug [reap_for_handle] urb type=1 status=0 transferred=4
[ 1.996700] [00003195] libusb: debug [handle_bulk_completion] handling completion status 0
of bulk urb 1/1
[ 1.996729] [00003195] libusb: debug [handle_bulk_completion] last URB in transfer -->
complete!
[ 1.996755] [00003195] libusb: debug [disarm_timerfd]
[ 1.996797] [00003195] libusb: debug [usbi_handle_transfer_completion] transfer 0x21dd438 has
callback 0xb6f4ad95
[ 1.996825] [00003195] libusb: debug [sync_transfer_cb] actual_length=4
[ 1.996885] [00003195] libusb: debug [libusb_free_transfer] transfer 0x21dd438
read = 4 байт: 0 1 0 0
[ 1.997142] [00003195] libusb: debug [libusb_alloc_transfer] transfer 0x21ed3c8
[ 1.997173] [00003195] libusb: debug [libusb_submit_transfer] transfer 0x21ed3c8
...
[ 2.012711] [00003195] libusb: debug [handle_bulk_completion] last URB in transfer -->
complete!
[ 2.012735] [00003195] libusb: debug [disarm_timerfd]
[ 2.012764] [00003195] libusb: debug [usbi_handle_transfer_completion] transfer 0x21ed3c8 has
callback 0xb6f4ad95
[ 2.012789] [00003195] libusb: debug [sync_transfer_cb] actual_length=4
[ 2.012814] [00003195] libusb: debug [libusb_free_transfer] transfer 0x21ed3c8
read = 4 байт: 0 2 0 0
[ 2.012864] [00003195] libusb: debug [libusb_alloc_transfer] transfer 0x21ecf88
[ 2.012884] [00003195] libusb: debug [libusb_submit_transfer] transfer 0x21ecf88
[ 2.012903] [00003195] libusb: debug [add_to_flying_list] arm timerfd for timeout in 5000ms
(first in line)
...
[ 2.028742] [00003195] libusb: debug [usbi_handle_transfer_completion] transfer 0x21ecf88 has
callback 0xb6f4ad95
[ 2.028775] [00003195] libusb: debug [sync_transfer_cb] actual_length=4
[ 2.028802] [00003195] libusb: debug [libusb_free_transfer] transfer 0x21ecf88
read = 4 байт: 0 2 3 0
```

```
[ 2.028854] [00003195] libusb: debug [libusb_alloc_transfer] transfer 0x21f10c0
[ 2.028876] [00003195] libusb: debug [libusb_submit_transfer] transfer 0x21f10c0
[ 2.028896] [00003195] libusb: debug [add_to_flying_list] arm timerfd for timeout in 5000ms
(first in line)
...
[ 2.036914] [00003195] libusb: debug [usbi_handle_transfer_completion] transfer 0x21f10c0 has
callback 0xb6f4ad95
[ 2.036987] [00003195] libusb: debug [sync_transfer_cb] actual_length=4
[ 2.037049] [00003195] libusb: debug [libusb_free_transfer] transfer 0x21f10c0
read = 4 байт: 0 2 3 0
[ 2.037137] [00003195] libusb: debug [libusb_alloc_transfer] transfer 0x21f0ef0
[ 2.037167] [00003195] libusb: debug [libusb_submit_transfer] transfer 0x21f0ef0
[ 2.037201] [00003195] libusb: debug [add_to_flying_list] arm timerfd for timeout in 5000ms
(first in line)
...
[ 2.052863] [00003195] libusb: debug [usbi_handle_transfer_completion] transfer 0x21f0ef0 has
callback 0xb6f4ad95
[ 2.052886] [00003195] libusb: debug [sync_transfer_cb] actual_length=4
[ 2.052916] [00003195] libusb: debug [libusb_free_transfer] transfer 0x21f0ef0
read = 4 байт: 0 2 5 0
[ 2.053034] [00003195] libusb: debug [libusb_release_interface] interface 1
[ 2.053156] [00003195] libusb: debug [libusb_attach_kernel_driver] interface 1
[ 2.060124] [00003195] libusb: debug [libusb_close]
[ 2.060231] [00003195] libusb: debug [usbi_remove_pollfd] remove fd 9
[ 2.060294] [00003195] libusb: debug [libusb_exit]
[ 2.060320] [00003195] libusb: debug [libusb_exit] destroying default context
```

На этом мы закончим наш краткий экскурс в такую интересную тему, как использование `libusb`. Хотелось бы надеяться, что изложения основных принципов достаточно, чтобы *начать* писать код под `libusb`. Все остальное — в документации!

Еще одним несомненным достоинством, спонтанно возникшим в ходе долгой жизни проекта и участия в нем большого числа сторонних заинтересованных авторов, явилось то, что для `libusb` были созданы языковые обертки, позволяющие использовать этот API работы с USB-устройствами из кода на самых различных языках программирования. На сегодня это как минимум: C и C++, Java, C#, Go, Ada, Python, Perl, Ruby, Lua, Common Lisp, Ocaml, Haskell. Например, используя в Python расширение PyUSB (одно из нескольких, доступных для Python!), мы могли бы писать непосредственно из Python-кода что-то типа следующего:

```
import usb.core
import usb.util as util
...
VID = 0x04d9
PID = 0x8010
dev = usb.core.find(idVendor=VID, idProduct=PID)
if dev.is_kernel_driver_active(0):
   dev.detach_kernel_driver(0)
```

```
dev.set_configuration() # do reset and set active conf. Must be done before claim.
util.claim_interface(dev, None)
try:
   dev.read(0x81, 64)
except usb.core.USBError:
   pass
...
```

GPIO

С некоторых, уже и не таких близких пор производители *одноплатных* компьютеров (независимо от их процессорной архитектуры) стали оснащать их чипами (и внешними разъемами) подсистемы GPIO (General-Purpose Input/Output) — особенно для компьютеров промышленного назначения. Подсистема GPIO позволяет компьютеру управлять определенным числом *дискретных* линий для электрического управления (высокое/низкое напряжение на линии) реальным нестандартным оборудованием в проектах автоматического управления. Эта возможность относится не только к минимальным («игрушечным») образцам — на сегодня предлагается уже очень широкий спектр серьезных моделей для таких возможностей (см., например, рис. 6.8).

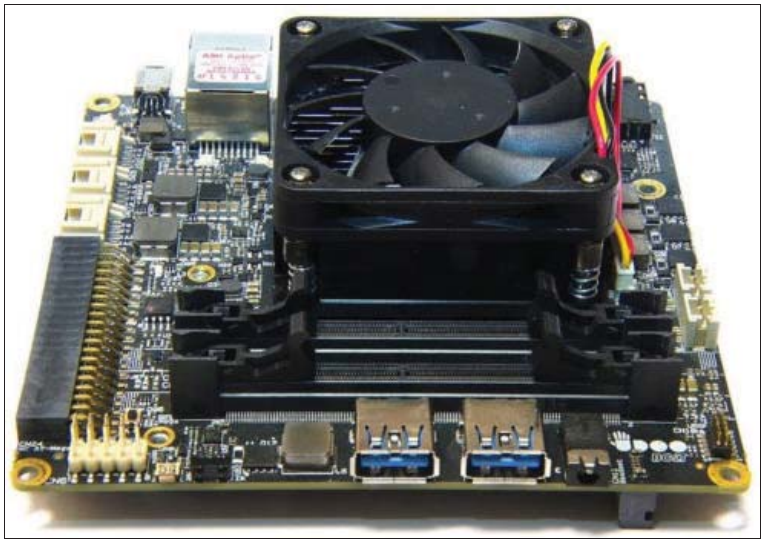


Рис. 6.8. Одна из множества доступных моделей — AMD Ryzen Embedded SBC с 4-ядерным процессором AMD V1605B, оперативной памятью до 64 Гбайт, флеш-памятью eMMC на 32 Гбайт, разъемами M.2 и SATA для подключения устройств хранения и поддержкой до четырех дисплеев 4K через порты HDMI 2.0

Подсистема GPIO не относится напрямую к нашему предмету — написанию модулей ядра, но для нее существует отдельный API ядра (префикс имен вида `gpio_*`), а сама она настолько значима в плане замены некоторых ядерных возможностей, что ее рассмотрение нельзя оставить в стороне.

Линии GPIO идентифицируются целыми числами без знака в диапазоне `0...MAX_INT` — на разных платформах эти номера (диапазоны) конкретно могут радикально отличаться.

В версии 2.6.26 в Linux появляется стандартный интерфейс для работы с GPIO через sysfs (называемый *старым* — не в смысле устаревшим, а появившимся раньше). В /sys это каталог:

```
$ ls -l /sys/class/gpio
итого 0
--w--w---- 1 root gpio 4096 авг 25 14:14 export
lrwxrwxrwx 1 root gpio    0 авг 25 14:14 gpiochip0 -> ...
--w--w---- 1 root gpio 4096 авг 25 14:14 unexport
```

В этом экземпляре описан один контроллер (чип) GPIO (может быть и несколько). Число контролируемых линий в нашем случае:

```
$ cat /sys/class/gpio/gpiochip*/ngpio
54
```

Для того чтобы взять под контроль любую из наличествующих линий, заносим ее номер (в примере `16`) в `export` (все операции с GPIO требуют прав root):

```
# echo 16 > /sys/class/gpio/export
# ls -l /sys/class/gpio
итого 0
--w--w---- 1 root gpio 4096 авг 25 21:05 export
lrwxrwxrwx 1 root root    0 авг 25 21:05 gpio16 -> ...
lrwxrwxrwx 1 root gpio    0 авг 25 14:14 gpiochip0 -> ...
--w--w---- 1 root gpio 4096 авг 25 14:14 unexport
```

Теперь через вновь появившийся каталог мы ведем работу с этой **линией**:

```
# tree /sys/class/gpio/gpio16
/sys/class/gpio/gpio16
├── active_low
├── device -> ../../../gpiochip0
├── direction
├── edge
├── power
│   ├── autosuspend_delay_ms
│   ├── control
│   ├── runtime_active_time
│   ├── runtime_status
│   └── runtime_suspended_time
├── subsystem -> ../../../../../../../class/gpio
├── uevent
└── value
3 directories, 10 files
```

Через `direction` мы управляем *направлением* работы линии: вывод или ввод данных. Изначально направление установлено на ввод:

```
# cat /sys/class/gpio/gpio16/direction
in
```

Задав направление, мы можем выводить управляющее воздействие и считывать данные с линии (данные *дискретной* линии, естественно, могут быть только 0 или 1):

```
# echo out > /sys/class/gpio/gpio16/direction
# cat /sys/class/gpio/gpio16/direction
out
# cat /sys/class/gpio/gpio16/value
0
# echo 1 > /sys/class/gpio/gpio16/value
# cat /sys/class/gpio/gpio16/value
1
```

Когда нам не нужна больше используемая линия — записываем ее номер в unexport:

```
# echo 16 > /sys/class/gpio/unexport
# ls -l /sys/class/gpio
итого 0
--w--w---- 1 root gpio 4096 авг 25 21:05 export
lrwxrwxrwx 1 root gpio    0 авг 25 14:14 gpiochip0 -> ...
--w--w---- 1 root gpio 4096 авг 25 21:10 unexport
```

Восстановилось исходное состояние!

Показанного минимального объема вполне хватает, чтобы понять, что этого *интерфейса* достаточно для построения *управляющих* систем на языках Shell, C/C++, Python практически любым инструментарием. Позже появился и набор утилит (устанавливаются из стандартного репозитория дистрибутива с пакетом gpiod), реализующих CLI-интерфейс в GPIO:

```
$ ls /bin/*gpio*
/bin/gpiodetect  /bin/gpiofind  /bin/gpioget  /bin/gpioinfo  /bin/gpiomon  /bin/gpioset
$ ls /usr/bin/*gpio*
/usr/bin/gpiodetect  /usr/bin/gpiofind  /usr/bin/gpioget  /usr/bin/gpioinfo  /usr/bin/gpiomon
/usr/bin/gpioset
```

Вот некоторые примеры использования этого интерфейса (это уже другого типа микрокомпьютер):

```
# gpiodetect
gpiochip0 [1c20800.pinctrl] (224 lines)
gpiochip1 [1f02c00.pinctrl] (32 lines)
# gpioinfo gpiochip1
gpiochip1 - 32 lines:
    line   0:      unnamed       unused   input  active-high
    line   1:      unnamed       unused   input  active-high
    line   2:      unnamed  "usb0-vbus"  output  active-high [used]
    line   3:      unnamed        "sw4"   input   active-low [used]
    line   4:      unnamed       unused   input  active-high
    line   5:      unnamed       unused   input  active-high
```

```
    line   6:      unnamed   "vdd-cpux"  output  active-high [used]
    line   7:      unnamed       unused   input  active-high
    line   8:      unnamed       unused   input  active-high
    line   9:      unnamed       unused   input  active-high
    line  10:      unnamed "orangepi:green:pwr" output active-high [used]
    line  11:      unnamed       unused   input  active-high
    line  12:      unnamed       unused   input  active-high
    line  13:      unnamed       unused   input  active-high
    line  14:      unnamed       unused   input  active-high
    line  15:      unnamed       unused   input  active-high
    line  16:      unnamed       unused   input  active-high
    line  17:      unnamed       unused   input  active-high
    line  18:      unnamed       unused   input  active-high
    line  19:      unnamed       unused   input  active-high
    line  20:      unnamed       unused   input  active-high
    line  21:      unnamed       unused   input  active-high
    line  22:      unnamed       unused   input  active-high
    line  23:      unnamed       unused   input  active-high
    line  24:      unnamed       unused   input  active-high
    line  25:      unnamed       unused   input  active-high
    line  26:      unnamed       unused   input  active-high
    line  27:      unnamed       unused   input  active-high
    line  28:      unnamed       unused   input  active-high
    line  29:      unnamed       unused   input  active-high
    line  30:      unnamed       unused   input  active-high
    line  31:      unnamed       unused   input  active-high
# gpioset gpiochip1  11=1
# gpioget gpiochip1 11
1
# gpioset gpiochip1 11=0
# gpioget gpiochip1 11
0
```

Еще позже (ядро 4.8) появляется (дополнительно) новый интерфейс, реализуемый модулями ядра в составе дерева исходных кодов ядра и представляющий контроллеры GPIO как символьные устройства в /dev, а интерфейс к ним — как операции `ioctl()`:

```
$ ls -l /dev/gpio*
crw------- 1 root root 254, 0 авг 25 17:17 /dev/gpiochip0
crw------- 1 root root 254, 1 авг 25 17:17 /dev/gpiochip1
```

Этот интерфейс включает (<linux/gpio.h>) обширный API (коды вида `GPIO_*_IOCTL`) пространства *пользователя*, предоставляющий пользовательским приложениям весь спектр операций по диагностике и управлению подсистемой GPIO.

В завершение в качестве иллюстрации приведем пример использования этого API для диагностики устройств GPIO (см. папку GPIO в сопровождающем книгу файловом архиве):

info.c

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <getopt.h>
#include <linux/gpio.h>

#define DEV_NAME "/dev/gpiochip0"

int main(int argc, char* argv[]) {
   int fd, ret, debug = 0;
   struct gpiochip_info info;
   struct gpioline_info line_info;
   char dev_name[80];
   while ((ret = getopt(argc, argv, "v")) != -1) {
      switch (ret) {
         case 'v':
            debug++;
            break;
         default:
            fprintf(stderr,
                    "Usage: %s options dev_name.\n"
                    "Options:\n"
                    "\t -v increase output level\n",
                    argv[0]);
            return -1;
      }
   }
   strcat(dev_name, 1 == argc ? DEV_NAME : argv[optind]);
   fd = open(dev_name, O_RDONLY);
   if (fd < 0) {
      printf("Unabled to open %s: %s\n", dev_name, strerror(errno));
      return 1;
   }
   // Query GPIO chip information
   ret = ioctl(fd, GPIO_GET_CHIPINFO_IOCTL, &info);
   if (ret == -1) {
      printf("Unable to get chip info from ioctl: %s", strerror(errno));
      close(fd);
      return 2;
   }
```

```
   printf("Chip name: %s\n", info.name);
   printf("Chip label: %s\n", info.label);
   printf("Number of lines: %d\n", info.lines);
   if (0 == debug)
      return 0;
   for (int i = 0; i < info.lines; i++) {
      line_info.line_offset = i;
      ret = ioctl(fd, GPIO_GET_LINEINFO_IOCTL, &line_info);
      if (ret == -1) {
         printf("Unable to get line info from offset %d: %s", i, strerror(errno));
      }
      else {
         printf("offset: %d, name: %s, consumer: %s. Flags:\t[%s]\t[%s]\t[%s]\t[%s]\t[%s]\n",
                i,
                line_info.name,
                line_info.consumer,
                (line_info.flags & GPIOLINE_FLAG_IS_OUT) ? "OUTPUT" : "INPUT",
                (line_info.flags & GPIOLINE_FLAG_ACTIVE_LOW) ? "ACTIVE_LOW" : "ACTIVE_HIGHT",
                (line_info.flags & GPIOLINE_FLAG_OPEN_DRAIN) ? "OPEN_DRAIN" : "...",
                (line_info.flags & GPIOLINE_FLAG_OPEN_SOURCE) ? "OPENSOURCE" : "...",
                (line_info.flags & GPIOLINE_FLAG_KERNEL) ? "KERNEL" : "");
      }
   }
   (void)close(fd);
   return 0;
}
```

И теперь мы можем из своего приложения получить диагностику, которую уже раньше видели из утилиты `gpioinfo`:

```
$ sudo ./info -v /dev/gpiochip1
Chip name: gpiochip1
Chip label: 1f02c00.pinctrl
Number of lines: 32
offset: 0, name: , consumer: . Flags:   [INPUT] [ACTIVE_HIGHT]  [...]   [...]   []
offset: 1, name: , consumer: . Flags:   [INPUT] [ACTIVE_HIGHT]  [...]   [...]   []
offset: 2, name: , consumer: usb0-vbus. Flags:  [OUTPUT] [ACTIVE_HIGHT]  [...]   [...]   [KERNEL]
offset: 3, name: , consumer: sw4. Flags:        [INPUT] [ACTIVE_LOW]    [...]   [...]   [KERNEL]
offset: 4, name: , consumer: . Flags:   [INPUT] [ACTIVE_HIGHT]  [...]   [...]   []
offset: 5, name: , consumer: . Flags:   [INPUT] [ACTIVE_HIGHT]  [...]   [...]   []
```

В папке GPIO сопровождающего книгу файлового архива представлено приложение `allops` (с моими минимальными правками), которое позволяет выполнять весь спектр операций с GPIO (диагностика, чтение, запись):

```
$ ./allops
Usage: ./allops options dev_name.
```

```
Options:
     -l: print gpio chip info
     -r <offset>: Read GPIO value at offset (INPUT mode)
     -w <offset>: Write GPIO value at offset (OUTPUT mode). Option -v should be set
     -v <0|1>: value that should be written to the GPIO, used only with option -w
     -p <offset>: Polling raising event on the GPIO at offset
$ sudo ./allops -r11 /dev/gpiochip1
Value of GPIO at offset 11 (INPUT mode) on chip /dev/gpiochip1: 0
$ sudo ./allops -w11 -v1 /dev/gpiochip1
Write value 1 to GPIO at offset 11 (OUTPUT mode) on chip /dev/gpiochip1
$ sudo ./allops -r11 /dev/gpiochip1
Value of GPIO at offset 11 (INPUT mode) on chip /dev/gpiochip1: 1
```

- Глава 7 -
Расширенные возможности программирования

В правильном вопросе всегда уже скрыт ответ,
и в правильном поиске уже угадывается искомое.
Протоиерей Андрей Ткачев

Существует великое многообразие возможностей для программиста, пишущего в адресном пространстве ядра. На практике не все они востребованы одинаково широко, некоторые используются заметно реже, чем типовые средства, с которыми мы разбирались в предшествующих главах. Но все эти более изощренные варианты весьма значимы для детального понимания происходящего в ядре, а поэтому хотя бы некоторые из них должны быть коротко рассмотрены.

> ***Примечание***
>
> Многие из таких возможностей (реализующие действия, подобные аналогичным таким же операциям в пространстве пользователя, в более привычном контексте), в литературе и обсуждениях по ядру относятся к общей группе API под названием «хелперы» (или «хелперы пространства пользователя»), где информацию о них и следует искать.

Большинство возможностей, о которых здесь пойдет речь, редко и скупо описываются в публикациях и упоминаются в обсуждениях. Но информацию о реализации и использовании подобных возможностей вы всегда можете получить путем *экспериментирования* с тестирующим кодом над «живой» операционной системой.

И вот что важно отметить, прежде чем перейти к делу:

- все примеры кода этой главы проверялись и отлаживались на работоспособность. Там, где по тексту *не оговорено обратное* (а таких мест достаточно много), коды сопровождались до версии ядра 3.8–3.13 и зачастую в 32-битной реализации. Вести дальше сопровождение работоспособности не имело смысла: все эти возможности — на уровне *хакинга*, они работоспособны в текущей версии ядра, но теряют работоспособность уже в следующей подверсии из-за изменений в API и требуют корректировки... хотя зачастую и *незначительной*;
- другая причина некорректности выражается в том, что безошибочные коды в следующей версии перестают *компилироваться* из-за ошибок. Причин за годы наблюдения здесь выявлено несколько: а) изменение *прототипа* вызовов API (параметры, их число, типы, последовательность...); б) *перенос* групп API из одного хедер-файла определений в другой; в) *исключение* некоторых вызовов API

или даже целых их групп и г) *замена* их на другие эквиваленты. **Все** коды этой главы, хотя многие и не выверялись повторно на *исполнимость*, выверены и *откорректированы* в смысле проведения сборки — всё *компилируется* без ошибок (корректировка производилась под ядра уровня 5.4–5.15).

И в завершение этого небольшого предисловия: коды, приведенные в этой главе, описываются (и детально) не в качестве материала для выполнения, а как *подтверждение* того, что даже такие сложные реализации — на уровне *хакинга* — возможны и представляют собой *образец* того, как это можно сделать.

Операции с файлами данных

Операции с данными в именованных файлах (разных: регулярных файлах, FIFO и др.) не относятся к тем возможностям, которыми код ядра (модуля) должен *активно* пользоваться, — для этого не видно оснований[1] (то же, например, касается и операций с абсолютным хронологическим временем). Но, во-первых, такие операции вполне возможны, а во-вторых, существует как минимум одна ситуация, когда они насущно необходимы, — это чтение конфигурационных данных модуля (при запуске) из его конфигурационных файлов. Как будет показано далее, такие возможности не только осуществимы из кода ядра — они, более того, могут реализовываться несколькими альтернативными способами.

Рассмотрим это на примерах (приведенных в папке file сопровождающего книгу файлового архива):

mod_read.c

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

static char* file = NULL;
module_param(file, charp, 0);

#define BUF_LEN 255
#define DEFNAME "/etc/apt/sources.list";

static char buff[BUF_LEN + 1] = DEFNAME;

static int __init kread_init(void) {
    struct file *f;
    loff_t pos = 0;
    if(file != NULL) strcpy(buff, file);
    printk("+++ openning file: %s\n", buff);
    f = filp_open(buff, O_RDONLY, 0);
```

[1] Попытка модуля ядра активно работать с файлами данных уже должна настораживать — как возможный признак плохой архитектурной проработки ведущегося проекта.

```
    if(IS_ERR(f)) {
        printk("+++ file open failed: %s\n", buff);
        return -ENOENT;
    }
    while(1) {
        #define SIZE 80
        char data[SIZE + 1];
        size_t n;
        // ssize_t kernel_read(struct file *file, void *buf, size_t count, loff_t *pos);
        n = kernel_read(f, data, SIZE, &pos);
        if(0 == n) {
            printk("+++ read EOF\n");
            break;
        } else if(n >= 0) {
            data[n] = '\0';
            printk("+++ read %d bytes [%d]:\n%s\n",
                   (int)n, (int)pos, data);
        } else {
            printk("+++ kernel_read failed\n");
            filp_close(f, NULL);
            return -EIO;
        }
    }
    printk("+++ close file\n");
    filp_close(f, NULL);
    return -EPERM;
}

module_init( kread_init );
MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_VERSION("2.1");
MODULE_LICENSE("GPL");
```

Теперь посмотрим, как это работает (используемый в примере текстовый файл ./xxx был подготовлен заранее и содержит несколько строк текста, а файл с именем ./yyy отсутствует и указан в примере как недозволенное имя файла):

```
$ cat xxx
*1 .........
*2 .........
*3 .........
$ cat xxx | wc -l
3
```

Читаем этот тестовый файл из модуля ядра:

```
$ sudo insmod mod_read.ko file=xxx
insmod: ERROR: could not insert module mod_read.ko: Operation not permitted
$ dmesg | tail -n8
```

```
[27769.984601] +++ openning file: xxx
[27769.984620] +++ read 42 bytes [42]:
               *1 .........
               *2 .........
               *3 .........

[27769.984621] +++ read EOF
[27769.984621] +++ close file
```

Пытаемся читать несуществующий файл — обратите внимание, как изменился код ошибки загрузки модуля:

```
$ sudo insmod mod_read.ko file=yyy
insmod: ERROR: could not insert module mod_read.ko: Unknown symbol in module
$ dmesg | tail -n2
[28210.334800] +++ openning file: yyy
[28210.334814] +++ file open failed: yyy
```

И для полноты картины прочитаем файл из каталога конфигураций /etc. Это уже ближе к реалиям — с полным путевым именем в файловой системе:

```
$ cat /etc/apt/sources.list
#deb cdrom:[Linux Mint 20.1 _Ulyssa_ - Release amd64 20210103]/ focal contrib main
# This system was installed using small removable media
# (e.g. netinst, live or single CD). The matching "deb cdrom"
# entries were disabled at the end of the installation process.
# For information about how to configure apt package sources,
# see the sources.list(5) manual.
$ sudo insmod mod_read.ko file=/etc/apt/sources.list
insmod: ERROR: could not insert module mod_read.ko: Operation not permitted
$ dmesg | tail -n19
[26581.320357] +++ openning file: /etc/apt/sources.list
[26581.320365] +++ read 80 bytes [80]:
               #deb cdrom:[Linux Mint 20.1 _Ulyssa_ - Release amd64 20210103]/ focal contrib ma
[26581.320366] +++ read 80 bytes [160]:
               in
               # This system was installed using small removable media
               # (e.g. netinst, live
[26581.320367] +++ read 80 bytes [240]:
                or single CD). The matching "deb cdrom"
               # entries were disabled at the end of t
[26581.320368] +++ read 80 bytes [320]:
               he installation process.
               # For information about how to configure apt package so
[26581.320369] +++ read 41 bytes [361]:
               urces,
               # see the sources.list(5) manual.

[26581.320369] +++ read EOF
[26581.320370] +++ close file
```

Чтение «рваное», потому что мы читаем небольшими отрезками по длине ограниченного буфера (80 байтов) чтобы не усложнять код, но нам этого более чем достаточно. Это тот же файл, который показывает нам системная утилита `cat`.

Обратная операция — запись в файл из модуля — скорее всего, еще более редко востребованная операция, чем чтение. Но и она, во-первых, совершенно возможна, а во-вторых, бывает полезна, например, для протоколирования событий и особенно в целях отладки. Следующий пример демонстрирует такую возможность:

mod_write.c

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

static char* log = NULL;
module_param(log, charp, 0);

#define BUF_LEN 255
#define DEFLOG "./module.log"
#define TEXT "..............\n"

static int __init kread_init(void) {
    struct file *f;
    ssize_t n = 0;
    loff_t offset = 0;
    char buff[BUF_LEN + 1] = DEFLOG;
    if(log != NULL) strcpy(buff, log);
    printk("+++ openning file: %s\n", buff);
    f = filp_open(buff,
                O_CREAT | O_RDWR | O_TRUNC,
                S_IRUSR | S_IRGRP | S_IROTH | S_IWUSR);
    if(IS_ERR(f)) {
        printk("+++ file open failed: %s\n", buff);
        return -ENOENT;
    }
    printk("+++ file open %s\n", buff);
    strcpy(buff, TEXT);
    // extern ssize_t kernel_write(struct file *, const void *, size_t, loff_t *);
    if((n = kernel_write(f, buff, strlen(buff), &offset)) != strlen(buff)) {
        printk("! failed to write: %d\n", (int)n);
        return -EIO;
    }
    printk("+++ write %d bytes\n", (int)n);
    printk("+++ close file\n");
    filp_close(f, NULL);
    return -EPERM;
}
```

```
module_init( kread_init );
MODULE_AUTHOR("Oleg Tsiliuric");
MODULE_VERSION("2.1");
MODULE_LICENSE("GPL");
```

Вот как будет выглядеть работа этого кода:

```
$ sudo insmod mod_write.ko log=xxxx.log
insmod: ERROR: could not insert module mod_write.ko: Operation not permitted
$ dmesg | tail -n4
[27016.083315] +++ openning file: xxxx.log
[27016.083375] +++ file open xxxx.log
[27016.083398] +++ write 16 bytes
[27016.083399] +++ close file
$ ls -l xxxx.log
-rw-r--r-- 1 root root 16 авг 18 17:33 xxxx.log
$ cat xxxx.log
................
```

Обращаем внимание на то, что создаваемый и записываемый файл журнала xxxx.log, независимо от того, от имени какого пользователя мы собирались загружать модуль, создается от имени владельца (root), — код модуля исполняется из ядра и только от этого имени.

Предыдущие примеры использовали специальный вызов ядра `kernel_read()`, предназначенный только для такой цели. Но в ядре имеется более высокий уровень абстракции API ядра: имена (вызовы), экспортируемые *виртуальной файловой системой* (VFS), — вызовы вида `vfs_*()`. Наши вызовы файловых операций из пространства пользователя и обслуживаются этим набором операций, который гораздо шире:

```
$ sudo cat /proc/kallsyms | grep ' T ' | grep ' vfs_'
ffffffff9681c1c0 T vfs_fadvise
ffffffff968c7840 T vfs_fallocate
ffffffff968c84e0 T vfs_truncate
ffffffff968c9940 T vfs_open
ffffffff968c9f50 T vfs_setpos
ffffffff968c9fc0 T vfs_llseek
ffffffff968ca000 T vfs_readf
ffffffff968ca040 T vfs_writef
ffffffff968ca5e0 T vfs_dedupe_file_range_one
ffffffff968ca730 T vfs_dedupe_file_range
ffffffff968cb1e0 T vfs_clone_file_range
ffffffff968cbb30 T vfs_write
ffffffff968cbec0 T vfs_iter_read
ffffffff968cc4c0 T vfs_iter_write
ffffffff968cd6d0 T vfs_copy_file_range
ffffffff968cde40 T vfs_read
```

```
ffffffff968ce630 T vfs_readv
ffffffff968cf550 T vfs_get_tree
ffffffff968d1790 T vfs_get_super
ffffffff968d2960 T vfs_getattr_nosec
ffffffff968d2a10 T vfs_getattr
ffffffff968d2a60 T vfs_statx_fd
ffffffff968d2af0 T vfs_statx
ffffffff968d8ca0 T vfs_get_link
ffffffff968dbff0 T vfs_rmdir
ffffffff968dc1a0 T vfs_unlink
ffffffff968dc3e0 T vfs_tmpfile
ffffffff968dc4f0 T vfs_rename
ffffffff968dcf80 T vfs_whiteout
ffffffff968dd080 T vfs_mknod
ffffffff968dd2b0 T vfs_create
ffffffff968dd440 T vfs_symlink
ffffffff968dd5d0 T vfs_mkobj
ffffffff968dd770 T vfs_mkdir
ffffffff968dd930 T vfs_link
ffffffff968e0050 T vfs_path_lookup
ffffffff968e1580 T vfs_readlink
ffffffff968e2d10 T vfs_ioctl
ffffffff968ec3c0 T vfs_ioc_fssetxattr_check
ffffffff968ec960 T vfs_ioc_setflags_prepare
ffffffff968f25f0 T vfs_create_mount
ffffffff968f27d0 T vfs_kern_mount
ffffffff968f27f0 T vfs_submount
ffffffff968f9950 T vfs_getxattr_alloc
ffffffff968f9a70 T vfs_getxattr
ffffffff968f9bb0 T vfs_listxattr
ffffffff968fa810 T vfs_removexattr
ffffffff968faeb0 T vfs_setxattr
ffffffff96908df0 T vfs_fsync_range
ffffffff96908e70 T vfs_fsync
ffffffff9690b800 T vfs_get_fsid
ffffffff9690bac0 T vfs_statfs
ffffffff9690d260 T vfs_dup_fs_context
ffffffff9690d3f0 T vfs_parse_fs_param
ffffffff9690d5b0 T vfs_parse_fs_string
ffffffff9690d9c0 T vfs_clean_context
ffffffff969411f0 T vfs_cancel_lock
ffffffff96943640 T vfs_test_lock
ffffffff96943d40 T vfs_setlease
ffffffff96944f40 T vfs_lock_file
ffffffff982eb671 T vfs_caches_init_early
ffffffff982eb6ef T vfs_caches_init
```

В следующем примере мы используем такой набор API ядра:

mod_vfs.c

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

static char* file = NULL;
module_param(file, charp, 0);

#define BUF_LEN 120
#define DEFNAME "/etc/apt/sources.list";
static char buff[BUF_LEN + 1] = DEFNAME;

static int __init kread_init( void ) {
    struct file *f;
    int errno = -EPERM;
    size_t len;
    mm_segment_t old_fs;
    loff_t file_offset = 0;

    if(file != NULL) strcpy(buff, file);
    printk("+++ openning file: %s\n", buff);
    f = filp_open(buff, O_RDONLY, 0);
    if(IS_ERR(f)) {
        printk("+++ file open failed: %s\n", buff);
        return -ENOENT;
    }

    len = vfs_llseek(f, 0L, SEEK_END); // длина файла
    if(len <= 0) {
        printk("+++ failed to lseek %s\n", buff);
        return -EINVAL;
    }
    printk("+++ file size = %d bytes\n", (int)len);
    vfs_llseek(f, 0L, SEEK_SET);       // установка в начало

    old_fs = get_fs();                 // буфер в kernel space
    set_fs(KERNEL_DS);
    while(1) {
        #define SIZE 120
        char data[SIZE + 1];
        size_t n;
        // extern ssize_t vfs_read(struct file *, char __user *, size_t, loff_t *);
        n = vfs_read(f, data, SIZE, &file_offset);
```

```
        if(n < 0) {
            printk("+++ failed to read: %ld\n", n);
            errno = -EIO;
            break;
        }
        else if(n == 0) {
            printk("+++ read EOF\n");
            break;
        }
        else {
            data[n] = '\0';
            printk("+++ read %d bytes:\n%s\n", (int)n, data);
            errno = -EPERM;
        }
    }
    set_fs(old_fs);

    printk( KERN_ALERT "+++ close file\n" );
    filp_close(f, NULL);
    return (int)errno;
}

module_init(kread_init);
MODULE_LICENSE("GPL");
```

Этот вариант сложнее в кодировании, но он более гибкий в реализации. Гибкость проявляется в возможности использования *всего* набора функций файловых операций (таких как показанная в примере `vfs_llseek()`), привычных из POSIX-программирования, а не только узкого подмножества вызовов. А сложность состоит в том, что операции виртуальной файловой системы «заточены» на работу с *буферами данных* в пространстве пользователя и *проверяют* принадлежность адресов-параметров на принадлежность этому пространству. Для выполнения тех же операций с данными пространства ядра нужно снять эту проверку на время операции и восстановить ее после (собственно, регистр `FS` переустановить из пользовательского пространства на пространство ядра). Что и достигается использованием макровызовов: `get_fs()` и `set_fs(KERNEL_DS)`.

> ***Примечание***
>
> Обратите внимание, что в предыдущих версиях ядра это делалось совсем по-другому. Как — показано в архиве примеров для версии 3.18 (см. там папку file/_old_vers/3.18/). За этим надо тщательно следить — если этого не сделать или сделать неправильно, файловая операция завершится с кодом: `Bad address`.

И убеждаемся в работоспособности модуля над тем же тестовым файлом данных:

```
$ sudo insmod mod_vfs.ko file=xxx
[sudo] пароль для olej:
insmod: ERROR: could not insert module mod_vfs.ko: Operation not permitted
```

```
$ dmesg | tail -n9
[30568.982201] +++ openning file: xxx
[30568.982207] +++ file size = 42 bytes
[30568.982225] +++ read 42 bytes:
               *1 .........
               *2 .........
               *3 .........

[30568.982226] +++ read EOF
[30568.982227] +++ close file
$ ls -l xxx
-rw-rw-r-- 1 olej olej 42 авг 18 18:30 xxx
```

Запуск новых процессов из ядра

Можно ли запустить новый пользовательский процесс из кода модуля? Интуитивный ответ: наверняка можно, поскольку все и каждый процесс, выполняющиеся в системе, запущены именно из кода ядра. Остается вопрос: как?

Новые процессы пользовательского пространства могут запускаться кодом ядра *по форме* аналогично тому, как запускаются они и в пользовательском коде вызовами группы `exec*()`. Но *по содержанию* это действие имеет несколько другой смысл. Процессы из *пользовательского* кода создаются в два шага: первоначально выполняется операция `fork()`, которой создается *новое адресное пространство*, являющееся абсолютным дубликатом порождающего процесса. Это адресное пространство позже и становится пространством нового процесса, когда в нем производится вызов одной из функций семейства `exec()`. В пространстве ядра это должно происходить иначе — в нем нельзя создать дубликат ядерного пространства. Для выполнения запуска нового процесса здесь предоставляется специальный вызов `call_usermodehelper()`.

Простейший пример демонстрирует возможность порождения новых процессов в системе по инициативе ядра (см. папку exec в сопровождающем книгу файловом архиве):

mod_exec.c

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

static char *str = NULL;
module_param(str, charp, S_IRUGO);

int __init exec_init(void) {
   int rc;
   char *argv[] = { "wall", "\nthis is wall message ", NULL },
        *envp[] = { NULL },
        msg[ 80 ];
```

```
   if(str) {
      sprintf(msg, "\n%s", str);
      argv[1] = msg;
   }
   rc = call_usermodehelper("/usr/bin/wall", argv, envp, 0);
   if(rc) {
      printk(KERN_INFO "failed to execute : %s\n", argv[0]);
      return rc;
   }
   printk(KERN_INFO "execute : %s %s\n", argv[0], argv[1]);
   msleep(100);
   return -1;
}

module_init(exec_init);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");
MODULE_VERSION("3.3");
```

Вызов `call_usermodehelper()` получает параметры точно так же, как *системный* вызов пользовательского пространства `execve()` (через который выполняются все прочие *библиотечные* вызовы семейства `exec*()`). Необходимые детали вы найдете в справочной странице:

```
$ man 2 execve
```

Вот как срабатывает созданный модуль при нормальном течении исполнения:

```
$ sudo insmod mod_exec.ko
insmod: ERROR: could not insert module mod_exec.ko: Operation not permitted
$ dmesg | tail -n2
[39162.307779] execute : wall
               this is wall message
$ journalctl -n30 | grep wall
авг 18 21:08:09 R420 kernel: execute : wall
                                this is wall message
```

Модуль успешно загружается (виден нормальный запуск автономного пользовательского приложения), выводя *широковещательное* сообщение на те терминалы системы, на которые это предварительно было разрешено командой `msg`. Если приложение не может быть запущено — чаще всего из-за неправильно указанного *полного* путевого имени файла запускаемой программы, — код завершения загрузки модуля будет другим. Это существенно важно, учитывая, что модули исполняются в практически «глухонемом» режиме:

```
$ sudo insmod mod_exec.ko
insmod: error inserting 'mod_exec.ko': -1 Unknown symbol in module
$ dmesg | tail -n1
failed to execute : /bin/wall
```

Особенность и ограниченность метода запуска приложения вызовом call_usermodehelper() из ядра состоит в том, что процесс запускается **без** *управляющего терминала* и с нестандартным для него окружением! Это легко увидеть, если в качестве пользовательского процесса использовать утилиту echo, а строки запуска изменить (в папке exec сопровождающего книгу файлового архива для сравнения включен такой модуль: mod_execho.c):

```
char *argv[] = { "/bin/echo", "this is wall message", NULL };
...
rc = call_usermodehelper("/bin/echo", argv, envp, 0);
```

Результатом тут будет:

```
$ sudo insmod mod_execho.ko
insmod: error inserting 'mod_execho.ko': -1 Operation not permitted
$ dmesg | tail -n30 | grep -v ^audit
execute : /bin/echo echo message
```

Здесь имеет место *нормальное* выполнение утилиты echo, но мы не увидим результата ее работы (вывода на терминал), поскольку у нее просто нет этого управляющего терминала.

Всю основную работу в модуле по созданию и запуску процесса, как легко видеть, выполняет вызов call_usermodehelper() (<linux/kmod.h>), детали которого понятны из его прототипа:

```
static inline int call_usermodehelper(char *path, char **argv, char **envp, enum umh_wait wait);
enum umh_wait {
   UMH_NO_WAIT = -1,   /* don't wait at all */
   UMH_WAIT_EXEC = 0,  /* wait for the exec, but not the process */
   UMH_WAIT_PROC = 1,  /* wait for the process to complete */
};
```

Сигналы UNIX

Сигналы являются одной из немногих концептуальных основ операционных систем семейства UNIX, придающих им характерный вид. *Сигналы UNIX* (именно такое полное название применяется для них в литературе) часто могут быть не видны «на поверхности» пользователю, но играют значительную роль внутри системы, — достаточно обратить внимание на то, что без помощи сигналов мы не смогли бы завершить ни один процесс в системе (нажимая <^C> или выполняя команды kill или killall). Детальное углубление в теорию сигналов увело бы нас слишком далеко от наших намерений, так что в сферу дальнейшего рассмотрения будет попадать только статус сигналов UNIX в ядре.

Да, речь идет именно о возможности отправки сигналов процессу пространства пользователя или потоку пространства ядра. То, что мы привычно делаем в пользовательском пространстве командой kill. Точно так же, как и запуск нового процесса по аналогии с пользовательским пространством, можно посылать из ядра сигналы UNIX как пользовательским процессам, так и потокам пространства ядра. Воз-

можность таких взаимодействий должна быть понятна из простого соображения, заключающегося в том, что большинство вызовов API ядра, которые рассматривались на протяжении всего протяженного предыдущего изложения, и таких, которые переводят текущее выполнение в блокированное состояние, имеют две альтернативные формы: форму безусловную, ожидающую наступления финального условия, и форму, допускающую прерывание ожидания получением *сигнала*. Остается вопрос: *как* реализовать такую возможность?

Для уяснения возможностей использования сигналов из ядра (и в ядре) я воспользовался несколько видоизмененным (см. папку signal в сопровождающем книгу файловом архиве) проектом из [6]. Идея этого в меру громоздкого, 3-компонентного теста проста:

- пользовательское приложение `sigreq` («мишень», на которую направляются сигналы), регистрирующее получаемые сигналы;
- модуль ядра ioctl_signal.ko, которому можно «заказать», какому пользовательскому процессу (любому по PID) отсылать сигнал (этот процесс и станет служить «мишенью» — в качестве целеуказания мы будем указывать `sigreq`);
- диалоговый пользовательский процесс `ioctl`, который указывает модулю ядра ioctl_signal.ko, какой именно сигнал отсылать (по номеру) и какому процессу. Процесс `ioctl` будет передавать команды для ioctl_signal.ko посредством вызовов `ioctl()`, что мы уже видели при рассмотрении драйверов символьных устройств.

Общие определения, необходимые для команд `ioctl()`, вынесены в отдельный файл ioctl.h:

ioctl.h

```
#define MYIOC_TYPE 'k'
#define MYIOC_SETPID   _IO(MYIOC_TYPE,1)
#define MYIOC_SETSIG   _IO(MYIOC_TYPE,2)
#define MYIOC_SENDSIG  _IO(MYIOC_TYPE,3)
#define SIGDEFAULT SIGKILL
```

Команды `ioctl()`, которые отрабатываются модулем:

- `MYIOC_SETPID` — установить PID процесса, которому будет направляться сигнал;
- `MYIOC_SETSIG` — установить номер отсылаемого сигнала;
- `MYIOC_SENDSIG` — отправить сигнал.

Собственно код модуля:

ioctl_signal.c

```
#include <linux/module.h>
#include "ioctl.h"
#include "lab_miscdev.h"
```

```
static int sig_pid = 0;
static struct task_struct *sig_tsk = NULL;
static int sig_tosend = SIGDEFAULT;

static inline long mycdrv_unlocked_ioctl(struct file *fp, unsigned int cmd, unsigned long arg)
{
   int retval;
   switch(cmd) {
      case MYIOC_SETPID:
         sig_pid = (int)arg;
         printk(KERN_INFO "Setting pid to send signals to, sigpid = %d\n", sig_pid);
         /* sig_tsk = find_task_by_vpid (sig_pid); */
         sig_tsk = pid_task(find_vpid(sig_pid), PIDTYPE_PID);
         break;
      case MYIOC_SETSIG:
         sig_tosend = (int)arg;
         printk(KERN_INFO "Setting signal to send as: %d \n", sig_tosend);
         break;
      case MYIOC_SENDSIG:
         if(!sig_tsk) {
            printk(KERN_INFO "You haven't set the pid; using current\n");
            sig_tsk = current;
            sig_pid = (int)current->pid;
         }
         printk(KERN_INFO "Sending signal %d to process ID %d\n", sig_tosend, sig_pid);
         retval = send_sig(sig_tosend, sig_tsk, 0);
         printk(KERN_INFO "retval = %d\n", retval);
         break;
      default:
         printk(KERN_INFO " got invalid case, CMD=%d\n", cmd);
         return -EINVAL;
   }
   return 0;
}

static const struct file_operations mycdrv_fops = {
   .owner = THIS_MODULE,
   .unlocked_ioctl = mycdrv_unlocked_ioctl,
   .open = mycdrv_generic_open,
   .release = mycdrv_generic_release
};

module_init(my_generic_init);
module_exit(my_generic_exit);
MODULE_AUTHOR("Jerry Cooperstein");
MODULE_LICENSE("GPL v2");
```

В этом файле содержится интересующий нас обработчик функций `ioctl()`, а 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 показано ранее, при рассмотрении операций символьного устройства.

Пока остановим внимание на группе функций, находящих процесс по его PID, что близко смыкается с задачей запуска процесса, решавшейся ранее, — для этого используется инструментарий:

```
#include <linux/sched.h>
struct task_struct *find_task_by_vpid(pid_t nr);
#include <linux/pid.h>
struct pid *find_vpid(int nr);
struct task_struct *pid_task(struct pid *pid, enum pid_type);
enum pid_type {
   PIDTYPE_PID,
   PIDTYPE_PGID,
   PIDTYPE_SID,
   PIDTYPE_MAX
};
```

Вот тестовая задача, выполняющая в диалоге с пользователем последовательность команд `ioctl()` над модулем — одну за другой: установка PID процесса, установка номера сигнала, отправка 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 приемником-регистратором отправляемых сигналов («мишень»):

sigreq.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include "ioctl.h"

static void sig_handler(int signo) {
   printf("---> signal %d\n", signo);
}

int main(int argc, char *argv[]) {
   unsigned long sig = SIGDEFAULT;
   printf("my own PID is %d\n", getpid());..
   sig = SIGDEFAULT;
   if(argc > 1) sig = atoi(argv[ 1 ]);
   if(SIG_ERR == signal(sig, sig_handler))
      printf("set signal handler error\n");
   while(1) pause();
   exit(0);
}
```

В этом приложении (как и в предыдущем) для установки обработчика сигнала используется старая, так называемая *ненадежная модель* обработки сигналов, с ис-

пользованием вызова `signal()`, но в нашем случае это никак не влияет на достоверность получаемых результатов.

Начнем проверку с конца — просто с отправки процессу-регистратору сигнала консольной командой `kill`, но прежде нужно разобраться 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 проверки функционирования может быть использован любой (почти) из этого набора сигналов UNIX — мы выберем безобидный (не имеющий специального предназначения) сигнал `SIGUSR1` (сигнал номер 10). Вот как отреагировал процесс-регистратор («мишень») на получение этого сигнала:

```
$ ./sigreq 10
my own PID is 56459
---> signal 10
---> signal 10
^C
$ kill -n 10 56459
$ kill -n SIGUSR1 56459
```

А теперь выполним весь комплекс: процесс `ioctl` последовательностью вызовов `ioctl()` заставляет загруженный модуль ядра отправить указанный сигнал процессу `sigreq`:

```
$ sudo insmod ioctl_signal.ko
$ lsmod | head -n2
Module                  Size  Used by
ioctl_signal           16384  0
$ dmesg | tail -n2
[21081.470356] Succeeded in registering character device mycdrv
$ ls -l /dev | grep my
crw-------   1 root root       10,  55 июн 28 15:08 mycdrv
$ sudo ./ioctl 10 56795
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
$ ./sigreq 10
my own PID is 56795
---> signal 10
...
$ sudo rmmod ioctl_signal
$ dmesg | tail -n14
[21081.470356]  Succeeded in registering character device mycdrv
[21271.047313]  attempting to open device: mycdrv:
[21271.047315]  MAJOR number = 10, MINOR number = 55
[21271.047315]  successfully open  device: mycdrv:
[21271.047316] I have been opened  1 times since being loaded
[21271.047316] ref=1
[21271.047389] Setting pid to send signals to, sigpid = 56795
[21271.047394] Setting signal to send as: 10
[21271.047398] Sending signal 10 to process ID 56795
[21271.047414] retval = 0
[21271.047420]  closing character device: mycdrv:
```

Отправку сигнала в этой реализации осуществляет вызов `send_sig()` — он и еще большая группа функций, связанных с отправкой сигналов, определены в <linux/sched.h>. Вот некоторые из них:

```
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);
```

Описание 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 — это и еще одна из самых слабо освещенных в литературе.

Вокруг экспорта символов ядра...

Теперь углубимся детально в самое путанное понятие из области ядра — *экспорт символов* ядра. Чтобы имя пространства ядра было доступно для связывания в вашем модуле, для этого имени должны выполняться два условия: а) имя должно иметь глобальную область видимости (в вашем модуле такие имена не должны объявляться `static`) и б) имя должно быть явно объявлено *экспортируемым* — т. е. явно записано параметром макроса `EXPORT_SYMBOL` или `EXPORT_SYMBOL_GPL`. Мы уже встречались с понятием экспортирования в начале нашего экскурса в технику модулей ядра, но только сейчас сможем разобраться с ним подробно (наработав определенный багаж для создания тестовых модулей).

> ***Примечание***
>
> Реализация здесь будет показана для 32-битного дистрибутива и ядра 4.9, в 64-битной системе имена системных вызовов будут немного отличаться по виду, но все рассуждения остаются в силе.

```
$ inxi -S
System:
  Host: antix21 Kernel: 4.9.0-279-antix.1-486-smp arch: i686 bits: 32
    Desktop: IceWM v: 2.9.7
    Distro: antiX-21_386-base Grup Yorum 31 October 2021
```

Выберем для сравнительного изучения два сходных системных вызова: `sys_open` и `sys_close`:

```
$ sudo cat /proc/kallsyms | grep ' T ' | grep sys_close
c91a9c70 T sys_close
$ sudo cat /proc/kallsyms | grep ' T ' | grep ' sys_open'$
c91ab810 T sys_open
```

Оба имени: `sys_open` и `sys_close` известны в таблице символов ядра как глобальные имена (`T`) в секции кода. Сделаем (см. папку sys_call_table/export в сопровождающем книгу файловом архиве) простейший модуль ядра:

md_32c.c

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

extern int sys_close(int fd);

static int __init sys_init(void) {
   void* Addr;
   Addr = (void*)sys_close;
   printk(KERN_INFO "sys_close address: %p\n", Addr);
   return -1;
}
module_init(sys_init);
MODULE_LICENSE("GPL v2");
```

```
$ sudo insmod md_32c.ko
insmod: ERROR: could not insert module md_32c.ko: Operation not permitted
olej@antix21:~/2022/kernel/sys_call_table/export
$ dmesg | tail -n1
[124910.472207] sys_close address: c91a9c70
```

Всё идет так, как и можно было предполагать: адрес обработчика системного вызова `sys_close` (экспортированный ядром и полученный в ходе выполнения) совпадает со значением, полученным ранее из /proc/kallsyms. Теперь посмотрим на точно такой же модуль, но относительно обработчика для симметричного системного вызова `sys_open`:

md_32o.c

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

extern int sys_open(int fd);

static int __init sys_init(void) {
   void* Addr;
   Addr = (void*)sys_open;
   printk(KERN_INFO "sys_open address: %p\n", Addr);
   return -1;
}
module_init(sys_init);
MODULE_LICENSE("GPL v2");
```

Здесь описанный прототип `sys_open()` не соответствует реальному формату вызова обработчика системного вызова для `open()`, но это не имеет значения, т. к. мы не собираемся производить вызов, а только получаем адрес для связывания... Но адрес-то как раз и не получается:

```
$ make
make -C /lib/modules/4.9.0-279-antix.1-486-smp/build
M=/home/olej/2022/kernel/sys_call_table/export modules
make[1]: вход в каталог «/usr/src/linux-headers-4.9.0-279-antix.1-486-smp»
  CC [M]  /home/olej/2022/kernel/sys_call_table/export/md_32o.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: "sys_open" [/home/olej/2022/kernel/sys_call_table/export/md_32o.ko] undefined!
  CC      /home/olej/2022/kernel/sys_call_table/export/md_32o.mod.o
  LD [M]  /home/olej/2022/kernel/sys_call_table/export/md_32o.ko
make[1]: выход из каталога «/usr/src/linux-headers-4.9.0-279-antix.1-486-smp»
```

Уже на этапе сборки мы получаем предупреждение (`WARNING`), но, не обратив на него внимания, пытаемся загрузить модуль (модуль-то создался!):

```
$ ls *.ko
md_32c.ko  md_32o.ko
$ sudo insmod md_32o.ko
insmod: ERROR: could not insert module md_32o.ko: Unknown symbol in module
$ dmesg | tail -n1
[125175.046023] md_32o: Unknown symbol sys_open (err 0)
```

Это определенно не то, что мы ожидали! Такой модуль не может быть загружен, потому как он противоречит правилам целостности ядра: содержит неразрешенный внешний символ — этот символ *не экспортируется ядром для связывания*.

Ссылаться по именам на объекты в коде своего модуля мы можем только на те имена, которые *экспортированы* (либо ядром, либо любым *ранее* загруженным модулем ядра). Где мы можем уточнить, какие из символов ядра являются экспортируемыми, а какие — нет? Когда речь идет о нескольких десятках тысяч символов ядра (прежде всего нас должны интересовать имена для внешнего связывания `' T '`):

```
$ cat /proc/kallsyms | wc -l
48161
$ cat /proc/kallsyms | grep ' T ' | wc -l
18865
```

Информацию по экспортируемым символам ищем здесь:

```
$ grep 'vmlinux' /lib/modules/`uname -r`/build/Module.symvers | head
0x6fbf0e92      ipv6_chk_custom_prefix          vmlinux         EXPORT_SYMBOL
0xc560732c      cros_ec_check_result            vmlinux         EXPORT_SYMBOL
0xef924f99      sata_pmp_error_handler          vmlinux         EXPORT_SYMBOL_GPL
0x55417264      unregister_vt_notifier          vmlinux         EXPORT_SYMBOL_GPL
0xcfc9cf89      debugfs_use_file_finish         vmlinux         EXPORT_SYMBOL_GPL
0x309b6730      set_anon_super                  vmlinux         EXPORT_SYMBOL
0x464ee68d      __cleancache_invalidate_page    vmlinux         EXPORT_SYMBOL
0xd3d34f27      kmem_cache_alloc                vmlinux         EXPORT_SYMBOL
0xce0162d0      replace_page_cache_page         vmlinux         EXPORT_SYMBOL_GPL
0x9432b036      net_cls_cgrp_subsys_enabled_key vmlinux         EXPORT_SYMBOL_GPL
```

Вот такой примерно формат имеет каждая строка файла Module.symvers, соответствующая описанию одного экспортируемого символа:

- имя символа (вторая колонка);
- модуль, который экспортирует символ, с указанием *пути* к файлу модуля, или `vmlinux`, если символ экспортируется непосредственно ядром;
- тип экспортирования: `EXPORT_SYMBOL` или `EXPORT_SYMBOL_GPL`.

Итого:

```
$ cat /lib/modules/`uname -r`/build/Module.symvers | wc -l
20380
$ grep 'vmlinux' /lib/modules/`uname -r`/build/Module.symvers | wc -l
8998
$ grep 'vmlinux' /lib/modules/`uname -r`/build/Module.symvers | grep EXPORT_SYMBOL_GPL | wc -l
4222
```

Здесь мы видим, что число экспортируемых *ядром* символов (8998) вдвое меньше общего числа имен (функций для внешнего связывания) ядра (20380), и, более того, около половины экспортируемых имен экспортируется под лицензией GPL (т. е. даже экспортируемые имена далеко не во всех проектах могут использоваться). Среди них, в частности, есть sys_close, но нет sys_open:

```
$ cat /lib/modules/`uname -r`/build/Module.symvers | grep sys_close
0x268cc6a2      sys_close       vmlinux EXPORT_SYMBOL
$ cat /lib/modules/`uname -r`/build/Module.symvers | grep sys_open
$
```

Если же сборка модуля производится в отдельном каталоге (на период отработки) и интерес представляет контроль символов, экспортируемых именно этим модулем, то информацию о них ищем в локальном файле Module.symvers в рабочем каталоге сборки.

Всё то, что показано здесь, радикально зависит от: а) архитектуры процессора; б) версии и разрядности ядра и, самое главное, в) от того, как определялись при сборке ядра конфигурационные параметры вот этой группы:

```
$ cat /boot/config-`uname -r` | grep CONFIG_KALLSYMS
CONFIG_KALLSYMS=y
CONFIG_KALLSYMS_ALL=y
CONFIG_KALLSYMS_ABSOLUTE_PERCPU=y
CONFIG_KALLSYMS_BASE_RELATIVE=y
```

В зависимости от этих факторов в вашей инсталляции картина внешне может существенно отличаться, но здесь показаны неизменные принципы, которые позволяют найти решения в каждой инсталляции.

Неэкспортируемые символы ядра

Означает ли показанное ранее, что *только* экспортируемые символы ядра доступны в коде нашего модуля? Нет, это означает только, что *рекомендуемый* способ связывания **по** *имени* применим только к экспортируемым именам. Экспортирование обеспечивает еще один дополнительный рубеж контроля целостности ядра (целостность *монолитного* ядра, как видим, это совсем не игрушки) — минимальная некорректность приводит к полному краху операционной системы, иногда при этом она даже не успевает выкрикнуть: Oops... Особенно это относится к модулям, подобным тем, к рассмотрению которых мы приступаем (см. папку sys_call_table/call_table/ в сопровождающем книгу файловом архиве).

Как оказалось, и все другие имена (функций, переменных, структур) из /proc/kallsyms доступны для использования нашему модулю, если модуль возьмет их оттуда сам. В простейшем для понимания изложении это могло бы выглядеть так, что модуль должен вычитать /proc/kallsyms (чтение мы уже рассматривали раньше), найти там адрес интересующего его имени (а интересовать меня будет sys_call_table как самое фундаментальное понятие ядра) и далее им воспользоваться... Это *очень грубая* схема и, как будет показано далее, это даже не схема, а модель, объясняющая

принцип. Но она полностью реализована и работоспособна в примере модуля mod_rct.c (здесь я приведу только центральную часть кода, перебирающую символы ядра, а полный работающий и выверенный код — в архиве):

md_rct.c

```
...
static char* file = "/proc/kallsyms";
...
#define BUF_LEN 255
static  char buff[BUF_LEN + 1];
...
    if(!IS_ENABLED(CONFIG_KALLSYMS)) {
        printk("+ no config: CONFIG_KALLSYMS\n");
        return -ENOENT;
    }
    f = filp_open(file, O_RDONLY, 0);
    if(IS_ERR(f)) {
        printk("+ failed to open: %s\n", file);
        err = -ENOENT;
        goto close;
    }
    printk("+ openning file: %s\n", file);
    if(!(f->f_mode & FMODE_READ)) {
        err = -EBADF;
        goto close;
    }
    if(!f->f_op) {
        if(!f->f_op->read)
            printk("+ read op. => %px\n", f->f_op->read);
    }
    else
        printk("+ op. table => %px\n", (void*)0);

    while(1) {
        size_t k;
        char *p = buff, *find = NULL;
        int n;
        for(n = 0; n < BUF_LEN; n++) {
#if (LINUX_VERSION_CODE < KERNEL_VERSION(4, 14, 0))
//   extern int kernel_read(struct file *, loff_t, char *, unsigned long);
            k = kernel_read(f, pos, p++, 1);
            pos += k;
#else
//   extern ssize_t kernel_read(struct file *, void *, size_t, loff_t *);
            k = kernel_read(f, p++, 1, &pos);
```

```
#endif
            if(k < 0) {
                printk("+ failed to read\n");
                err = -EIO;
                goto close;
            }
            if(0 == k) break;
            *p = '\0';
            if('\n' == *(p - 1)) break; // считана строка
        }
        if((debug != 0) && (strlen(buff) > 0)) {
            if('\n' == buff[strlen(buff) - 1]) printk("+ %s", buff);
            else printk("+ %s|\n", buff);
        }
        if(0 == k) break;                  // EOF
        if(NULL == (find = strstr(buff, " sys_call_table\n"))) continue;
        put_table(buff);
        break;
    }

close:
    printk("+ close file: %s\n", file);
    filp_close(f, NULL);
    return err;
}
```

Да, это сложно, громоздко и натужно, но сам принцип такое решение хорошо разъясняет. Вот как выглядит исполнение этого модуля (я запускаю его с `time`, и видно, что даже на очень быстром процессоре перебор многих тысяч имен занимает немало времени — свыше полсекунды):

```
$ inxi -C
CPU:       Topology: 2x 10-Core model: Intel Xeon E5-2470 v2 bits: 64
           type: MT MCP SMP L2 cache: 50.0 MiB
           Speed: 1200 MHz min/max: 1200/3200 MHz Core speeds (MHz): 1: 1295 2: 1345 3: 1284
           4: 1456 5: 1285 6: 1913 7: 1444 8: 2040 9: 1296 10: 1888 11: 1400 12: 1336 13: 1234
           14: 1270 15: 1263 16: 1286 17: 1307 18: 1292 19: 1938 20: 1233 21: 2023 22: 1234
           23: 1309 24: 1707 25: 1756 26: 1951 27: 1357 28: 1388 29: 1310 30: 2758 31: 1510
           32: 2326 33: 1375 34: 2468 35: 1395 36: 1792 37: 2389 38: 1611 39: 1526 40: 1646
$ inxi -S
System:    Host: R420 Kernel: 5.4.0-124-generic x86_64 bits: 64 Desktop: Cinnamon 5.2.7
Distro: Linux Mint 20.3 Una
$ uname -r
5.4.0-124-generic
$ time sudo insmod mod_rct.ko
insmod: ERROR: could not insert module mod_rct.ko: Operation not permitted
real    0m0,538s
```

```
user    0m0,004s
sys     0m0,503s
$ dmesg | tail -n7
[13604.789747] + openning file: /proc/kallsyms
[13604.789749] + op. table => 0000000000000000
[13605.279758] + ffffffff996013c0 R sys_call_table
[13605.279759] + ffffffff996013c0
[13605.279760] + sys_call_table address = ffffffff996013c0
[13605.279765] + sys_call_table : ffffffff988ce0d0 ffffffff988ce1f0 ffffffff988c9da0
ffffffff988c7e50 ffffffff988d31b0 ffffffff988d33d0 ffffffff988d3270 ffffffff988e70a0
ffffffff988cb9f0 ffffffff98636820  ...
[13605.279766] + close file: /proc/kallsyms
```

Помимо адреса таблицы `sys_call_table` (таблицы системных вызовов ядра) модуль выводит (для контроля) 10 первых точек входа этой таблицы. Проверим, что представляют собой эти адреса обратным поиском:

```
$ sudo grep ffffffff988ce0d0 /proc/kallsyms
ffffffff988ce0d0 T __x64_sys_read
$ sudo grep ffffffff988ce1f0 /proc/kallsyms
ffffffff988ce1f0 T __x64_sys_write
$ sudo grep ffffffff988c9da0 /proc/kallsyms
ffffffff988c9da0 T __x64_sys_open
$ sudo grep ffffffff988c7e50 /proc/kallsyms
ffffffff988c7e50 T __x64_sys_close
$ sudo grep ffffffff988d31b0 /proc/kallsyms
ffffffff988d31b0 T __x64_sys_newstat
$ sudo grep ffffffff988d33d0 /proc/kallsyms
ffffffff988d33d0 T __x64_sys_newfstat
$ sudo grep ffffffff988d3270 /proc/kallsyms
ffffffff988d3270 T __x64_sys_newlstat
$ sudo grep ffffffff988e70a0 /proc/kallsyms
ffffffff988e70a0 T __x64_sys_poll
$ sudo grep ffffffff988cb9f0 /proc/kallsyms
ffffffff988cb9f0 T __x64_sys_lseek
$ sudo grep ffffffff98636820 /proc/kallsyms
ffffffff98636820 T __x64_sys_mmap
```

Это в точности начало того массива номеров обработчиков системных вызовов Linux (в пространстве пользователя), индексы которого мы видели в начале одной из предыдущих глав (для 32-битной системы это будет unistd_32.h):

```
$ ls -l /usr/include/asm/unistd_*
-rw-r--r-- 1 root root 11475 авг  4 04:48 /usr/include/asm/unistd_32.h
-rw-r--r-- 1 root root  9286 авг  4 04:48 /usr/include/asm/unistd_64.h
-rw-r--r-- 1 root root 16367 авг  4 04:48 /usr/include/asm/unistd_x32.h
$ head -n13 /usr/include/asm/unistd_64.h
#ifndef _ASM_X86_UNISTD_64_H
#define _ASM_X86_UNISTD_64_H 1
```

```
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
```

Недостаток того, что показано здесь, — громоздкость и *искусственность*. Но /proc/kallsyms — это не что иное, как некоторые структуры ядра, и, к счастью, ядро предоставляет нам вызов `kallsyms_lookup_name()` (экспортирует его), позволяющий делать поиск в этой структуре. Вот, насколько все стало проще:

mod_kct.c

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

static int __init ksys_call_tbl_init(void) {
   void** psct = (void**)kallsyms_lookup_name("sys_call_table");
   printk("+ sys_call_table address = %px\n", psct);
   if(psct) {
      int i;
      char table[200] = "sys_call_table : ";
      for(i = 0; i < 10; i++)
         sprintf( table + strlen( table ), "%px ", psct[i]);
      printk( "+ %s ...\n", table );
   }
   return -EPERM;
}

module_init( ksys_call_tbl_init );

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");
MODULE_VERSION("3.4");
```

Всё! Вот как происходит исполнение этого варианта — это в точности тот же результат, но только в 7 раз быстрее по времени:

```
$ time sudo insmod mod_kct.ko
insmod: ERROR: could not insert module mod_kct.ko: Operation not permitted
real    0m0,077s
user    0m0,010s
sys     0m0,018s
```

```
$ dmesg | tail -n2
[17826.826422] + sys_call_table address = ffffffff996013c0
[17826.826426] + sys_call_table : ffffffff988ce0d0 ffffffff988ce1f0 ffffffff988c9da0
ffffffff988c7e50 ffffffff988d31b0 ffffffff988d33d0 ffffffff988d3270 ffffffff988e70a0
ffffffff988cb9f0 ffffffff98636820  ...
```

Но!.. В каждой бочке меда должна быть своя ложка дегтя — пытаемся сделать то же, но в намного более ранней версии ядра::

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
$ make
...
WARNING: "kallsyms_lookup_name" [/home/olej/2011_WORK/LINUX-books/examples.DRAFT/
sys_call_table/call_table/mod_kct.ko] undefined!
...
```

Конечно! И в ядре 2.6.32 (в этой конкретной сборке ядра!) такое имя тоже есть:

```
$ cat /proc/kallsyms | grep kallsyms_ | grep T
c046ca7c T module_kallsyms_on_each_symbol
c046e815 T module_kallsyms_lookup_name
c0471581 T kallsyms_lookup
c04716ec T kallsyms_lookup_size_offset
c0471764 T kallsyms_on_each_symbol
c04717f2 T kallsyms_lookup_name
```

Но оно *не экспортируется*:

```
$ cat /lib/modules/`uname -r`/build/Module.symvers | grep kallsyms_
0x00000000       kallsyms_on_each_symbol   vmlinux     EXPORT_SYMBOL_GPL
```

Однако это несчастье легко преодолеть! Для чего мы воспользуемся другим экспортируемым именем, которое только что видели в таблице символов: `kallsyms_on_each_symbol` — этот вызов обеспечивает выполнение заказанной пользовательской функции последовательно *для всех* имен ядра. Он сложнее, и здесь нужны краткие пояснения. Указанный вызов имеет прототип (<linux/kallsyms.h>):

```
int kallsyms_on_each_symbol(int (*fn)(void *, const char *, struct module *, unsigned long),
                            void *data);
```

Первым параметром (`fn`) — из *двух* имеющихся — он получает указатель на вашу пользовательскую функцию, которая и будет последовательно вызываться *для всех символов* в таблице ядра, а вторым (`data`) — указатель на *произвольный* блок данных (параметров), который будет передаваться *в каждый* вызов функции `fn`. Это достаточно обычная практика, подобные аналогии мы видим, например, при создании нового потока (как потока ядра, так и потока пользовательского пространства в POSIX API).

Прототип пользовательской функции `fn`, которая циклически вызывается для каждого имени:

```
int func(void *data, const char *symb, struct module *mod, unsigned long addr);
```

где:

- `data` — блок параметров, заполненный в вызывающей единице и переданный из вызова функции `kallsyms_on_each_symbol()` (второй параметр вызова), как это описано ранее. Здесь как раз и следует передать имя того символа, который мы разыскиваем;
- `symb` — символьное изображение (строка) имени из таблицы имен ядра, которое обрабатывается на текущем вызове `func`;
- `mod` — модуль ядра, к которому относится обрабатываемый символ;
- `addr` — адрес символа в адресном пространстве ядра (собственно, то, что мы и ищем).

Перебор имен таблицы ядра можно *прервать* на текущем шаге (из соображений эффективности — если мы уже обработали требуемые нам символы), когда пользовательская функция `func` возвратит ненулевое значение. Этого более чем достаточно для того, чтобы построить следующий, третий эквивалент нашим предыдущим модулям:

mod_koes.c

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

static int nsym = 0;
static unsigned long taddr = 0;

int symb_fn(void* data, const char* sym, struct module* mod, unsigned long addr) {
   nsym++;
   if(0 == strcmp((char*)data, sym)) {
      printk("+ sys_call_table address = %lx\n", addr);
      taddr = addr;
      return 1;
   }
   return 0;
}

static int __init ksys_call_tbl_init(void) {
   int n = kallsyms_on_each_symbol(symb_fn, (void*)"sys_call_table");
   if(n != 0) {
      int i;
      char table[200] = "sys_call_table : ";
      printk("+ find in position %d\n", nsym);
      for(i = 0; i < 10; i++) {
         unsigned long sa = *((unsigned long*)taddr + i);
```

```
            sprintf(table + strlen(table), "%px ", (void*)sa);
        }
        printk("+ %s ...\n", table);
    }
    else printk("+ symbol not found\n");
    return -EPERM;
}

module_init(ksys_call_tbl_init);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");
MODULE_VERSION("3.4");
```

Первоначально, для убедительности, возвратимся и выполним этот код в ядре 2.6.32, где у нас не работал предыдущий пример (т. е. вообще в любое ядро, где предыдущий метод не работает):

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
$ time sudo insmod mod_koes.ko
insmod: error inserting 'mod_koes.ko': -1 Operation not permitted
real    0m0.042s
user    0m0.005s
sys     0m0.027s
$ dmesg | tail -n30 | grep +
+ sys_call_table address = c07ab3d8
+ find in position 25239
+ sys_call_table : c044ec61 c0444f63 c040929c c04e149d c04e12fc c04dec35 c04dea99 c0444767
c04dec60
$ cat /proc/kallsyms | grep c04dec35
c04dec35 T sys_open
```

Это и есть [5] системный вызов в 32-битной системе:

```
$ grep NR_open /usr/include/asm/unistd_32.h
#define __NR_open 5
#define __NR_openat 295
#define __NR_open_by_handle_at 342
#define __NR_open_tree 428
```

После чего вернемся в более свежую сборку:

```
$ uname -r
5.4.0-124-generic
$ time sudo insmod mod_koes.ko
insmod: ERROR: could not insert module mod_koes.ko: Operation not permitted
real    0m0,076s
user    0m0,007s
sys     0m0,016s
```

```
$ dmesg | tail -n3
[19573.359810] + sys_call_table address = ffffffff996013c0
[19573.359811] + find in position 51962
[19573.359815] + sys_call_table : ffffffff988ce0d0 ffffffff988ce1f0 ffffffff988c9da0
ffffffff988c7e50 ffffffff988d31b0 ffffffff988d33d0 ffffffff988d3270 ffffffff988e70a0
ffffffff988cb9f0 ffffffff98636820  ...
$ cat /proc/kallsyms | wc -l
144140
```

Хорошо видно, что:

- это те же результаты, что и в предыдущих примерах;
- это практически те же затраты времени, что и в предыдущем случае — с `kallsyms_lookup_name`;
- для нахождения требуемого символа читалась не вся таблица имен ядра (144 140 символов), а только до нахождения интересующего символа: в нашем случае (51 962) — около одной ее трети.

> ***Обратите внимание!***
>
> Здесь я искал символ ядра `sys_call_table` (зачем и почему именно его, вскоре станет понятно). Но таким же способом можно найти и использовать *любой* символ (вызов API ядра), который *не является* экспортируемым!

Использование неэкспортируемых символов

Теперь мы умеем получать в своем коде модуля адреса *любых*, а не только экспортируемых ядром символов. Это прямой путь к их практическому использованию. Первейшим применением, и просто просящимся к реализации, была бы подмена функции обработчика системного вызова, о которой мы говорили ранее. Но именно потому, что это не тривиально, мы не станем пока этим заниматься[2]. А сделаем куда более красивый трюк (см. папку sys_call_table/new_sys в сопровождающем книгу файловом архиве) — в качестве иллюстрации возможности и реализации технических приемов мы выполним пользовательский библиотечный[3] вызов `printf()` из кода модуля ядра (а значит, и любого системного вызова!). Выглядит это так:

mod_wrc.c

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

static unsigned long waddr = 0;
static char buf[80];
static int len;
```

[2] Кроме того, вирусописателям тоже нужно оставить кое-что на самостоятельную проработку…

[3] В начале книги мы говорили, что из ядра нельзя вызывать библиотечные вызовы POSIX API. Дело в том, что из ядра можно всё! Но только... осторожно.

```
int symb_fn(void* data, const char* sym, struct module* mod, unsigned long addr) {
   if(0 == strcmp((char*)data, sym)) {
      waddr = addr;
      return 1;
   }
   else return 0;
}

/* <linux/syscalls.h>
asmlinkage long sys_write(unsigned int fd, const char __user *buf,
                          size_t count); */
static asmlinkage long (*sys_write) (
   unsigned int fd, const char __user *buf, size_t count);

static int do_write(void) {
   int n;
   mm_segment_t old_fs = get_fs();
   set_fs(KERNEL_DS);                  // буфер в kernel space
   sys_write = (void*)waddr;
   n = sys_write(1, buf, len);
   set_fs(old_fs);                     // восстановить
   return n;
}

static int __init wr_init(void) {
   int n = kallsyms_on_each_symbol(symb_fn, (void*)"sys_write");
   if(n != 0) {
      sprintf(buf, "адрес системного обработчика sys_write = %lx\n", waddr);
      len  = strlen(buf) + 1;
      printk("+ [%d]: %s", len, buf);
      n = do_write();
      printk("+ write return : %d\n", n);
   }
   else
      printk("+ %d: symbol not found\n", n);
   return -EPERM;
}

module_init(wr_init);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");
```

Собственно, выполняем мы в этом примере не `printf()`, но, вспомнив, что *библиотечный* вызов `printf()`, как обсуждалось ранее, является не чем иным, как последовательностью *библиотечного* вызова `sprintf()` (форматирование вывода) с последующим *системным* вызовом `write(1, ...)`, мы фактически решаем поставленную задачу, т. к. имитируем в точности ту же последовательность вызовов:

```
$ uname -r
4.9.0-279-antix.1-486-smp
$ sudo insmod mod_wrc.ko
адрес системного обработчика sys_write = c91ae1e0
insmod: ERROR: could not insert module mod_wrc.ko: Operation not permitted
$ dmesg | tail -n2
[147438.442785] + [77]: адрес системного обработчика sys_write = c91ae1e0
[147438.442947] + write return : 77
```

Вывод строки происходит *до* вывода сообщения о завершении выполнения кода модуля и на *графический терминал* (X11) — мы видели уже такое мельком при рассмотрении системных вызовов. Естественно, вызов `write()` произведен из функции инициализации модуля и поэтому осуществляет вывод на управляющий терминал *вызывающего* процесса — в нашем случае такой процесс: `insmod`. Для проверки (адреса обработчика `sys_write` — хотя что тут проверять?) проделаем:

```
$ cat /proc/kallsyms | grep 'T sys_write'
c91ae1e0 T sys_write
c91ae4f0 T sys_writev
```

В коде модуля нет ничего сложного... за исключением, возможно, мелочей: откуда я взял прототип *для своей* новой функции `sys_write()`, которой позже присвою адрес системного обработчика вызова `sys_write`? Вот определение, о котором идет речь:

```
asmlinkage long sys_write(unsigned int fd, const char __user *buf,
                          size_t count);
```

Конечно же, бесстыдно списал из заголовочного файла <linux/syscalls.h>. О чем даже вписал комментарий в код модуля — чтобы это не забыть. И советую и вам в точности так же поступать в отношении и *всех других* системных вызовов. Потому что в рассматриваемом случае игра идет уже «на грани фола», и любое «не угадал» в отношении прототипа немедленно чревато крахом системы. В частности, краеугольным камнем для некоторых системных обработчиков, да и других неэкспортируемых функций ядра будет наличие или отсутствие определения `asmlinkage`: при его наличии параметры вызова будут поочередно (от первого до последнего) заноситься в регистры процессора, а при его отсутствии — заталкиваться в стек (от последнего к первому) вызывающим процессом в соответствии с соглашениями о вызове языка C.

Еще одна из таких мелочей: откуда что взялось и какой смысл вот этих действий?

```
mm_segment_t old_fs = get_fs();
set_fs(KERNEL_DS);                  // буфер в kernel space
...
set_fs(old_fs);                     // восстановить
```

Многие API ядра (и `sys_write` в их числе, а также и большинство системных вызовов) ожидают и *проверяют*, что переданные им на выполнение адреса данных принадлежат *пользовательскому пространству*. Здесь же мы, чтобы «обмануть» API ядра, указываем, что данные находятся в *пространстве ядра*. Мы уже видели

мельком этот трюк ранее, но здесь важно повторить: в предыдущих версиях ядра это делалось совсем другими макросами (в архиве прилагается пример в прежнем синтаксисе).

Выполненный пример порождает еще ряд интересных вопросов, и было бы жалко не удовлетворить любопытство в отношении них. Сработает такой вызов только для вывода в *графический терминал* (X11), или и в *текстовую консоль* (<Ctrl><Alt><F2>, например)? Да, сработает.

А зачем здесь в качестве строки вывода использовалась замысловатая русскоязычная строка, что совсем не приветствуется в программировании ядра? А потому, что в тракте прохождения сообщений от ядра задействовано слишком много последовательных слоев и компонентов (реализация `printk()`, демон журнала, файл журнала, терминальная система UNIX, терминал, визуализатор `dmesg`, ...), и такой вывод может быть хорошей «проверкой на вшивость» согласованности и прозрачности всех этих компонентов. И он любопытен... Проверим предварительно установки:

```
$ echo $LANG
ru_RU.UTF-8
```

И зададим уровень диагностики ниже порога вывода (*только* с терминала root, не `sudo` — мы об этом говорили ранее), иначе просто бессмысленно запускать модуль из консоли:

```
# cat /proc/sys/kernel/printk
3       4       1       7
# echo 8 > /proc/sys/kernel/printk
# cat /proc/sys/kernel/printk
8       4       1       7
```

Выполним *в консоли* все то же действие:

```
# sudo insmod mod_wrc.ko
...
```

И мы увидим в консоли достаточно странную картину:

- вывод `write()` (который выполняется модулем) выведет ожидаемую строку: «адрес...»;
- вывод `dmesg` и `cat` /var/log/messages (выполняемых в *консоли*!) выведет: «адрес...»;
- любимая народная программа hello_world (для проверки) выведет в консоли русскоязычную строку: «Привет мир!»;
- а вот диагностика *на консоль* `printk()` из ядра выведет чудовищную строку с умлаутами (которую я даже не знаю, как откопировать с консоли в текст, чтобы вам показать). Более того, эта строка длиной 77 символов (UNICODE, UTF-8, однако)...

То есть реализация `printk()` в ядре: а) выводит диагностику не через системный журнал, а параллельно демону журналирования; б) пытается как-то интерпретировать поток символов UNICODE, пытаясь преобразовывать их побайтно в ASCII; в) тем самым узувечив символы UNICODE.

Итог этих опытов может быть кратко сформулирован так: добавьте к множеству инструментов программирования, доступных в ядре, набор *всех* системных вызовов POSIX API пространства пользователя. Естественно, толкование результатов некоторых из таких системных вызовов в контексте ядра может быть весьма двусмысленным, а иногда такие результаты и просто бессмысленны. Но сами вызовы — выполнимые!

Подмена системных вызовов

Системные вызовы из пользовательских процессов, как это детально обсуждалось ранее, **все** проходят через таблицу с именем `sys_call_table` (это своего рода case-селектор, который передает управление на обработчик требуемого запроса). Индекс для адреса обработчика каждого системного вызова в этом массиве и определяется *номером* системного вызова. Это и есть таблица входов для связи системных вызовов пространства пользователя с обработчиками этих вызовов в пространстве ядра:

- для 32-битной системы:

```
$ head -n15  /usr/include/asm/unistd_32.h | tail -n12
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
$ cat /usr/include/asm/unistd_32.h | wc -l
430
```

- аналогично для 64-битной системы:

```
$ head -n15  /usr/include/asm/unistd_64.h | tail -n12
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
#define __NR_mprotect 10
#define __NR_munmap 11
```

```
$ cat /usr/include/asm/unistd_64.h | wc -l
352
```

Как видим, системные вызовы в 32- и 64-битных системах отличаются как: а) по числу, так и б) по порядку следования (по номерам аналогичных системных вызовов).

Иногда хотелось бы подменить в таблице позицию (адрес обработчика) или добавить ее — это техника, благополучно известная программистам еще со времен MS-DOS, и применяется в различных операционных системах. Такая модификация бывает нужна, например (этим перечень возможностей далеко не исчерпывается):

- для мониторинга и накопления статистики по какому-либо существующему системному вызову;
- для добавления **собственного** обработчика нового системного вызова, который затем используется прикладными программами пространства пользователя целевого пакета;
- так делают программы-вирусы или недоброжелательные программы, пытающиеся перехватить контроль над компьютером.

Из сказанного уже понятно, что намерение модификации таблицы системных вызовов представляется заманчивым и в общем лежит на поверхности. Но реализовать это (с некоторых пор) становится не так просто! Добавить новый системный вызов можно, в принципе, **двумя** основными способами (все остальное будет какой-то их взаимной комбинацией):

- *статически* — добавив свой файл реализации arch/i386/kernel/new_calls.c в дерево исходных кодов Linux, добавив запись в таблицу системных вызовов arch/i386/kernel/syscall_table.S и включив свою реализацию в сборку ядра, дописав в arch/i386/kernel/Makefile среди многих подобных строку вида:

  ```
  obj-y += new_calls.o
  ```

  После этого мы собираем новое ядро системы, в котором будет реализован требуемый новый системный вызов в пространстве ядра. Весь процесс детально описан в [25]. Но мы не будем обращаться к этому способу просто потому, что это не входит в круг наших интересов;
- *динамически* — во время выполнения, дополнив таблицу `sys_call_table[]` ядра ссылкой на код собственного модуля, который и реализует новый системный вызов (и сделать это действие в пространстве ядра может, естественно, только код модуля ядра).

До определенного времени (ранее версии 2.6) ядро экспортировало адрес таблицы системных вызовов `sys_call_table[]`. Сейчас же этот символ может присутствовать (но может и отсутствовать) в таблице имен ядра (/proc/kallsyms), но не экспортируется для использования модулями:

```
$ cat /proc/kallsyms | grep 'sys_call'
c052476b t proc_sys_call_handler
c07ab3d8 R sys_call_table
```

Тем не менее ядро всегда импортирует символ sys_close[4], находящийся в начальных позициях таблицы sys_call_table[], который экспортируется ядром:

```
$ cat /proc/kallsyms | grep sys_close
c04dea99 T sys_close
```

Некоторые изощренные программы, во множестве варьируемые в форумных обсуждениях, разыскивают это известное значение в сегменте кода ядра, определяют по нему местоположение таблицы sys_call_table[], после чего могут динамически добавлять новые или подменять существующие системные вызовы. Но вам не нужна никакая такая изощренность, если вы детально разберетесь с тем, как ядро экспортирует символы для использования их из кода модулей, и как можно оперировать с символами, не экспортируемыми ядром, — то чем мы занимались ранее.

Итак, мы уже собрали здесь весь арсенал достаточных средств, чтобы это сделать. Для экспериментов выберем системный вызов sys_write. Нам предстоит только определить адрес обработчика этого системного вызова, как мы делали это ранее, и заменить его на свою функцию обработчика. Но загрузка так написанного модуля закончится серьезной неудачей:

```
$ sudo insmod mod_wrchg_1.ko
...
Message from syslogd@notebook at Dec 31 01:56:41 ...
 kernel:CR2: 00000000c07ab3e8
$ dmesg | tail -n100 | grep -v audit
! адрес sys_call_table = c07ab3d8
! адрес в позиции 4[__NR_write] = c04e12fc
! адрес sys_write = c04e12fc
! адрес нового sys_write = fe1f1024
! CR0 = 8005003b
BUG: unable to handle kernel paging request at c07ab3e8
IP: [<fe1f40b6>] wrchg_init+0xb6/0xd4 [mod_wrchg_1]
*pdpt = 0000000000a8c001 *pde = 0000000036881063 *pte = 00000000007ab161
Oops: 0003 [#1] SMP
...
```

И в результате мы получим аварийно установленный модуль, который невозможно будет даже удалить, не прибегая к перезагрузке системы:

```
$ lsmod | grep mod_
mod_wrchg              2732  1
```

Неудача связана с тем, что таблица адресов системных вызовов находится в сегменте *только для чтения*, и попытка записи в нее приводит к ошибке защиты памяти (обращаем внимание на символ R):

```
$ cat /proc/kallsyms | grep sys_call_table
c07ab3d8 R sys_call_table
```

[4] На это обращают внимание многие пишущие на эту тему. Причину такой избирательности я не могу объяснить — мы еще вернемся детально к рассмотрению этого системного вызова далее.

Этому делу можно помочь — голь на выдумки хитра: мы ведь выполняем код модуля в режиме *супервизора*, в нулевом кольце защиты процессора *x*86, где все допустимо! Мы просто на время перезаписи точки входа таблицы `sys_call_table` отменим контроль записи в сегмент, объявленный доступным только для чтения (а затем так же его восстановим). Заметьте, что этот пример работает только для архитектуры *x*86, но и защиту записи мы наблюдаем в *x*86! На другой платформе будут найдены свои аналогичные решения. В архитектуре *i*68 за контроль записи отвечает 16-й бит в управляющем регистре процессора `CR0` (называемый битом `WP`), в архитектуре *x*86-64 картина будет отличаться, и я еще вернусь к этому на пару слов.

Итак, рассмотрим код примера (см. папку sys_call_table/new_sys в сопровождающем книгу файловом архиве), выполняющего требуемое нам действие. Но используемая здесь некоторая функциональность понадобится нам и далее, в других рассматриваемых модулях, поэтому вынесем ее в отдельные включаемые файлы и посмотрим на них внимательнее:

CR0.c

```
// 16 бит WP:          |
//                     V
//   3    2    2    1    1    1    0    0  0
//   1    7    3    9    5    1    7    3  0
//   0000 0000 0000 0001 0000 0000 0000 0000 => 0x00010000
//   1111 1111 1111 1110 1111 1111 1111 1111 => 0xfffeffff

// показать управляющий регистр CR0
#define show_cr0()                          \
{  register unsigned r_eax asm ("eax"); \
   asm("pushl %eax");                   \
   asm("movl %cr0, %eax");              \
   printk("! CR0 = %x\n", r_eax);       \
   asm("popl %eax");                    \
}

//код выключения защиты записи:
#define rw_enable()              \
asm("pushl %eax \n"              \
     "movl %cr0, %eax \n"        \
     "andl $0xfffeffff, %eax \n" \
     "movl %eax, %cr0 \n"        \
     "popl %eax");

//код включения защиты записи:
#define rw_disable()             \
asm("pushl %eax \n"              \
     "movl %cr0, %eax \n"        \
```

```
    "orl $0x00010000, %eax \n"  \
    "movl %eax, %cr0 \n"        \
    "popl %eax");
```

Здесь показан (комментарием) формат управляющего регистра `CR0` для 32-разрядных процессоров и несколько макросов, оперирующих с этим регистром. Макросы: `rw_enable()` (разрешающий запись в сегмент для чтения) и `rw_disable()` (восстанавливающий контроль записи) — реализованы как инлайновые ассемблерные вставки, причем *без параметров*, а поэтому регистры в них можно указать и как `%eax`, и как `%cr0` (с одним префиксом `%`, а не двойным).

В таком виде это работает только на 32-разрядной платформе. Для 64-разрядной платформы все принципиально остается так же, но: а) должны использоваться 64-разрядные операции (суффикс `q` в записи мнемоник команд AT&T); б) вместо регистра `%eax` должен определяться регистр `%rax` и в) другой бит `CR0`, ответственный за защиту записи, — утверждается, что это должно быть:

```
asm("andq $0xfffffffffffeffff, %rax");
...
asm("orq $0x0000000000001000, %rax");
```

К особенностям 64-разрядной платформы мы вернемся к концу обсуждения.

> ***Примечание***
>
> Ключевым действием здесь является *снятие защиты записи* со страницы памяти. Изучение литературы по этой теме показало, что можно предложить как минимум четыре совершенно различных способа добиться этого результата — в условиях (архитектуре), когда не работает один, можно воспользоваться другим. Поскольку эти способы совершенно не очевидные, в *приложении 3* приводится моя статья 2015 года *«Четыре способа писать в защищенную страницу»*, которая расширяет указанные возможности на 64-битное ядро (в кодах) и коды которой выверены непосредственно при подготовке книги.

Другой включаемый файл содержит реализацию функции `find_sym()`, осуществляющей поиск заказанного символа ядра (параметр функции), и возвращает найденный адрес (или возвращает `NULL`, если такого символа в ядре не существует):

find.c

```
static void* find_sym(const char *sym) {
   static unsigned long faddr = 0; // static!!!
   // ----------- вложенная функция - расширение GCC ---------
   int symb_fn(void* data, const char* sym, struct module* mod, unsigned long addr) {
      if(0 == strcmp((char*)data, sym)) {
         faddr = addr;
         return 1;
      }
      else return 0;
   };
```

```
   // ------------------------------------------------------
   kallsyms_on_each_symbol(symb_fn, (void*)sym);
   return (void*)faddr;
}
```

В коде функции `find_sym()` использовано такое синтаксическое расширение GCC (не допускаемое стандартами языка C), как определение вложенной функции `symb_fn()`, локальной по отношению к вызывающей (это очень характерно, например, для языков группы PASCAL Н. Вирта). Такой прием можно счесть и за трюкачество, но он позволил описать возврат адреса любого имени `sym` из таблицы /proc/kallsyms, не прибегая ни к каким глобальным переменным для общего использования.

Теперь мы готовы рассмотреть код самого модуля, который выглядит так:

mod_wrchg.c

```
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/uaccess.h>
#include <linux/unistd.h>

#include "../find.c"
#include "../CR0.c"

asmlinkage long (*old_sys_write) (
       unsigned int fd, const char __user *buf, size_t count);

asmlinkage long new_sys_write (
       unsigned int fd, const char __user *buf, size_t count) {
   int n;
   if(1 == fd) {
      static const char prefix[] = ":-) ";
      mm_segment_t old_fs = get_fs();
      set_fs(KERNEL_DS);
      n = old_sys_write(1, prefix, strlen(prefix));
      set_fs(old_fs);
   }
   n = old_sys_write(fd, buf, count);
   return n;
}
EXPORT_SYMBOL(new_sys_write);

static void **taddr; // адрес таблицы sys_call_table

static int __init wrchg_init(void) {
   void *waddr;
   if((taddr = find_sym("sys_call_table")) != NULL)
      printk("! адрес sys_call_table = %p\n", taddr);
```

```
   else {
      printk("! sys_call_table не найден\n");
      return -EINVAL;
   }
   old_sys_write = (void*)taddr[__NR_write];
   printk("! адрес в позиции %d[__NR_write] = %p\n", __NR_write, old_sys_write);
   if((waddr = find_sym("sys_write")) != NULL)
      printk("! адрес sys_write = %p\n", waddr);
   else {
      printk("! sys_write не найден\n");
      return -EINVAL;
   }
   if(old_sys_write != waddr) {
      printk("! непонятно! : адреса не совпадают\n");
      return -EINVAL;
   }
   printk("! адрес нового sys_write = %p\n", &new_sys_write);
   show_cr0();
   rw_enable();
   taddr[__NR_write] = new_sys_write;
   show_cr0();
   rw_disable();
   show_cr0();
   return 0;
}

static void __exit wrchg_exit(void) {
   printk("! адрес sys_write при выгрузке = %p\n", (void*)taddr[__NR_write]);
   rw_enable();
   taddr[__NR_write] = old_sys_write;
   rw_disable();
   return;
}

module_init(wrchg_init);
module_exit(wrchg_exit);

MODULE_VERSION("3.4");
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");
```

Это самый большой и сложный пример из всех, которые мы рассматривали до сих пор. Но он стоит детального изучения. В нем достаточно много элементов, которые можно отнести к трюкачеству, и из них нужно отметить:

- новый обработчик `new_sys_write()` системного вызова `sys_write` при выводе на `SYSOUT` выводит *предшествующий* выводимой строке собственный префикс (строку `":-) "`), причем для этого ему необходимо сначала использовать сегмент дан-

ных в адресном пространстве ядра (взять данные строки вывода из области ядра) после чего восстановить контроль принадлежности адреса сегменту данных пространства пользователя (для вывода оригинальной переданной строки);

- при выгрузке модуля он должен *обязательно* восстановить прежнюю функцию обработчик `old_sys_write()`;
- в таком восстановлении скрыта потенциальная угроза критической для ядра ошибки, в том случае (крайне редком), если некоторый другой модуль подменит адрес обработчика *после* нашего. То, что здесь показано, годится лишь как иллюстративный упрощенный вариант;
- работа в ядре (а уж тем более с таблицей системных вызовов) — крайне рискованное занятие (как у сапера), поэтому в коде делается двойная перепроверка: адрес обработчика вызова, найденный как символ ядра `sys_write`, сравнивается с адресом в позиции `__NR_write` в таблице `sys_call_table`.

В конечном счете почти все эти элементы встречались ранее, и теперь мы только объединяем их вместе.

Сборка модуля (32-битная система):

```
$ inxi -Sxxx
System:
  Host: antix21 Kernel: 4.9.0-279-antix.1-486-smp arch: i686 bits: 32
    compiler: gcc v: 10.2.1 Desktop: IceWM v: 2.9.7 vt: 7 dm: slimski v: 1.5.0
    Distro: antiX-21_386-base Grup Yorum 31 October 2021
    base: Debian GNU/Linux 11 (bullseye)
$ make
make -C /lib/modules/4.9.0-279-antix.1-486-smp/build
M=/home/olej/2022/kernel/sys_call_table/new_sys modules
make[1]: вход в каталог «/usr/src/linux-headers-4.9.0-279-antix.1-486-smp»
  CC [M]  /home/olej/2022/kernel/sys_call_table/new_sys/mod_wrchg.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/olej/2022/kernel/sys_call_table/new_sys/mod_wrchg.mod.o
  LD [M]  /home/olej/2022/kernel/sys_call_table/new_sys/mod_wrchg.ko
make[1]: выход из каталога «/usr/src/linux-headers-4.9.0-279-antix.1-486-smp»
```

А вот как выглядит протокол выполнения этого примера (в чуть более ранней системе — 3.13.6):

```
$ sudo insmod mod_wrchg.ko
:-) $ :-) e:-) c:-) h:-) o:-)  :-) c:-) т:-) р:-) о:-) к:-) а:-)
:-) строка
:-) $ lsmod | head -n4
:-) :-) Module                  Size  Used by
:-) mod_wrchg               1382  0
:-) fuse                   48375  2
:-) ip6table_filter         2227  0
:-) $ sudo rmmod mod_wrchg
$
```

Мы подменили один из самых используемых при работе с *терминалом* системных вызовов Linux, поэтому приготовьтесь к тому, что объясняться с системой в командной строке станет очень непросто, — даже всего лишь для того, чтобы удалить установленный модуль. Но восстанавливается система после удаления корректно...

Теперь (*восстановив* оригинальный обработчик) можно посмотреть и на то, как это происходило с точки зрения системного журнала:

```
$ dmesg | tail -n120 | grep -v audit
! адрес sys_call_table = c07ab3d8
! адрес в позиции 4[__NR_write] = c04e12fc
! адрес sys_write = c04e12fc
! адрес нового sys_write = fd8ae024
! CR0 = 8005003b
! CR0 = 8004003b
! CR0 = 8005003b
! адрес sys_write при выгрузке = fd8ae024
```

В этом примере специально показывался отладочный вывод содержимого управляющего регистра `CR0` процессора.

Добавление новых системных вызовов

Эта задача очень похожа на предыдущую, причем ее практическое значение может оказаться существенно выше — например, для организации некоторой собственной функциональности в рамках *крупного прикладного* проекта. Один из способов реализации такой возможности упоминался ранее — это добавление кода в ядро с его последующей пересборкой (то, что было названо статической модификацией таблицы системных вызовов). Слабая сторона такого решения состоит в его плохой переносимости: проект сложно устанавливать на систему заказчика, ядро которой должно быть модифицировано. Больший интерес должна представлять возможность проделать это динамически.

В отличие от обсуждавшегося ранее примера подмены системного вызова, эта задача, при всем ее сходстве, имеет некоторые отягчающие особенности:

- размер оригинальной таблицы системных вызовов `sys_call_table` произвольно увеличивается от версии к версии ядра (да и зависит от конкретной процессорной платформы);
- константа, задающая размерность таблицы (известная в ядре как `__NR_syscall_max`), является препроцессорной константой периода компиляции и неизвестна на периоде выполнения (по крайней мере, мне неизвестна);
- пытаясь добавить собственный системный вызов, мы не имеем права выйти за пределы существующей таблицы.

Размер таблицы `sys_call_table` достаточно велик и меняется от версии к версии ядра:

```
$ cat /proc/kallsyms | grep ' sys_' | grep T | wc -l
345
```

Примечание

Это достаточно грубая оценка только порядка величины — некоторые обработчики в современных версиях подменены на другие их формы. Показательным примером этого является обработчик (ptregs_fork) вызова fork() в одной из начальных (__NR_fork = 2) позиций sys_call_table:

```
$ cat /proc/kallsyms | grep ptregs_fork
c040929c t ptregs_fork
$ cat /proc/kallsyms | grep sys_fork
c040ee13 T sys_fork
```

Смягчает указанные ограничивающие обстоятельства то, что таблица системных вызовов *неплотная* — в ней есть неиспользуемые позиции (оставшиеся от устаревших системных вызовов или зарезервированные на будущее). Все такие позиции заполнены одним адресом — указателем на функцию обработчика нереализованных вызовов sys_ni_syscall:

```
$ cat /proc/kallsyms | grep sys_ni_syscall
c045b9a8 T sys_ni_syscall
```

Следуя таким путем, мы можем добавить свой новый обработчик системного вызова в *любую* неиспользуемую позицию таблицы sys_call_table. Текстуально (в исходном коде, *статически*) структуру таблицы мы можем детально рассмотреть *для интересующей нас платформы* в дереве исходных кодов ядра. Для образца воспользуемся деревом ядра 3.0.9 (в приведенном коде показаны только неиспользуемые позиции, а комментарии вида __NR_# в конце строк добавлены мной):

```
$ cat /usr/src/linux-3.0.9/arch/x86/kernel/syscall_table_32.S
ENTRY(sys_call_table)
        .long sys_restart_syscall     /* 0 - old "setup()" system call, used for restarting */
        .long sys_exit
        .long ptregs_fork
...
        .long sys_ni_syscall    /* old break syscall holder */       //17
        .long sys_ni_syscall    /* old stty syscall holder */        //31
        .long sys_ni_syscall    /* old gtty syscall holder */        //32
        .long sys_ni_syscall    /* 35 - old ftime syscall holder */  //35
        .long sys_ni_syscall    /* old prof syscall holder */        //44
        .long sys_ni_syscall    /* old lock syscall holder */        //53
        .long sys_ni_syscall    /* old mpx syscall holder */         //56
        .long sys_ni_syscall    /* old ulimit syscall holder */      //58
        .long sys_ni_syscall    /* old profil syscall holder */      //98
        .long sys_ni_syscall    /* old "idle" system call */         //112
        .long sys_ni_syscall    /* old "create_module" */            //127
        .long sys_ni_syscall    /* 130: old "get_kernel_syms" */     //130
        .long sys_ni_syscall    /* reserved for afs_syscall */       //137
        .long sys_ni_syscall    /* Old sys_query_module */           //167
        .long sys_ni_syscall    /* reserved for streams1 */          //188
        .long sys_ni_syscall    /* reserved for streams2 */          //189
```

```
        .long sys_ni_syscall    /* reserved for TUX */              //222
        .long sys_ni_syscall                                        //223
        .long sys_ni_syscall                                        //251
        .long sys_ni_syscall    /* sys_vserver */                   //273
        .long sys_ni_syscall    /* 285 */ /* available */           //285
...
        .long sys_setns                                             // 346
```

Здесь мы видим, что для этой версии ядра таблица имеет 347 позиций, из которых 21 не задействована (и никогда не может быть задействована, потому что назначить новый системный вызов устаревшему старому — это слишком рискованный ход: никто не гарантирует систему от выполнения весьма старых программ, а последствия этого были бы непредсказуемы для ядра).

Анализу неиспользуемых позиций (в динамике) и будет посвящен первый рассматриваемый модуль (см. папку sys_call_table/add_sys в сопровождающем книгу файловом архиве — содержимое папки add_sys выделено из папки new_sys только для того, чтобы не загромождать проект, и задействует совместно используемые включаемые файлы, рассмотренные ранее):

ni-test.c

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

static unsigned long asct = 0, anif = 0;

int symb_fn(void* data, const char* sym, struct module* mod, unsigned long addr) {
   if(0 == strcmp("sys_call_table", sym))
      asct = addr;
   else if(0 == strcmp("sys_ni_syscall", sym))
      anif = addr;
   return 0;
}

#define SYS_NR_MAX_OLD 340
// - этот размер таблицы взят довольно произвольно из ядра 2.6.37
static void show_entries(void) {
   int i, ni = 0;
   char buf[ 200 ] = "";
   for(i = 0; i <= SYS_NR_MAX_OLD; i++) {
      unsigned long *taddr = ((unsigned long*)asct) + i;
      if(*taddr == anif) {
         ni++;
         sprintf(buf + strlen(buf), "%03d, ", i);
      }
   }
```

```
   printk("! найдено %d входов: %s\n", ni, buf);
}

static int __init init_driver(void) {
   kallsyms_on_each_symbol(symb_fn, NULL);
   printk("! адрес таблицы системных вызовов = %lx\n", asct);
   printk("! адрес нереализованных вызовов  = %lx\n", anif);
   if(0 == asct || 0 == anif) {
      printk("! не найдены символы ядра\n");
      return -EFAULT;
   }
   show_entries();
   return -EPERM;
}
module_init(init_driver);

MODULE_DESCRIPTION("test unused entries");
MODULE_AUTHOR("Oleg Tsiliuric <olej@front.ru>");
MODULE_LICENSE("GPL");
```

Код достаточно простой, и, не вдаваясь в обсуждение деталей, смотрим, что он покажет (выполнение в ядре 2.6.32):

```
$ sudo insmod ni-test.ko
insmod: error inserting 'ni-test.ko': -1 Operation not permitted
$ dmesg | tail -n 18 | grep !
! найдено 26 входов: 017, 031, 032, 035, 044, 053, 056, 058, 098, 112, 127, 130, 137, 167,
188, 189, 222, 223, 251, 273, 274, 275, 276, 285, 294, 317,
```

Резюме:

- почти 10% размера таблицы системных вызовов не используются;
- стабильность этого списка очень высока (сознательно для анализа кода была выбрана версия 3.0.9, а для исполнения в динамике — версия 2.6.32, отстоящие друг от друга более чем на 2 года выпуска);

Теперь мы готовы создать модуль, реализующий новый системный вызов, и программу пользовательского пространства (процесс), использующий такой вызов. Номер нового вызова определен в общем заголовочном файле, для согласованности используемом обоими программами:

syscall.h

```
// номер нового системного вызова
#define __NR_own 223
// может быть взят любой, полученный при загрузке модуля ni-test.ko
// для ядра 2.6.32 был получен ряд:
// 017, 031, 032, 035, 044, 053, 056, 058, 098, 112, 127, 130, 137,
// 167, 188, 189, 222, 223, 251, 273, 274, 275, 276, 285, 294, 317,
```

Прежде всего создадим тестовую программу, использующую новый системный вызов, например так:

syscall.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "syscall.h"

static void do_own_call(char *str) {
   int n = syscall(__NR_own, str, strlen(str));
   if(n >= 0)
      printf("syscall return %d\n", n);
   else
      printf("syscall error %d : %s\n", n, strerror(-n));
}

int main(int argc, char *argv[]) {
   char str[] = "DEFAULT STRING";
   if(1 == argc) do_own_call(str);
   else
      while(--argc > 0) do_own_call(argv[ argc ]);
   return EXIT_SUCCESS;
};
```

Программа может делать один или серию (если использовать несколько параметров в командной строке) системных вызовов и передает символьный параметр в вызов (подобно тому, как это делает, например `sys_write`) — для того чтобы в реализационной части системного вызова показать, как эта строка копируется из пространства пользователя в пространство ядра (или могла бы возвращаться обратно). Но главный интерес здесь представляет код возврата: успех или неудача выполнения системного вызова.

А вот и модуль, подхватывающий выполнение этого системного вызова в пространстве ядра:

add-sys.c

```
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/uaccess.h>
#include <linux/unistd.h>

#include "../find.c"
#include "../CR0.c"
#include "syscall.h"
```

```
// системный вызов с двумя параметрами:
asmlinkage long (*old_sys_addr) (const char __user *buf, size_t count);

asmlinkage long new_sys_call (const char __user *buf, size_t count) {
   static char buf_msg[ 80 ];
   int res = copy_from_user(buf_msg, (void*)buf, count);
   buf_msg[ count ] = '\0';
   printk("! передано %d байт: %s\n", count, buf_msg);
   return res;
};
EXPORT_SYMBOL(new_sys_call);

static void **taddr; // адрес таблицы sys_call_table

static int __init new_sys_init(void) {
   void *waddr;
   if((taddr = find_sym("sys_call_table")) != NULL)
      printk("! адрес sys_call_table = %p\n", taddr);
   else {
      printk("! sys_call_table не найден\n");
      return -EINVAL;
   }
   old_sys_addr = (void*)taddr[ __NR_own ];
   printk("! адрес в позиции %d[__NR_own] = %p\n", __NR_own, old_sys_addr);
   if((waddr = find_sym("sys_ni_syscall")) != NULL)
      printk("! адрес sys_ni_syscall = %p\n", waddr);
   else {
      printk("! sys_ni_syscall не найден\n");
      return -EINVAL;
   }
   if(old_sys_addr != waddr) {
      printk("! непонятно! : адреса не совпадают\n");
      return -EINVAL;
   }
   printk("! адрес нового sys_call = %p\n", &new_sys_call);
   rw_enable();
   taddr[ __NR_own ] = new_sys_call;
   rw_disable();
   return 0;
}

static void __exit new_sys_exit(void) {
   printk("! адрес sys_call при выгрузке = %p\n", (void*)taddr[ __NR_own ]);
   rw_enable();
   taddr[ __NR_own ] = old_sys_addr;
   rw_disable();
   return;
}
```

```
module_init(new_sys_init);
module_exit(new_sys_exit);

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

Здесь также делается двойная проверка (перестраховка) соответствия адреса в заданной (__NR_own) позиции таблицы sys_call_table адресу sys_ni_syscall.

И теперь оцениваем то, что у нас получилось:

```
$ ./syscall
syscall return 38 : Function not implemented
```

Это было до загрузки модуля, реализующего требуемый программе системный вызов. Загружаем такой модуль:

```
$ sudo insmod add-sys.ko
$ lsmod | grep 'sys'
add_sys                 1432  0
$ dmesg | tail -n 30 | grep !
! адрес sys_call_table = c07ab3d8
! адрес в позиции 223[__NR_own] = c045b9a8
! адрес sys_ni_syscall = c045b9a8
! адрес нового sys_call = fdd7c024
$ ./syscall новые аргументы
syscall return 0 : Success
syscall return 0 : Success
```

Программа сделала два системных вызова:

```
$ dmesg | tail -n 30 | grep !
! адрес sys_call_table = c07ab3d8
! адрес в позиции 223[__NR_own] = c045b9a8
! адрес sys_ni_syscall = c045b9a8
! адрес нового sys_call = fdd7c024
! передано 18 байт: аргументы
! передано 10 байт: новые
```

Выгружаем реализующий модуль:

```
$ sudo rmmod add-sys
$ dmesg | tail -n 50 | grep !
! адрес sys_call_table = c07ab3d8
! адрес в позиции 223[__NR_own] = c045b9a8
! адрес sys_ni_syscall = c045b9a8
! адрес нового sys_call = fdd7c024
! передано 18 байт: аргументы
! передано 10 байт: новые
! адрес sys_call при выгрузке = fdd7c024
```

И повторяем выполнение только что успешно выполнявшейся программы:

```
$ ./syscall 1 2 3
syscall return 38 : Function not implemented
syscall return 38 : Function not implemented
syscall return 38 : Function not implemented
```

Ядро не в состоянии поддержать более выполнение требуемого программе системного вызова!

По образу и подобию показанного в системе может быть установлен произвольный новый обработчик системного вызова.

Скрытый обработчик системного вызова

Интересен сам по себе вопрос: может ли модуль в принципе установить обработчик (подменив существующий или добавив новый) таким образом, чтобы ядро системы «не знало» об этом, не выявляя такого модуля средствами диагностики `lsmod` или в файловой системе /proc? Интерес этот чисто прагматический — ответ на поставленный вопрос определяет, могут ли вредоносные программы произвести подобные изменения. И ответ на него — положительный! Для этого такой модуль должен:

- выделить динамически блок памяти для кода функции-обработчика нового системного вызова (а, возможно, и отдельный блок для собственных данных);
- переместить код функции-обработчика в такую динамическую область, возможно, установив *признак выполнимости* (в 64-битной или PAE-архитектуре) для страниц этой динамической памяти;
- установить в таблице `sys_call_table` адрес обработчика на этот *перемещенный* код;
- завершить функцию инициализации модуля с кодом, отличным от нуля, — тем самым модуль фактически не устанавливается и не фиксируется ядром.

Идея здесь основана на том, что блоки памяти, выделяемые динамически запросами `kmalloc()` и другими (мы подробно рассматривали это ранее), продолжают существовать, даже если модуль, их создавший, прекратил существование. В этом случае в памяти возникают объекты, к которым потеряны все пути доступа и которые не могут быть никаким образом уничтожены. Подобные механизмы, если они присутствуют (сознательно или по ошибке), ведут к постепенной деградации операционной системы. Примеры их использования проясняют такие ситуации лучше, чем десятки увещеваний «на словах».

Так что рассмотрим все это на примере (см. папку sys_call_table/hidden в сопровождающем книгу файловом архиве), который во многом аналогичен предыдущему, — он использует те же включаемые файлы определений, а также очень подобный тестовый процесс пространства пользователя:

hidden.c

```
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/uaccess.h>
#include <linux/unistd.h>
#include <../arch/x86/include/asm/cacheflush.h>

#include "../find.c"
#include "../CR0.c"
#include "syscall.h"

// описания функций-обработчиков для разных вариантов:
// asmlinkage long new_sys_call(const char __user *buf, size_t count)
#define VARNUM 5
#ifndef VARIANT
#define VARIANT 0
#endif
#if VARIANT>VARNUM
#undef VARIANT
#define VARIANT 0
#endif

static long shift; // величина сдвига тела функции системного обработчика
#if VARIANT == 0
   #include "null.c"
#elif VARIANT == 1
   #include "local.c"
#elif VARIANT == 2
   #include "getpid.c"
#elif VARIANT == 3
   #include "strlen_1.c"
#elif VARIANT == 4
   #include "strlen_2.c"
#elif VARIANT == 5
   #include "write.c"
#else
   #include "null.c"
#endif

static int __init new_sys_init(void) {
   void *waddr, *move_sys_call,
       **taddr;                  // адрес таблицы sys_call_table
   size_t sys_size;
   printk("!... SYSCALL=%d, VARIANT=%d\n", NR, VARIANT);
   if((taddr = find_sym("sys_call_table")) != NULL)
      printk("! адрес sys_call_table = %p\n", taddr);
```

```
   else
      return -EINVAL | printk("! sys_call_table не найден\n");
   if(NULL == (waddr = find_sym("sys_ni_syscall")))
      return -EINVAL | printk("! sys_ni_syscall не найден\n");
   if(taddr[ NR ] != waddr)
      return -EINVAL | printk("! системный вызов %d занят\n", NR);
   {  unsigned long end;
      asm("movl $L2, %%eax \n"
          :"=a"(end)::
         );
      sys_size = end - (long)new_sys_call;
      printk("! статическая функция: начало= %p, конец=%lx, длина=%d \n",
              new_sys_call, end, sys_size);
  }
   // выделяем блок памяти под функцию-обработчик
   move_sys_call = kmalloc(sys_size, GFP_KERNEL);
   if(!move_sys_call) return -EINVAL | printk("! memory allocation error!\n");
   printk("! выделен блок %d байт с адреса %p\n", sys_size, move_sys_call);
   // копируем резидентный код нового обработчика в выделенный блок памяти
   memcpy(move_sys_call, new_sys_call, sys_size);
   shift = move_sys_call - (void*)new_sys_call;
   printk("! сдвиг кода = %lx байт\n", shift);
   // снять бит NX-защиты со страницы
   //int set_memory_x(unsigned long addr, int numpages);
   set_memory_x((long unsigned)move_sys_call, sys_size / PAGE_SIZE + 1);
   printk("! адрес нового sys_call = %p\n", move_sys_call);
   rw_enable();
   taddr[ NR ] = move_sys_call;
   rw_disable();
   return -EPERM;
}
module_init(new_sys_init);

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

Вот, собственно, и всё. Но при рассмотрении разных вариантов сборки кода следует учесть наличие файлов (null.c, local.c и им подобных), которые содержат разные реализации функции-обработчика нового системного вызова с единым именем и прототипом:

```
asmlinkage long new_sys_call(const char __user *buf, size_t count);
```

Прототип этого нового системного вызова выбран достаточно произвольно — так же как и все существующие системные вызовы, различающиеся параметрами и их числом. Простейший обработчик (null.c) показывает только принципиальную возможность, поэтому устанавливаемый им обработчик системного вызова с номером

NR (определяется параметром сборки SYSCALL и по умолчанию равен 223) ничего осознанного не делает, но только возвращает признак нормального завершения:

null.c

```
asmlinkage long new_sys_call(const char __user *buf, size_t count) {
   asm("movl $0, %%eax\n"  // эквивалент return 0;
       "popl %%ebp    \n"
       "ret           \n"
       "L2: nop       \n"  // нам нужна метка L2 после return
       ::: "%eax"
      );
   return 0;               // только для синтаксиса
};
```

Поскольку код такого модуля портит таблицу sys_call_table и не может ее восстанавливать, а при многократном выполнении будет оставлять после себя блоки потерянной (навсегда!) памяти, то нам нужно обзавестись дуальным к нему модулем, который восстанавливает первичное состояние таблицы:

restore.c

```
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/uaccess.h>
#include <linux/unistd.h>

#include "../find.c"
#include "../CR0.c"
#include "syscall.h"

static int __init new_sys_init(void) {
   void **taddr;                  // адрес таблицы sys_call_table
   void *waddr, *old_sys_addr;
   if((taddr = find_sym("sys_call_table")) != NULL)
      printk("! адрес sys_call_table = %p\n", taddr);
   else return -EINVAL | printk("! sys_call_table не найден\n");
   if((waddr = find_sym("sys_ni_syscall")) != NULL)
      printk("! адрес sys_ni_syscall = %p\n", waddr);
   else return -EINVAL | printk("! sys_ni_syscall не найден\n");
   old_sys_addr = (void*)taddr[ NR ];
   printk("! адрес в позиции %d = %p\n", NR, old_sys_addr);
   if(old_sys_addr != waddr) {
      kfree(old_sys_addr);
      rw_enable();
      taddr[ NR ] = waddr;  // восстановить sys_ni_syscall
      rw_disable();
```

```
      printk("! итоговый адрес обработчика %p\n", taddr[ NR ]);
   }
   else
      printk("! итоговый адрес обработчика не изменяется\n");
   return -EPERM;
}
module_init(new_sys_init);

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

Теперь у нас есть всё, чтобы проверить, как это работает:

```
$ make VARIANT=0
...
$ ./syscall
syscall return -38 [ffffffda], reason: Function not implemented
$ sudo insmod hidden.ko
insmod: error inserting 'hidden.ko': -1 Operation not permitted
$ lsmod | grep hid
$
$ ./syscall 1 23 456
syscall return 0 [00000000], reason: Success
syscall return 0 [00000000], reason: Success
syscall return 0 [00000000], reason: Success
$ dmesg | tail -n60 | grep !
!... SYSCALL=223, VARIANT=0
! адрес sys_call_table = c07ab3d8
! статическая функция: начало= f7ecd000, конец=f7ecd00f, длина=15
! выделен блок 15 байт с адреса d9322030
! сдвиг кода = e1455030 байт
! адрес нового sys_call = d9322030
$ sudo insmod restore.ko
insmod: error inserting 'restore.ko': -1 Operation not permitted
$ ./syscall
syscall return -38 [ffffffda], reason: Function not implemented
```

Как легко видеть, модуль hidden.ko в системе не установлен, но новый системный вызов с номером `NR` замечательно отрабатывается.

Единственное место в коде hidden.c, с подобным которому мы до сих пор не встречались и которое требует минимальных комментариев, — это строка, в которой производится вызов:

```
int set_memory_x(unsigned long addr, int numpages);
```

В старших моделях *x*86, работающих в 64-битном режиме или в расширенном режиме PAE, введен бит `NX` защиты страницы памяти от выполнения (старший бит в записи таблицы страниц). В ядре Linux этот аппаратный механизм защиты применяется, начиная с версии 2.6.30. Вызов `set_memory_x()` снимает ограничение вы-

полнения для numpages последовательных страниц (размером PAGESIZE каждая), начиная со страницы адреса addr. Аналогично вызов восстанавливает защиту страницы от выполнения. Для 32-битного ядра *x*86 (не PAE!) этот механизм, как уже было сказано, не используется.

Ассемблерная вставка в функции обработчика представляет полный эквивалент оператора return 0; (в полном соответствии с соглашениями языка C — с выталкиванием регистра %ebp из стека...). Понадобился такой эквивалент *только* потому, что нам нужна метка L2 (ее адрес) после оператора return, а компилятор gcc такие метки после завершения кода «оптимизирует». Это место не должно смущать. Весь целевой код обработчика должен предшествовать этой вставке.

Сложность написания кода для таких обработчиков очень высока, но, в принципе, все это реализуемо. Фактически при этом предстоит вручную, без помощи компилятора gcc (опция -fPIC), реализовать нечто подобное PIC-кодированию (Position Independed Code) — позиционно-независимому (в памяти) кодированию. Такое написание обработчиков само по себе любопытно, и некоторым его вопросам будет посвящено все остальное изложение до конца раздела. Если вас не интересуют эти детали, вы можете безболезненно опустить всю его оставшуюся часть.

Сложности перемещаемой функции обработчика связаны с тем, что:

- функция не может использовать никакие внешние переменные, описанные вне ее тела, — после завершения функции инициализации модуля все такие области памяти будут освобождены. Могут использоваться только локальные переменные в стеке;
- точно то же самое относится и к другим (локальным) функциям, описанным в модуле. Здесь приходит на помощь такое расширение gcc, как *вложенные описания функций*, которые будут перемещены вместе с телом объемлющей их функции-обработчика.

Пример сказанного — следующий обработчик:

local.c

```
asmlinkage long new_sys_call(const char __user *buf, size_t count) {
   long res = 1000;
   int inc(int in) {            // вложенное описание функции
      return ++in;
   }
   res = res + inc(count);
   res = inc(count);
   asm("movl %%ebx, %%eax\n\t"   // эквивалент return res;
        "leave            \n\t"
        "ret              \n\t"
        "L2: nop          \n\t"  // нам нужна метка L2 после return
        ::"b"(res):"%eax"
      );
   return 0;                     // только для синтаксиса
};
```

И результат использования такого обработчика:

```
$ make VARIANT=1
...
$ sudo insmod hidden.ko
insmod: error inserting 'hidden.ko': -1 Operation not permitted
$ lsmod | grep hid
$ ./syscall
syscall return 16 [00000010], reason: Success
$ ./syscall 123
syscall return 5 [00000005], reason: Success
$ ./syscall 1 22 333 4444
syscall return 6 [00000006], reason: Success
syscall return 5 [00000005], reason: Success
syscall return 4 [00000004], reason: Success
syscall return 3 [00000003], reason: Success
```

Следующая сложность состоит в том, что функции API ядра, экспортируемые или не экспортируемые ядром (`strlen()`, `printk()`, `sys_getpid()`, `sys_write()`, ...), в этом коде нельзя вызывать в привычном виде, записав просто вызов функции по ее имени. Адрес такого вызова разрешится в момент *статической* линковки (смещение адреса запишется в поле команды), а при вызове приведет к вызову со смещением и краху выполнения. Вот примитивный пример того, как это может быть сделано работоспособно:

getpid.c

```
//c044e690 T sys_getpid
asmlinkage long new_sys_call(const char __user *buf, size_t count) {
   long res = 0;
   long (*own_getpid)(void);
   own_getpid = 0xc044e690UL;
   res = own_getpid();
   asm("leave            \n"
       "ret               \n"  // эквивалент return res;
       "L2: nop           \n"  // нам нужна метка L2 после return
       ::"a"(res):
      );
   return 0;                   // только для синтаксиса
}
```

Адрес (неэкспортируемый) функции `sys_getpid()` в нашем случае для простоты взят *статически* из /proc/kallsym, но он может находиться и более сложными способами — динамически — как это обсуждалось ранее:

```
$ sudo insmod hidden.ko
insmod: error inserting 'hidden.ko': -1 Operation not permitted
$ ./syscall
```

```
syscall return 19783 [00004d47], reason: Success
$ ./syscall 12 13
syscall return 19793 [00004d51], reason: Success
syscall return 19793 [00004d51], reason: Success
$ ps -A | tail -n3
19792 ?        00:00:00 sleep
19796 pts/12   00:00:00 ps
19797 pts/12   00:00:00 tail
```

Другой, более реалистичный пример того же рода:

strlen_2.c

```
asmlinkage long new_sys_call(const char __user *buf, size_t count) {
   long res = 0;
   size_t own_strlen(const char* ps) { // вложенная функция
      long res = 0;
      asm(
        "movl    8(%%ebp), %%eax \n"     // ps -> call parameter
        "movl    %%eax, (%%esp)  \n"
        "call   *%%ebx           \n"
        :"=a"(res):"b"((long)(&strlen)): // strlen() из API ядра
      );
      return res;
   }
   res = own_strlen(buf);
   asm("leave            \n"
        "ret              \n"            // эквивалент return res;
        "L2: nop          \n"            // нам нужна метка L2 после return
        ::"a"(res):
      );
   return 0;                             // только для синтаксиса
}
```

Этот же пример демонстрирует, как функция использует строчную переменную *из пространства пользователя*, адрес которой передан ей в системном вызове:

```
$ make VARIANT=4
...
$ sudo insmod hidden.ko
insmod: error inserting 'hidden.ko': -1 Operation not permitted
$ lsmod | grep hid
$
$ ./syscall 1 23 456 'новая строка'
syscall return 24 [00000018], reason: Success
syscall return 4 [00000004], reason: Success
syscall return 3 [00000003], reason: Success
syscall return 2 [00000002], reason: Success
```

```
$ dmesg | tail -n60 | grep !
!... SYSCALL=223, VARIANT=4
! адрес sys_call_table = c07ab3d8
! статическая функция: начало= f7ede000, конец=f7ede018, длина=24
! выделен блок 24 байт с адреса d9085940
! сдвиг кода = e11a7940 байт
! адрес нового sys_call = d9085940
```

Этот же пример вскрывает очередную неприятность при написании подобных перемещаемых функций: мы успешно использовали ранее *скалярные* (`int`, `long`, ...) локальные переменные, объявленные внутри функции. Но это не относится к массивам, в частности к строкам, размещенным как локальные переменные в теле функции.

Модифицируем вызов в предыдущем примере так:

```
res = own_strlen("1234567");
```

Или вот так:

```
char str[ 80 ] = "1234567";
res = own_strlen(str);
```

Этим мы переведем его в неработоспособное состояние: хотя указатель строки размещен в стеке и доступен, но указываемая строка размещается где-то отдельно (в сегменте данных), и значение ее указателя оказывается некорректным.

А вот еще один любопытный и работоспособный пример:

write.c

```
//c04e12fc T sys_write
asmlinkage long new_sys_call(const char __user *buf, size_t count) {
   long res = 0;
   asmlinkage size_t (*own_write)(int fd, const char* s, size_t l);
   own_write = (void*)0xc04e12fc;
   res = own_write(1, buf, count);
   asm("leave            \n"
       "ret               \n"  // эквивалент return res;
       "L2: nop           \n"  // нам нужна метка L2 после return
       ::"a"(res):
     );
   return 0;                   // только для синтаксиса
};

$ make VARIANT=5
...
$ sudo insmod hidden.ko
insmod: error inserting 'hidden.ko': -1 Operation not permitted
$ lsmod | grep hid
$
```

```
$ ./syscall 1 23 456 'новая строка'
новая строка
syscall return 24 [00000018], reason: Success
456
syscall return 4 [00000004], reason: Success
23
syscall return 3 [00000003], reason: Success
1
syscall return 2 [00000002], reason: Success
```

Здесь системный вызов не только корректно принимает переданную ему строку (подобно тому, как это делает `sys_write`), но и выводит параметр своего вызова, эту строку, не в системный журнал, а *на терминал*.

В папке sys_call_table/hidden сопровождающего книгу файлового архива приведены для рассмотрения еще несколько дополнительных вариантов написания такого перемещаемого обработчика.

Динамическая загрузка модулей

До сих пор мы устанавливали собранные модули выполнением команды `insmod` — когда это происходило на этапе разработки, или `modprobe` — когда модуль отлажен и инсталлирован с системе. Симметрично мы удаляли все установленные модули командой `rmmod`. Во всех этих случаях операции над модулями производятся *статически*. Возникает вопрос: а можно ли модули устанавливать (и выгружать) *динамически* — т. е. по требованию, из собственного программного кода? Раз это `insmod` умеет делать — то и мы сможем!

Это вызывает встречный вопрос: а зачем это надо? Понадобится это в самых разнообразных случаях и по разным причинам:

- программы-утилиты `insmod`, `modprobe` и `rmmod` сами по себе являются программами пользовательского пространства, и просто любопытно, как ими в коде выполняются подобные действия;
- в некоторых случаях от системы требуется некая функциональность, которая на текущий момент не предоставляется и которая обеспечивается подгружаемым модулем ядра. В таких случаях было бы в высшей степени удобно подгрузить такой модуль непосредственно по требованию его использования. Классический и общеизвестный пример такой ситуации — команда монтирования в Linux файловой подсистемы (типа qnx4, minix и др.), которая не подгружена в системе по умолчанию, например:

```
$ lsmod | grep minix
$ sudo mount -t minix /dev/sda5 /mnt/sda5
$ ls /mnt/sda5
bin  boot  dev  etc  home  mnt  proc  root  sbin  tmp  usr  var
$ lsmod | grep minix
minix                19212  1
```

В этом случае модуль (minix.ko) подгружается также по запросу из *пользовательской* утилиты `mount`;

- разработчики определенного класса оборудования могли бы иметь *родовой* (generic) модуль драйвера, который подгружал бы по необходимости *видовой* модуль драйвера под конкретную модель оборудования в этом классе. Классическими примерами такого случая (возможно, и решаемых в каждой ситуации другими средствами) являются целые семейства драйверов (по *типу* плат) в таких интерфейсах к платам *класса* E1/T1 в IP-телефонии, как интерфейс DAHDI (компании Digium) или интерфейс компании Sangoma. В этом случае типовые модули динамически подгружаются из среды родового *модуля ядра*;
- развитием предыдущего подхода может быть создана техника построения плагинов — возможность расширения функциональности сложной системы посредством добавления к ней новых специфических частей, не затрагивая переделками код существующих ее частей. В области пространства пользователя такая возможность реализуется за счет использования разделяемых библиотек. В области *модулей ядра* эта возможность могла бы реализоваться посредством динамической загрузки модулей.

Этими случаями покрываются далеко не все области, где пригодилась бы динамическая загрузка модулей.

...из процесса пользователя

Для загрузки модуля из пространства пользовательского процесса (как это делают `insmod`, `modprobe` и `rmmod`) существует *системный вызов* `sys_init_module()`:

```
asmlinkage long sys_init_module(void __user *umod, unsigned long len, const char __user *uargs);
```

Для выгрузки модуля — соответственно *системный вызов* `sys_delete_module()`:

```
asmlinkage long sys_delete_module(const char __user *name, unsigned int flags);
```

Но... Всё, что касается операций динамических загрузки и выгрузки модулей, покрыто изрядным мраком, man-описания и существующие в Интернете посты описывают какие-то устаревшие и мутные реализации, относящиеся ко временам на границе версий ядра 2.4 и 2.6. Так что здесь придется изрядно поэкспериментировать и пособирать воедино оговорки и намеки, разбросанные по исходным текстам ядра Linux.

Поскольку эта задача актуальна практически и не экзотична, я покажу ее в свежей инсталляции:

```
$ inxi -S
System:    Host: R420 Kernel: 5.4.0-124-generic x86_64 bits: 64 Desktop: Cinnamon 5.2.7
Distro: Linux Mint 20.3 Una
```

Итак, вот интересующие нас системные вызовы (`sys_init_module` и `sys_delete_module` — вскоре они нам понадобятся) на случай 32- и 64-битных систем соответственно:

```
$ grep _module /usr/include/x86_64-linux-gnu/asm/unistd_32.h
#define __NR_create_module 127
#define __NR_init_module 128
#define __NR_delete_module 129
#define __NR_query_module 167
#define __NR_finit_module 350
$ grep _module /usr/include/x86_64-linux-gnu/asm/unistd_64.h
#define __NR_create_module 174
#define __NR_init_module 175
#define __NR_delete_module 176
#define __NR_query_module 178
#define __NR_finit_module 313
```

Отметим еще раз в уме, что *номера* системных вызовов в 32- и 64-битных системах отличаются, — они просто не коррелированы (поэтому при необходимости употребляйте в коде не числовые номера, а не поленитесь потратить время и найти соответствующую символьную константу вида `__NR_*`).

Для всех примеров (см. папку load_module/umaster в сопровождающем книгу файловом архиве) нам нужен тестовый подопытный модуль, который мы и будет динамически загружать и выгружать:

slave.c

```
#include "../common.c"

static char* parm1 = "";
module_param(parm1, charp, 0);

static char* parm2 = "";
module_param(parm2, charp, 0);

static char this_mod_file[40];

static int __init mod_init(void) {
   set_mod_name(this_mod_file, __FILE__);
   printk("+ модуль %s загружен: parm1=%s, parm2=%s\n", this_mod_file, parm1, parm2);
   return 0;
}

static void __exit mod_exit(void) {
   printk("+ модуль %s выгружен\n", this_mod_file);
}
```

Модуль ссылается на общую часть ../common.c — здесь и во всех последующих примерах, связанных с динамической загрузкой модулей (для сокращения объемов тривиальных кодов), использован общий включаемый файл. Вот эта часть:

common.c

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Oleg Tsiliuric <olej.tsil@gmail.com>");

static int  __init mod_init(void);
static void __exit mod_exit(void);

module_init(mod_init);
module_exit(mod_exit);

inline void __init set_mod_name(char *res, char *path) {
   char *pb = strrchr(path, '/') + 1,
        *pe = strrchr(path, '.');
   strncpy(res, pb, pe - pb);
   sprintf(res + (pe - pb), "%s", ".ko");
}
```

Чтобы привыкнуть к модулю, собрав его, поработаем с ним автономно — понаблюдаем на будущее, как внешне выглядит именно его загрузка/выгрузка:

```
$ sudo insmod slave.ko
$ dmesg | tail -n1
[43464.851759] + модуль slave.ko загружен: parm1=, parm2=
$ sudo rmmod slave
$ dmesg | tail -n2
[43464.851759] + модуль slave.ko загружен: parm1=, parm2=
[43516.110535] + модуль slave.ko выгружен
```

При необходимости модуль готов принять до двух символьных параметров (нам никоим образом не нужны сами параметры, но мы должны быть уверены, что наш будущий загрузчик не препятствует этому):

```
$ sudo insmod slave.ko parm1='строка1' parm2='строка2'
$ dmesg | tail -n1
[53593.085751] + модуль slave.ko загружен: parm1=строка1, parm2=строка2
$ sudo rmmod slave
$ dmesg | tail -n2
[53593.085751] + модуль slave.ko загружен: parm1=строка1, parm2=строка2
[53624.451325] + модуль slave.ko выгружен
```

Вот и всё, что от подопытного модуля требуется. Важно то, что если модулю передать не те параметры, которые он ожидает, или не в том формате их записи, то он будет нещадно ругаться и не станет загружаться.

Теперь у нас все готово для того, чтобы приступить к изготовлению загрузчиков, работающих из пользовательского пространства. В качестве загрузчика соберем *приложение*:

inst1.c

```
#include <sys/stat.h>
#include <errno.h>
#include "common.h"

//  asmlinkage long (*sys_init_module)          // системный вызов sys_init_module()
//                  (void __user *umod, unsigned long len, const char __user *uargs);

int main(int argc, char *argv[]) {
   char parms[80] = "", file[80] = SLAVE_FILE;
   void *buff = NULL;
   int fd, res;
   off_t fsize;         /* общий размер в байтах */
   if(argc > 1) {
      strcpy(file, argv[1]);
      if(argc > 2) {
         int i;
         for(i = 2; i < argc; i++) {
            strcat(parms, argv[i]);
            strcat(parms, " ");
         }
      }
   }
   printf("загрузка модуля: %s %s\n", file, parms);
   fd = open(file, O_RDONLY);
   if(fd < 0) {
      printf("ошибка open: %m\n");
      return errno;
   }
   {  struct stat fst;
      if(fstat(fd, &fst) < 0) {
         printf("ошибка stat: %m\n");
         close(fd);
         return errno;
     }
     if(!S_ISREG(fst.st_mode)) {
         printf("ошибка: %s не файл\n", file);
         close(fd);
         return EXIT_FAILURE;
     }
     fsize = fst.st_size;
   }
   printf("размер файла модуля %s = %ld байт\n", file, fsize);
   buff = malloc(fsize);
   if(NULL == buff) {
      printf("ошибка malloc: %m\n");
```

```
      close(fd);
      return errno;
   }
   if(fsize != read(fd, buff, fsize)) {
      printf("ошибка read: %m\n");
      free(buff);
      close(fd);
      return errno;
   }
   close(fd);
   res = syscall(__NR_init_module, buff, fsize, parms);
   free(buff);
   if(res < 0) printf("ошибка загрузки: %m\n");
   else printf("модуль %s успешно загружен!\n", file);
   return res;
}
```

Приложения пользовательского пространства (это и другие) включают файл common.h (а не упоминавшийся ранее common.c!):

common.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <fcntl.h>

#define SLAVE_FILE "./slave.ko";
```

Здесь, главным образом, определяется имя файла загружаемого модуля (`./slave.ko`), чтобы его не вводить каждый раз при тестировании в виде параметра командной строки. Приложение громоздкое, но логика его крайне проста:

1. Ищется файл откомпилированного модуля ядра с указанным именем (переменная `file`).
2. Определяется полный размер этого файла (переменная `fsize`).
3. В точности под этот размер динамически выделяется буфер чтения `buff`.
4. В буфер читается *образ* загружаемого модуля (в формате объектного файла `.ko`).
5. Системному вызову `sys_init_module(void* umod, unsigned long len, const char* uargs)` передаются адрес образа модуля в памяти (`buff` — первый параметр) и его длина (`fsize` — второй параметр).
6. Если необходимо, в качестве третьего параметра `sys_init_module()` передается строка параметров загрузки модуля. Если параметры не используются, третьим параметром указывается пустая строка, но ни в коем случае не `NULL`.

Сразу же изготовим симметричное приложение для *выгрузки* модулей (эквивалент `rmmod` — от `rmmod` его отличает то, что мы наперед знаем имя модуля, который собираемся выгружать, поэтому не станем себя утруждать каждый раз набором этого имени в качестве параметра запуска нашего приложения):

rem1.c

```
#include "common.h"

// asmlinkage long (*sys_delete_module)    // системный вызов sys_delete_module()
//                 (const char __user *name, unsigned int flags);
// flags: O_TRUNC, O_NONBLOCK

int main(int argc, char *argv[]) {
   char file[80]  = SLAVE_FILE;
   int res;

   if(argc > 1) strcpy(file, argv[1]);
   char *slave_mod = strrchr(file, '/') != NULL ?
                     strrchr(file, '/') + 1 :
                     file;
   if(strrchr(file, '.') != NULL)
      *strrchr(file, '.') = '\0';
   printf("выгружается модуль %s\n", slave_mod);
   res = syscall(__NR_delete_module, slave_mod, O_TRUNC);
   if(res < 0) printf("ошибка выгрузки: %m\n");
   else printf("модуль %s успешно выгружен!\n", slave_mod);
   return res;
}
```

А теперь рассмотрим, как это все работает:

```
$ ls -l slave.ko
-rw-rw-r-- 1 olej olej 6544 авг 20 23:19 slave.ko
$ sudo ./inst1 slave.ko parm1='строка1' parm2='строка2'
загрузка модуля: slave.ko parm1=строка1 parm2=строка2
размер файла модуля slave.ko = 6544 байт
модуль slave.ko успешно выгружен!
$ dmesg | tail -n1
[50509.921854] + модуль slave.ko загружен: parm1=строка1, parm2=строка2
$ lsmod | head -n2
Module                  Size  Used by
slave                  16384  0
$ sudo ./rem1
выгружается модуль slave
модуль slave успешно выгружен!
$ dmesg | tail -n2
```

```
[50509.921854] + модуль slave.ko загружен: parm1=строка1, parm2=строка2
[50681.215028] + модуль slave.ko выгружен
```

Обратим внимание на то, как размер модуля, диагностируемый загрузчиком (он же размер файла модуля), отличается от диагностики размера загруженного модуля командой `lsmod`.

Нас могут упрекнуть, что мы, мол, достаточно грубо используем непрямой системный вызов `syscall()` (что не поощряется в культурном программировании) и напрямую манипулируем номерами системных вызовов `__NR_init_module` и `__NR_delete_module`. Но их можно в конечном итоге заменить вызовами *стандартной* системной библиотеки для этих же `syscall()`, заменив в листингах всего по паре строк (но здесь выявляется одна очень интересная подробность, о которой я расскажу позже):

inst2.c

```
extern int init_module(void *module_image, unsigned long len,
                       const char *param_values);
...
   res = init_module(buff, fsize, parms); // вызов sys_init_module()
...
```

rem2.c

```
extern int delete_module(const char *name, int flags);
...
   res = delete_module(slave_mod, O_TRUNC);  // flags: O_TRUNC, O_NONBLOCK
...
```

А теперь сборка и выполнение с тем же результатом:

```
$ sudo ./inst2 slave.ko parm1='строка1' parm2='строка2'
загрузка модуля: slave.ko parm1=строка1 parm2=строка2
размер файла модуля slave.ko = 6544 байт
модуль slave.ko успешно загружен!
$ dmesg | tail -n1
[50755.109764] + модуль slave.ko загружен: parm1=строка1, parm2=строка2
$ dmesg | tail -n1
[50755.109764] + модуль slave.ko загружен: parm1=строка1, parm2=строка2
$ lsmod | head -n2
Module                  Size  Used by
slave                  16384  0
$ sudo ./rem2
выгружается модуль slave
модуль slave успешно выгружен!
$ dmesg | tail -n2
[50755.109764] + модуль slave.ko загружен: parm1=строка1, parm2=строка2
[50799.318937] + модуль slave.ko выгружен
```

Примечание

Далее приводится то, о чем я только что обещал рассказать позже. Кому вкусные тонкости не интересны, могут пропустить дальнейший текст и перейти к следующему разделу...

Итак, почему я сразу не показал листинги с `init_module()` и `delete_module()` и зачем морочу вам голову с непрямыми системными вызовами `syscall()`? Да по очень простой причине — *нигде*, ни в литературе, ни в справочных руководствах Linux, ни в Интернете мне не удалось найти ни образцов корректного использования `init_module()` и `delete_module()`, ни даже их корректных прототипов вызова. Напротив, все источники полнятся просто синтаксически некорректными и устаревшими примерами (относящимися к давним неактуальным ядрам 2.4.*x*). Поэтому восстанавливать примеры их использования пришлось именно обратным реинжинирингом через `syscall()`. При компиляции показанных примеров без строк с `extern ...` (которые фактически не нужны) вы будете получать предупреждения вида:

```
$ make
gcc     inst2.c   -o inst2
inst2.c: In function 'main':
inst2.c:53:10: warning: implicit declaration of function 'init_module'; did you mean
'SYS_init_module'? [-Wimplicit-function-declaration]
   53 |    res = init_module(buff, fsize, parms); // вызов sys_init_module()
      |          ^~~~~~~~~~~
      |          SYS_init_module
gcc     rem2.c   -o rem2
rem2.c: In function 'main':
rem2.c:14:10: warning: implicit declaration of function 'delete_module'; did you mean
'SYS_delete_module'? [-Wimplicit-function-declaration]
   14 |    res = delete_module(slave_mod, O_TRUNC);  // flags: O_TRUNC, O_NONBLOCK
      |          ^~~~~~~~~~~~~
      |          SYS_delete_module
```

А при работе с ядром предупреждения — это уже знак больших неприятностей, мы говорили об этом ранее! И эта ситуация заняла у меня уйму времени... В системном `man` мы можем прочитать относительно заголовочных файлов довольно-таки странный текст:

```
$ man 2 init_module
...
Note: glibc provides no header file declaration of init_module() and no wrapper function
for finit_module();
...
```

С другой стороны, проследим за используемой стандартной библиотекой языка C:

```
$ ldd rem1
    linux-vdso.so.1 (0x00007ffe611c6000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f441a39d000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f441a5bc000)
$ pwd
/lib/x86_64-linux-gnu
```

```
$ ls -l libc.so.6
lrwxrwxrwx 1 root root 12 апр  7 04:24 libc.so.6 -> libc-2.31.so
$ sudo nm -D libc-2.31.so | grep ' T ' | grep _module
000000000011fa30 T create_module
000000000011fa60 T delete_module
0000000000025aa0 T __gconv_get_modules_db
000000000011fb50 T init_module
000000000011fd60 T query_module
```

В результате выясняется, что интересующие нас *библиотечные* вызовы замечательным образом присутствуют в библиотеке, но отсутствуют в заголовочных файлах этой (`libc`) библиотеки! Мне оставалось только восполнить код `extern` *прототипами* двух библиотечных вызовов, реконструированных по их определениям из ранних разрозненных источников (в рудиментных версиях ядра 2.4.*x* это было!).

Вывод: с *некоторых времен* разработчики ядра не желали афишировать операции с модулями ядра из пользовательского пространства (я бы на их месте тоже этого не желал), которыми пользуются тем не менее стандартные системные команды `insmod` и `rmmod`. И они убрали прототипы этих функций из заголовочных файлов, но оставили их самих в составе библиотеки!

В завершение — пара слов о 32-битных системах... Показанный код в неизменном виде столь же успешно собирается и выполняется и в 32-битных системах с достаточно свежим ядром:

```
$ sudo inxi -Sxxx
System:    Host: lmde32 Kernel: 5.10.0-17-686 i686 bits: 32 compiler: gcc v: 10.2.1
           Console: tty 2 wm: Mutter
           DM: LightDM 1.26.0 Distro: LMDE 5 Elsie base: Debian 11.2 bullseye
$ sudo ./inst2 slave.ko parm1='строка1' parm2='строка2'
загрузка модуля: slave.ko parm1=строка1 parm2=строка2
размер файла модуля slave.ko = 116096 байт
модуль slave.ko успешно загружен!
$ dmesg | tail -n1
[74051.068542] + модуль slave.ko загружен: parm1=строка1, parm2=строка2
$ sudo ./rem2
выгружается модуль slave
модуль slave успешно выгружен!
$ dmesg | tail -n2
[74051.068542] + модуль slave.ko загружен: parm1=строка1, parm2=строка2
[74089.458651] + модуль slave.ko выгружен
```

...из модуля ядра

В предыдущем разделе мы загружали (и выгружали) модуль slave.ko динамически из кода приложения. Теперь нам предстоит сделать то же самое, но уже из кода *вызывающего* модуля (master.ko). Вот образец такого модуля (см. папку load_module/master в сопровождающем книгу файловом архиве):

master.c

```
#include <linux/version.h>
#include <linux/fs.h>
#include <linux/vmalloc.h>
#include "../common.c"
#include "../find.c"

static char* file = "./slave.ko";
module_param(file, charp, 0);

static char this_mod_file[40],                  // имя файла master-модуля
            slave_name[80];                     // имя файла slave-модуля

static int __init mod_init(void) {
   long res = 0, len;
   struct file *f;
   void *buff;
   ssize_t n;
   asmlinkage long (*sys_init_module)           // системный вызов sys_init_module()
              (void __user *umod, unsigned long len, const char __user *uargs);
   set_mod_name(this_mod_file, __FILE__);
   if((sys_init_module = find_sym("sys_init_module")) == NULL) {
      printk("> sys_init_module не найден\n");
      res = -EINVAL;
      goto end;
   }
   printk("> адрес sys_init_module = %p\n", sys_init_module);
   strcpy(slave_name, file);
   f = filp_open(slave_name, O_RDONLY, 0);
   if(IS_ERR(f)) {
      printk("> ошибка открытия файла %s\n", slave_name);
      res = -ENOENT;
      goto end;
   }
   len = vfs_llseek(f, 0L, 2);                  // 2 - means SEEK_END
   if(len <= 0) {
      printk("> ошибка lseek\n");
      res = -EINVAL;
      goto close;
   }
   printk("> длина файла модуля = %d байт\n", (int)len);
   if(NULL == (buff = vmalloc(len))) {
      res = -ENOMEM;
      goto close;
   };
   printk("> адрес буфера чтения = %p\n", buff);
   vfs_llseek(f, 0L, 0);                        // 0 - means SEEK_SET
```

```
#if (LINUX_VERSION_CODE < KERNEL_VERSION(4, 14, 0))
// extern int kernel_read(struct file *, loff_t, char *, unsigned long);
   n = kernel_read(f, 0, buff, len);
#else
// extern ssize_t kernel_read(struct file *, void *, size_t, loff_t *);
   n = kernel_read(f, buff, len, NULL);
#endif
   printk("> считано из файла %s %d байт\n", slave_name, n);
   if(n != len) {
      printk("> ошибка чтения\n");
      res = -EIO;
      goto free;
   }
   {  mm_segment_t old_fs = get_fs();
      set_fs(KERNEL_DS);
      res = sys_init_module(buff, len, "");
      set_fs(old_fs);
      if(res < 0) {
                  printk("> ошибка зарузки модуля\n");
                  goto insmod;
           }
   }
   printk("> модуль %s загружен: file=%s\n", this_mod_file, file);
insmod:
free:
   vfree(buff);
close:
   filp_close(f, NULL);
end:
   return res;
}

static void __exit mod_exit(void) {
   asmlinkage long (*sys_delete_module)         // системный вызов sys_delete_module()
              (const char __user *name, unsigned int flags);
   // flags: O_TRUNC, O_NONBLOCK
   char *slave_mod = strrchr(slave_name, '/') != NULL ?
                     strrchr(slave_name, '/') + 1 :
                     slave_name;
   *strrchr(slave_mod, '.') = '\0';
   printk("> выгружается модуль %s\n", slave_mod);
   if((sys_delete_module = find_sym("sys_delete_module")) == NULL) {
      printk("> sys_delete_module не найден\n");
      return;
   }
   printk("> адрес sys_delete_module = %p\n", sys_delete_module);
   {  long res = 0;
```

```
      mm_segment_t old_fs = get_fs();
      set_fs(KERNEL_DS);
      res = sys_delete_module(slave_mod, 0);
      set_fs(old_fs);
      if(res < 0) {
         printk("> ошибка выгрузки модуля %s\n", slave_mod);
         return;
      }
   }
   printk("> модуль %s выгружен\n", this_mod_file);
}
```

Логика модуля в точности повторяет логику рассмотренного ранее приложения пользовательского пространства, только здесь, как всегда в коде ядра, все делается гораздо осторожнее и использован другой инструментарий. На завершающем этапе нам неизвестны адреса системных обработчиков `sys_init_module()` и `sys_delete_module()` — они *не экспортируются* ядром. Поэтому финалы (загрузки и выгрузки) выглядят так:

1. Адреса символов ядра `sys_init_module` и `sys_delete_module` разыскиваются функцией `find_sym()` (мы подобное не раз рассматривали ранее).
2. Выполняется косвенный вызов по указателям функций `sys_init_module` и `sys_delete_module`.
3. Эти адреса присваиваются функциональным переменным: `(*sys_init_module)(...)` и `(*sys_delete_module)(...)` (вообще-то, имена эти могли бы быть *произвольными* и совпали с именами системных вызовов... случайно).

Это и есть требуемые обработчики системных вызовов.

А теперь то, как все сказанное выглядит на деле... Я покажу это в 32-битной системе (4.19), но адаптировать приведенный код под вашу систему не составляет большого труда (оставим это в качестве домашнего задания на самостоятельную проработку):

```
$ inxi -S
System:
  Host: antix21 Kernel: 4.9.0-279-antix.1-486-smp arch: i686 bits: 32
    Desktop: IceWM v: 2.9.7
    Distro: antiX-21_386-base Grup Yorum 31 October 2021
olej@antix21:~/2022/kernel/load_module/master
$ uname -a
Linux antix21 4.9.0-279-antix.1-486-smp #1 SMP Sun Aug 8 20:59:37 EEST 2021 i686 GNU/Linux
olej@antix21:~/2022/kernel/load_module/master
```

Загрузка модулей «по цепочке» — каскадная:

```
$ ls -l *.ko
-rw-r--r-- 1 olej olej 7096 авг 21 16:10 master.ko
-rw-r--r-- 1 olej olej 4180 авг 21 16:10 slave.ko
```

```
$ sudo insmod master.ko
$ lsmod | head -n4
Module                  Size  Used by
slave                  16384  0
master                 16384  0
nls_utf8               16384  1
$ dmesg | tail -n6
[242902.333158] > адрес sys_init_module = c90d2ee0
[242902.333164] > длина файла модуля = 4180 байт
[242902.333246] > адрес буфера чтения = f8079000
[242902.333250] > считано из файла ./slave.ko 4180 байт
[242902.333944] + модуль slave.ko загружен: parm1=, parm2=
[242902.334116] > модуль master.ko загружен: file=./slave.ko
```

Здесь хорошо видно, зачем мы сделали отличающиеся префиксы, `>` и `+` — легко различимо, из какого модуля идет вывод. Модуль slave загружен позже, чем модуль master, и между ними нет никаких зависимостей, что правильно.

И выгрузка модулей:

```
$ sudo rmmod master
$ dmesg | tail -n4
[242965.479623] > выгружается модуль slave
[242965.480158] > адрес sys_delete_module = c90d0240
[242965.480160] + модуль slave.ko выгружен
[242965.484380] > модуль master.ko выгружен
$ lsmod | head -n2
Module                  Size  Used by
nls_utf8               16384  1
```

В заключение я повторю весь цикл происходящих действий — от начала загрузки двух модулей и заканчивая их выгрузкой:

```
$ dmesg | tail -n10
[242902.333158] > адрес sys_init_module = c90d2ee0
[242902.333164] > длина файла модуля = 4180 байт
[242902.333246] > адрес буфера чтения = f8079000
[242902.333250] > считано из файла ./slave.ko 4180 байт
[242902.333944] + модуль slave.ko загружен: parm1=, parm2=
[242902.334116] > модуль master.ko загружен: file=./slave.ko
[242965.479623] > выгружается модуль slave
[242965.480158] > адрес sys_delete_module = c90d0240
[242965.480160] + модуль slave.ko выгружен
[242965.484380] > модуль master.ko выгружен
```

Подключаемые плагины

Теперь у нас есть всё, чтобы создать макет использования отдельных модулей, динамически подключаемых по требованию к основному проекту в качестве плагинов. Создадим сознательно утрированный пример (см. папку load_module/plugin

в сопровождающем книгу файловом архиве), но он будет работать с динамическими модулями так же энергично, как и при любой реальной потребности:

1. Добавим новый системный вызов `__NR_str_trans` (мы это уже легко умеем делать):

syscall.h

```
// номер нового системного вызова
#define __NR_str_trans 223
```

2. Пользовательский процесс передает этим системным вызовом корневому модулю (master.ko) строку, изображающую некоторое числовое значение (в различных системах счисления: 8, 10, 16), и ожидает получить обратно вычисленное числовое значение:

syscall.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "syscall.h"

static void do_own_call(char *str) {
   int n = syscall(__NR_str_trans, str, strlen(str));
   if(n >= 0)
      printf("syscall return %d\n", n);
   else
      printf("syscall error %d : %s\n", n, strerror(-n));
}

int main(int argc, char *argv[]) {
   if(1 == argc) do_own_call("9876");
   else {
      int i;
      for(i = 1; i < argc; i++)
         do_own_call(argv[ i ]);
   }
   return EXIT_SUCCESS;
}
```

3. Корневой модуль (master.ko) устанавливает в таблицу системных вызовов *новый* системный обработчик (`__NR_str_trans`), но не занимается непосредственно переводом полученной строки в численное значение, — в зависимости от синтаксиса записи этой строки-параметра (восьмеричное, десятичное или шестнадцатеричное значение), он выбирает соответствующий модуль трансляции (oct.ko, dec.ko или hex.ko) для вычислений и *загружает* этот модуль.

Все модули-плагины экспортируют *одно и то же имя* точки входа (`str_translate`), поэтому никакие два из обрабатывающих модулей-плагинов *не могут быть загружены одновременно*.

4. Загрузив модуль-плагин, корневой модуль должен найти экспортируемое тем имя `str_translate()` и передать ему строку на обработку — полученный результат и будет итогом работы нового системного вызова.
5. После обработки полученного запроса модуль-плагин должен быть тут же динамически выгружен — во избежание конфликта экспортируемых имен при следующей загрузке модуля.

Модули-обработчики, как легко понять, — совершенно однотипные. Для еще большего сокращения объема все они используют общее включение:

slave.c

```
#include "../common.c"

static char this_mod_file[ 40 ];

long str_translate(const char *buf);
EXPORT_SYMBOL(str_translate);
```

Файл ../common.c мы уже встречали раньше, да и сами три плагина-обработчика имеют схожий вид (они отличаются только видом функции-обработчика `str_translate()`):

oct.c

```
#include "slave.c"

static const char dig[] = "01234567";

long str_translate(const char *buf) {
   long res = 0;
   const char *p = buf;
   printk("+ %s : запрос : %s\n", this_mod_file, buf);
   while(*p != '\0') {
      char *s = strchr(dig, *p);
      if(s == NULL) return -EINVAL;
      res = res * 8 + (s - dig);
      p++;
   }
   return res;
}
```

```
static int __init mod_init(void) {
   set_mod_name(this_mod_file, __FILE__);
   printk("+ модуль %s загружен\n", this_mod_file);
   return 0;
}

static void __exit mod_exit(void) {
   printk("+ модуль %s выгружен\n", this_mod_file);
}
```

dec.c

```
...
static const char dig[] = "0123456789";

long str_translate(const char *buf) {
   long res = 0;
   const char *p = buf;
   printk("+ %s : запрос : %s\n", this_mod_file, buf);
   while(*p != '\0') {
      char *s = strchr(dig, *p);
      if(s == NULL) return -EINVAL;
      res = res * 10 + (s - dig);
      p++;
   }
   return res;
};
...
```

hex.c

```
...
static const char digh[] = "0123456789ABCDEF",
                  digl[] = "0123456789abcdef";

long str_translate(const char *buf) {
   long res = 0;
   const char *p = buf;
   printk("+ %s : запрос : %s\n", this_mod_file, buf);
   while(*p != '\0') {
      char *s;
      int val;
      s = strchr(digh, *p);
      if(s != NULL)
         val = s - digh;
      else {
         s = strchr(digl, *p);
```

```
            if(s == NULL) return -EINVAL;
            val = s - digl;
         }
         res = res * 16 + val;
         p++;
      }
      return res;
};
...
```

Такая примитивная однотипность очень помогает отследить суть происходящего. Мы получили на этом шаге три идентичных модуля (dec.ko, hex.ko, oct.ko):

```
$ ls -l *.ko
-rw-rw-r-- 1 olej olej  95223 Фев 12 15:13 dec.ko
-rw-rw-r-- 1 olej olej  95553 Фев 12 15:13 hex.ko
-rw-rw-r-- 1 olej olej 126348 Фев 12 15:13 master.ko
-rw-rw-r-- 1 olej olej  95223 Фев 12 15:13 oct.ko
```

Каждый из этих трех модулей экспортирует *одно и то же* имя точки входа `str_translate()`. Поэтому любые два из таких модулей *не могут* быть загружены одновременно, в чем легко убедиться:

```
$ sudo insmod hex.ko
$ dmesg | tail -n30 | grep +
+ модуль hex.ko загружен
$ cat /proc/kallsyms | grep T | grep translate
c0579604 T isofs_name_translate
c063f888 T set_translate
c063f8a4 T inverse_translate
f8ab2000 T str_translate  [hex]
$ sudo insmod oct.ko
insmod: error inserting 'oct.ko': -1 Invalid module format
$ dmesg | tail -n30 | grep oct
oct: exports duplicate symbol str_translate (owned by hex)
```

Это интересный эксперимент вообще относительно экспортируемых символов ядра. Но невозможность загрузки таких модулей *одновременно* не означает невозможность их использования вовсе. Нам нужна просто некоторая внешняя программная оболочка, которая сможет загружать эксклюзивно каждый из этих модулей по требованию.

Функции такого программного обрамления и выполняет корневой модуль, загружающий модули-плагины и замыкающий всю конфигурацию программных компонентов:

master.c

```
#include <linux/fs.h>
#include <linux/vmalloc.h>
```

```
#include "syscall.h"
#include "../common.c"
#include "../find.c"
#include "CR0.c"

static char this_mod_file[ 40 ];      // имя файла master-модуля

static void **taddr,                  // адрес таблицы sys_call_table
            *old_sys_addr;            // адрес старого обработчика (sys_ni_syscall)

asmlinkage long (*sys_init_module)    // системный вызов sys_init_module()
                (void __user *umod, unsigned long len, const char __user *uargs);
asmlinkage long (*sys_delete_module)  // системный вызов sys_delete_module()
                (const char __user *name, unsigned int flags);

static long load_slave(const char* fname) {
   long res = 0;
   struct file *f;
   long len;
   void *buff;
   size_t n;
   f = filp_open(fname, O_RDONLY, 0);
   if(IS_ERR(f)) {
      printk("+ ошибка открытия файла %s\n", fname);
      return -ENOENT;
   }
   len = vfs_llseek(f, 0L, 2); // 2 - means SEEK_END
   if(len <= 0) {
      printk("+ ошибка lseek\n");
      return -EINVAL;
   }
   printk("+ длина файла модуля = %d байт\n", (int)len);
   if(NULL == (buff = vmalloc(len))) {
      filp_close(f, NULL);
      return -ENOMEM;
   };
   printk("+ адрес буфера чтения = %p\n", buff);
   vfs_llseek(f, 0L, 0);       // 0 - means SEEK_SET
   n = kernel_read(f, 0, buff, len);
   printk("+ считано из файла %s %d байт\n", fname, n);
   if(n != len) {
      printk("+ ошибка чтения\n");
      vfree(buff);
      filp_close(f, NULL);
      return -EIO;
   }
```

```
   filp_close(f, NULL);
   { mm_segment_t fs = get_fs();
     set_fs(get_ds());
     res = sys_init_module(buff, len, "");
     set_fs(fs);
   }
   vfree(buff);
   if(res < 0)
      printk("+ ошибка загрузки модуля %s : %ld\n", fname, res);
   return res;
}

static long unload_slave(const char* fname) {
   long res = 0;
   mm_segment_t fs = get_fs();
   set_fs(get_ds());
   if(strrchr(fname, '.') != NULL)
      *strrchr(fname, '.') = '\0';
   res = sys_delete_module(fname, 0);
   set_fs(fs);
   if(res < 0)
      printk("+ ошибка выгрузки модуля %s\n", fname);
   return res;
}

// новый системный вызов
asmlinkage long sys_str_translate(const char __user *buf, size_t count) {
   static const char* slave_name[] =          // имена файлов slave-модулей
                     { "dec.ko", "oct.ko", "hex.ko" };
   static char buf_msg[ 80 ], mod_file[ 40 ], *par1;
   int res = copy_from_user(buf_msg, (void*)buf, count), ind, trs;
   buf_msg[ count ] = '\0';
   long (*loaded_str_translate)(const char *buf);
   printk("+ системный запрос %d байт: %s\n", count, buf_msg);
   if(buf_msg[ 0 ] == '0') {
      if(buf_msg[ 1 ] == 'x') ind = 2; // hex
      else ind = 1;                    // oct
   }
   else if(strchr("123456789", buf_msg[ 0 ]) != 0)
      ind = 0;                         //dec
   else return -EINVAL;
   strcpy(mod_file, slave_name[ ind ]);
   par1 = buf_msg + ind;
   if((res = load_slave(mod_file)) < 0) return res;
   if((loaded_str_translate = find_sym("str_translate")) != NULL)
      printk("+ адрес обработчика = %p\n", loaded_str_translate);
```

```
   else {
      printk("+ str_translate не найден\n");
      return -EINVAL;
   }
   if((trs = loaded_str_translate(par1)) < 0)
      return trs;
   printk("+ вычислено значение %d\n", trs);
   res = unload_slave(mod_file);
   if(res < 0) return res;
   else return trs;
};
static int __init mod_init(void) {
   long res = 0;
   void *waddr;
   set_mod_name(this_mod_file, __FILE__);
   if((taddr = find_sym("sys_call_table")) != NULL)
      printk("+ адрес sys_call_table = %p\n", taddr);
   else {
      printk("+ sys_call_table не найден\n");
      return -EINVAL;
   }
   old_sys_addr = (void*)taddr[ __NR_str_trans ];
   printk("+ адрес в позиции %d[__NR_str_trans] = %p\n", __NR_str_trans, old_sys_addr);
   if((waddr = find_sym("sys_ni_syscall")) != NULL)
      printk("+ адрес sys_ni_syscall = %p\n", waddr);
   else {
      printk("+ sys_ni_syscall не найден\n");
      return -EINVAL;
   }
   if(old_sys_addr != waddr) {
      printk("+ непонятно! : адреса не совпадают\n");
      return -EINVAL;
   }
   printk("+ адрес нового sys_call = %p\n", &sys_str_translate);
   if((waddr = find_sym("sys_init_module")) == NULL) {
      printk("+ sys_init_module не найден\n");
      return -EINVAL;
   }
   printk("+ адрес sys_init_module = %p\n", waddr);
   sys_init_module = waddr;
   if((waddr = find_sym("sys_delete_module")) == NULL) {
      printk("+ sys_delete_module не найден\n");
      return -EINVAL;
   }
   printk("+ адрес sys_delete_module = %p\n", waddr);
   sys_delete_module = waddr;
   rw_enable();
```

```
   taddr[ __NR_str_trans ] = sys_str_translate;
   rw_disable();
   printk("+ модуль %s загружен\n", this_mod_file);
   return res;
}

static void __exit mod_exit(void) {
   printk("+ адрес syscall при выгрузке = %p\n", (void*)taddr[ __NR_str_trans ]);
   rw_enable();
   taddr[ __NR_str_trans ] = old_sys_addr;
   rw_disable();
   printk("+ восстановлен адрес syscall = %p\n", old_sys_addr);
   printk("+ модуль %s выгружен\n", this_mod_file);
   return;
}
```

И вот как выглядит работа модуля:

```
$ sudo insmod master.ko
$ ./syscall 0x77
syscall error -1 : Operation not permitted
$ dmesg | tail -n30 | grep +
+ адрес sys_call_table = c07ab3d8
+ адрес в позиции 223[__NR_str_trans] = c045b9a8
+ адрес sys_ni_syscall = c045b9a8
+ адрес нового sys_call = f99db024
+ адрес sys_init_module = c0470f50
+ адрес sys_delete_module = c046f4e8
+ модуль master.ko загружен
$ sudo ./syscall 077
syscall return 63
$ dmesg | tail -n30 | grep +
+ системный запрос 3 байт: 077
+ длина файла модуля = 95223 байт
+ адрес буфера чтения = f9c09000
+ считано из файла oct.ko 95223 байт
+ модуль oct.ko загружен
+ адрес обработчика = f9b83000
+ oct.ko : запрос : 77
+ вычислено значение 63
+ модуль oct.ko выгружен
$ sudo ./syscall 77
syscall return 77
$ dmesg | tail -n30 | grep +
+ системный запрос 2 байт: 77
+ длина файла модуля = 95223 байт
+ адрес буфера чтения = f9c41000
```

```
+ считано из файла dec.ko 95223 байт
+ модуль dec.ko загружен
+ адрес обработчика = f9c75000
+ dec.ko : запрос : 77
+ вычислено значение 77
+ модуль dec.ko выгружен
$ sudo ./syscall 0x77
syscall return 119
$ dmesg | tail -n30 | grep +
+ системный запрос 4 байт: 0x77
+ длина файла модуля = 95553 байт
+ адрес буфера чтения = f9c7b000
+ считано из файла hex.ko 95553 байт
+ модуль hex.ko загружен
+ адрес обработчика = f9caf000
+ hex.ko : запрос : 77
+ вычислено значение 119
+ модуль hex.ko выгружен
$ sudo ./syscall z77
syscall error -1 : Operation not permitted
$ sudo rmmod master
$ lsmod | head -n4
Module                  Size  Used by
minix                  19212  1
fuse                   48375  2
ip6table_filter         2227  0
$ dmesg | tail -n37 | grep +
+ адрес syscall при выгрузке = f99db024
+ восстановлен адрес syscall = c045b9a8
+ модуль master.ko выгружен
$ sudo ./syscall 0x77
syscall error -1 : Operation not permitted
```

Этот пример выводит так много промежуточных результатов, что дополнительно комментировать его работу нет необходимости. Может возникнуть закономерный последний вопрос: а где же здесь возможность дополнять в проект модули-плагины, не затрагивая код корневого модуля? Ее здесь в явном виде нет. Но стоит вынести соответствие критерия выбора (переменная `ind`) используемого плагина *с именем файла* модуля (массив `slave_name[]`) в некоторый текстовый конфигурационный файл, а читать такие файлы (из модуля) мы научились несколькими разделами ранее, — и вы получаете динамически расширяемую систему (система, которая *не знает* всех своих компонентов в момент запуска в эксплуатацию). Это элементарно просто, а не показано в и так предельно громоздком примере, только чтобы еще дальше его не усложнять.

Обсуждение

Вся эта глава — о нетривиальных возможностях модулей ядра — написана, конечно, не в намерении поощрить и развить хакерские наклонности некоторых читателей по части написания вирусов или другого вредоносного программного обеспечения[5], — в этом им воспрепятствует в первую очередь требование наличия привилегий root для операций с модулями и защищенность самой операционной системы (это вам не Windows!). Показаны эти возможности с тем, чтобы подвести итоги нашему рассмотрению и прийти к пониманию того, что:

- в пространстве ядра можно выполнить *практически все*! Модули являются полноценной составной частью ядра, поэтому сказанное относится и к ним в полной мере (представьте себе, как могли бы существовать в пользовательском пространстве некие возможности, недостижимые в ядре, если *все* такие возможности тому же пользовательскому пространству предоставляет ядро);
- код модуля выполняется в привилегированном режиме (режим супервизора, кольцо защиты 0 для *x*86-архитектуры), поэтому ему доступны *любые* операции, которые недоступны пользовательскому коду: привилегированные операции, работа с управляющими регистрами процессора (`CR0`—`CR4` для *x*86 и другие регистры MSR, Model Specific Register), работа с таблицами страниц, со структурами менеджера памяти MMU и многое другое... фактически *всё*;
- раз существует такая дуальность возможностей для пользовательских процессов и для ядра, то и API для использования таких возможностей столь же дуальны. Для пользовательских процессов — это стандарты POSIX API (поздние расширения), а для ядра — API ядра (которые ничем не стандартизованы). Те и другие отличаются по форме (имена вызовов и структур данных, прототипы вызовов, число параметров и др.), но подобны друг другу. Если при написании модулей у вас возникают затруднения — ищите *аналогии* в POSIX API (это особенно хорошо было видно на примерах чтения файлов из ядра).
- и, конечно, необходимо дотошное изучение заголовочных файлов в `ls` (/lib/modules/`uname-r`/build/include) и обширных документальных заметок в подкаталоге Documentation дерева исходных кодов вашего ядра. В этом чрезвычайно помогают онлайновые ресурсы проекта BootLin (**https://elixir.bootlin.com/linux/latest/source**);
- наконец, одну и ту же функциональность в коде можно реализовать обычно несколькими совершенно разными способами. Для реализаций ваших фантазий в пространстве ядра существует еще больше *альтернатив*, чем в пространстве пользователя. Хорошо показателен в этом смысле обсуждавшийся ранее пример открытия и чтения именованного файла. Эта задача, например, может быть реализована как минимум четырьмя различными способами (мы сейчас не обсуждаем эффективность и предпочтительность любого из них):

[5] Тот, в ком наличествуют такие наклонности, кого не отучили от них папа с мамой, и без дополнительных подсказок отыщет свой путь: «свинья грязь везде найдет».

- открытие `filp_open()` с последующим чтением с помощью `kernel_read()`;
- открытие `filp_open()` с последующим чтением с помощью `vfs_read()`;
- открытие системным вызовом `sys_open()` с последующим чтением с помощью системного вызова `sys_read()` — притом что системные вызовы осуществляются через команду `int 0x80` или ее эквиваленты;
- открытие вызовом функции-обработчика системным вызовом `sys_open()` с последующим чтением также с помощью вызова функции-обработчика системного вызовом `sys_read()` — притом что адреса функций-обработчиков находятся как неэкспортируемые символы ядра.

И даже эти предложенные варианты — далеко не всё, что можно применить в качестве альтернативных способов реализации абсолютно одной и той же функциональности в ядре.

В этом месте, наверное, уместно обратить внимание на постоянно повторяемую из одной публикации по ядру в другую мантру о том, что в коде (модулей) ядра недоступны для использования библиотеки (разделяемые, *.so) и, в частности, стандартная библиотека языка C (POSIX). Это, безусловно, так, но указанное ограничение следует понимать именно *узко технологически* — библиотеки недоступны *как формат* представления данных. Но *функциональность* кода, реализующего библиотечные вызовы API, вполне доступна — через системные же вызовы — и для кода модуля ядра, как это и показывалось здесь. Таким образом, *весь* API системных вызовов Linux также доступен — при некоторой изобретательности — и в коде модуля тоже, в тех случаях, конечно, когда эти вызовы обладают хоть каким-то смыслом в контексте ядра: вызов `sys_getpid()` можно выполнить из ядра, но в большинстве случаев возвращаемое им значение будет сложно интерпретировать, и часто это окажется просто «мусор».

> ***Примечание***
>
> Ранее я упоминал четыре кольца защиты процессоров *x*86, из которых кольцо 0 (наивысшего приоритета) используется для работы ядра, а кольцо 3 (наинизших привилегий) — для выполнения пользовательских приложений. Та же картина тиражируется в большинстве публикаций. Здесь уместно будет на этом коротко остановиться...

Такое состояние, действительно, существовало с момента появления защищенного режима, начиная как минимум с процессора 386 (рис. 7.1). Но дальше оно начинает меняться (и уточняться — потому что производители не очень спешили открывать эти подробности в техдокументации).

С появлением необходимости поддержки аппаратной виртуализации производители процессоров несколько изменили их архитектуру за счет введения дополнительных инструкций для предоставления прямого доступа к ресурсам процессора из гостевых (виртуальных) систем.

Процессор с поддержкой виртуализации может работать в двух режимах: root operation и non-root operation (технологии виртуализации отличаются: Intel VT (Intel Virtualization Technology), AMD-V, SVM (Secure Virtual Machines) — но существенной разницы в принципы это не вносит). В режиме root operation работает спе-

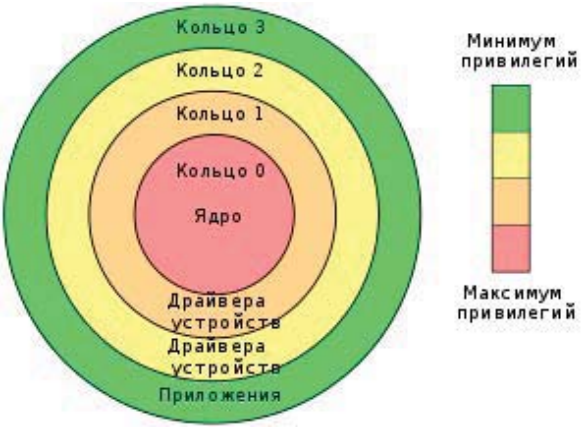


Рис. 7.1. Четыре кольца защиты процессоров *x*86

циальное программное обеспечение, являющееся «легковесной» прослойкой между гостевыми операционными системами и оборудованием, — монитор виртуальных машин (Virtual Machine Monitor, VMM), носящий также название *гипервизор* (hypervisor). Режим гипервизора и называют «кольцом –1» привилегий. Не все современные модели процессоров его поддерживают — посмотрев расширения системы команд, мы можем видеть (далее показаны процессоры Intel и AMD, у которых присутствуют расширения команд для поддержки виртуализации):

```
$ inxi -Cxxxf
CPU:
  Info: quad core model: Intel Core i7-4870HQ bits: 64 type: MT MCP
    smt: enabled arch: Haswell rev: 1 cache: L1: 256 KiB L2: 1024 KiB L3: 6 MiB
  Speed (MHz): avg: 2493 high: 2497 min/max: 800/3700 cores: 1: 2495
    2: 2477 3: 2495 4: 2495 5: 2497 6: 2495 7: 2496 8: 2497 bogomips: 39910
  Flags: abm acpi aes aperfmperf apic arat arch_perfmon avx avx2 bmi1 bmi2
    bts clflush cmov constant_tsc cpuid cpuid_fault cx16 cx8 de ds_cpl dtes64
    dtherm dts epb ept ept_ad erms est f16c flexpriority flush_l1d fma fpu
    fsgsbase fxsr ht ibpb ibrs ida invpcid invpcid_single lahf_lm lm mca mce
    md_clear mmx monitor movbe msr mtrr nonstop_tsc nopl nx pae pat pbe pcid
    pclmulqdq pdcm pdpe1gb pebs pge pln pni popcnt pse pse36 pti pts rdrand
    rdtscp rep_good sdbg sep smep smx ss ssbd sse sse2 sse4_1 sse4_2 ssse3
    stibp syscall tm tm2 tpr_shadow tsc tsc_adjust tsc_deadline_timer vme vmx
    vnmi vpid x2apic xsave xsaveopt xtopology xtpr
$ inxi -Cxxxf
CPU:
  Info: dual core model: AMD GX-212JC SOC with Radeon R2E Graphics bits: 64
    type: MCP smt: <unsupported> arch: Puma rev: 1 cache: L1: 128 KiB
    L2: 1024 KiB
  Speed (MHz): avg: 1198 min/max: N/A cores: 1: 1198 2: 1198 bogomips: 4791
  Flags: 3dnowprefetch abm acc_power aes aperfmperf apic arat avx bmi1
    bpext clflush cmov cmp_legacy constant_tsc cpb cr8_legacy cx16 cx8 de
```

```
decodeassists extapic extd_apicid f16c flushbyasid fpu fxsr fxsr_opt ht
hw_pstate ibs lahf_lm lbrv lm mca mce misalignsse mmx mmxext monitor
movbe msr mtrr nonstop_tsc nopl npt nrip_save nx osvw overflow_recov pae
pat pausefilter pclmulqdq pdpe1gb perfctr_l2 perfctr_nb pfthreshold pge
pni popcnt pse pse36 ptsc rdtscp rep_good sep skinit ssbd sse sse2 sse4_1
sse4_2 sse4a ssse3 svm svm_lock syscall topoext tsc tsc_scale vme vmmcall
wdt xsave xsaveopt
```

Следующий уровень (вниз) — это режим системного управления (System Management Mode, SMM) — режим исполнения на процессорах x86/x86-64, при котором приостанавливается исполнение другого кода (включая операционные системы и гипервизор) и запускается специальная программа, хранящаяся в SMRAM в наиболее привилегированном режиме.

Далее цитирую (из Википедии, статья «System Management Mode»):

- Технология SMM была впервые реализована в микропроцессоре Intel 386 SL. Изначально SMM работал только на специальных процессорах (SL), но в 1992 году была внедрена в 80486 и Intel Pentium. AMD реализовала технологию в Enhanced Am486 (1994). Все более современные *x*86/*x*86-64 процессоры поддерживают ее.
- Операционная система работает в защитном «Кольце 0»; однако гипервизор (в системах VT/AMD-V) является более привилегированным, и режим исполнения гипервизора условно называется «Кольцом −1». Соответственно SMM, являющийся более приоритетным, чем гипервизор, условно называют «Кольцом −2». Код, работающий в режиме SMM, получает неограниченный доступ ко всей системной памяти, включая память ядра и память гипервизора.
- SMM активируется при помощи прерываний SMI (System Management Interrupt, прерывание системного управления)...

Но и это еще не все... *Позднее* выясняется наличие существования еще одной, еще более приоритетной подсистемы («кольцо –3»), выполняющей отдельный программный код. Снова цитаты (из Википедии, статья «Intel Management Engine»):

- Intel Management Engine Interface (IMEI или Intel ME) — автономная подсистема, встроенная почти во все чипсеты процессоров Intel с 2008 года. Она состоит из проприетарной прошивки, исполняемой отдельным микропроцессором. Так как чипсет всегда подключен к источнику тока (батарейке или другому источнику питания), эта подсистема продолжает работать, даже когда компьютер отключен. Intel заявляет, что ME необходима для обеспечения максимальной производительности. Точный принцип работы по большей части не документирован, а исходный код обфусцирован с помощью кода Хаффмана, таблица для которого хранится непосредственно в аппаратуре, поэтому сама прошивка не содержит информации для своего раскодирования. Главный конкурент Intel, компания AMD, также встраивает в свои процессоры аналогичную систему — AMD Secure Technology (раньше называвшуюся Platform Security Processor), начиная с 2013 года.

- Начиная с ME 11, в основе лежит 32-битный *x*86-совместимый процессор на технологии Intel Quark с запущенной на нем операционной системой MINIX 3. Состояние ME хранится в разделе шины SPI с использованием файловой системы EFFS (Embedded Flash File System).
- Компания Positive Technologies обнаружила, что прошивка ME версии 11 использует MINIX 3.
- ME имеет свой MAC-адрес и IP-адрес для своего дополнительного интерфейса с прямым доступом к контроллеру Ethernet. Каждый пакет Ethernet-трафика переадресуется в ME даже до достижения операционной системы хоста, причем такое поведение поддерживается многими контроллерами, настраиваемыми по протоколу MCTP.

Эта информация приводится только в качестве справки, напоминания... но разработчик кода (модуля) ядра должен об этом иногда вспоминать...

- Глава 8 -
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, и многие из этих возможностей выполняются параллельно и в независимых потоках от наблюдаемого (в модуле);
- код модуля в полной мере не может быть выполнен под отладчиком и 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), когда становится возможен динамический контроль значений и состояний (диагностика), практически никогда нет возможности *изменять* значения для наблюдения за их поведением, как это практикуется в пользовательском пространстве с использованием GDB — эта особенность обусловливается не технологическими сложностями отладчиков, а уровнем последствий для операционной системы в результате таких вмешательств;
- ошибки, возникающие в коде ядра, может оказаться чрезвычайно трудно *воспроизвести*, и повторить ситуацию для анализа и наблюдения — многие редкие коллизии возникают во времени с вероятностью, скажем, в 10^{-5}...10^{-6} с, а то и гораздо реже;
- поиском ошибок ядра можно легко сломать всю систему и тем самым уничтожить и большую часть хранимых данных, которые использовались для их поиска;
- любая, даже самая незначительная ошибка в проверяемом коде *весьма часто* приводит к краху системы и необходимости перезагрузки. Если в пользовательском приложении, получив крах приложения, нам нужно только перезапустить приложение (внеся изменения), то в ядре это требует перезагрузки системы. В итоге *темп* разработки для ядра может оказаться по времени *на порядок* ниже ...если не на два.

Еще одна сложность отладки в пространстве ядра, на этот раз уже не технического свойства, состоит в том, что команда разработчиков ядра Linux крайне негативно относится вообще к идее интерактивных отладчиков для их ядра. Мотивируется это

тем, что при наличии и использовании развитых интерактивных отладчиков для ядра будет возрастать «легкость» в отношении решений, принимаемых к ядру, и это приведет к накоплению ошибок в ядре. Впрочем, в любом случае существовал и существует целый ряд проектов интерактивных отладчиков для ядра, но ни один из них не признан как «официальный», — многие такие проекты 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 и ни хотелось признать, но *основным* способом отладки модулей ядра было и остается использование вызова отладочного вывода `printk()` — это самый универсальный способ работы по отладке. Детали использования `printk()` и настройки демонов системного журнала рассматривались ранее. Тексты сообщений не должны задействовать (по крайней мере, без мотивированной на то необходимости) символы вне таблицы ASCII — в частности, нежелательно применять кирилицу в любой кодировке (хотя за последние 10 лет в вопросах языковой локализации произошли разительные изменения в лучшую сторону).

Но если не проявлять известную осторожность в использовании `printk()`, можно получить многие тысячи сообщений, созданных выполнением `printk()`, переполняющих 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

Конечно, для отладочных целей в ядре можно пытаться использовать общеизвестный Linux-отладчик GDB, но только для целей *наблюдения*. Из документов в дереве исходных кодов ядра:

> Можно пользоваться практически всеми командами программы GDB для чтения информации. Если ядро было собрано с отладочной информацией, то GDB сможет выдавать больше информации (разыменовывать указатели и выводить дампы структур данных). Нельзя изменять данные ядра. Нет возможности пошагово выполнять код ядра или останавливать его выполнение.

Как пишут, для запуска GDB используется команда (естественно, root обязателен):

```
# gdb /usr/src/linux/vmlinux /proc/kcore
...
```

Здесь первый параметр указывает *пересобранный* образ ядра (*не сжатый*, а загружаемый образ вашей системы. Находящийся, например, по имени /boot/vmlinuz — это сжатый образ), а второй параметр — это имя файла ядра, формируемое динамически:

```
$ ls -l /boot/vmlinuz
lrwxrwxrwx 1 root root 25 авг 10 10:44 /boot/vmlinuz -> vmlinuz-5.4.0-124-generic
$ sudo file /boot/vmlinuz-5.4.0-124-generic
/boot/vmlinuz-5.4.0-124-generic: Linux kernel x86 boot executable bzImage, version 5.4.0-124-
generic (buildd@lcy02-amd64-089) #140-Ubuntu SMP Thu Aug 4 02:23:37 UTC 2022, RO-rootFS,
swap_dev 0xD, Normal VGA
$ ls -l /proc/kcore
-r-------- 1 root root 140737477885952 авг 21 11:18 /proc/kcore
```

Я чисто экспериментально проверил такую возможность в типовой инсталляции дистрибутива Mint 20.3 безо всяких специальных подготовительных телодвижений:

```
$ lsb_release -a
No LSB modules are available.
Distributor ID:  Linuxmint
Description:     Linux Mint 20.3
Release:         20.3
Codename:        una
$ grep CONFIG_DEBUG_INFO /boot/config-`uname -r`
CONFIG_DEBUG_INFO=y
# CONFIG_DEBUG_INFO_REDUCED is not set
# CONFIG_DEBUG_INFO_SPLIT is not set
CONFIG_DEBUG_INFO_DWARF4=y
CONFIG_DEBUG_INFO_BTF=y
```

Запуск GDB:

```
# gdb vmlinux /proc/kcore
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
vmlinux: Нет такого файла или каталога.
[New process 1]
Core was generated by `BOOT_IMAGE=/boot/vmlinuz-5.4.0-124-generic root=UUID=cf9475ca-8800-
482b-9d09-30'.
#0  0x0000000000000000 in ?? ()
(gdb) help
```

```
List of classes of commands:

aliases -- Aliases of other commands.
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
status -- Status inquiries.
support -- Support facilities.
tracepoints -- Tracing of program execution without stopping the program.
user-defined -- User-defined commands.

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb) quit
```

Помимо GDB, создавался еще 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 продолжают активно развиваться). Только коротко упомянем такого рода инструменты, а детальное их использование оставим для энтузиастов на самостоятельную проработку:

- встроенный отладчик ядра kdb, являющийся неофициальным *патчем* к ядру от Silicon Graphics International Corp. (доступен по адресу **http://oss.sgi.com**). Для использования kdb необходимо взять патч в версии, в точности соответствующей версии отлаживаемого ядра, применить его, после чего пересобрать и переустановить ядро. В настоящее время существует только для архитектуры IA-32 (*x*86);
- патч kgdb, находящийся непосредственно в дереве исходных кодов ядра. Эта технология поддерживает удаленную отладку с другого хоста, соединенного 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 удаленную отладку по сети.

Нужно иметь в виду, что оба названных продукта kgdb имеют очень ограниченный спектр поддерживаемых процессорных платформ. Из числа тех, на которых работает Linux, реально это только *x*86 и PPC. Ряд самых интересных на сегодня платформ (ARM и др.) никак не затрагивается этими средствами.

В конечном итоге мой опыт (совершенно индивидуальный) попыток с ними работать приводит к выводу: использование интерактивных средств отладки при работе над модулями ядра — бесполезно! ...это, скорее, просто напрасно потраченное время.

Отладка в виртуальной машине

Весьма продуктивной оказывается отладка модулей в среде виртуальной машины (VM). В этом направлении у меня набрался изрядный положительный опыт, полученный с использованием таких динамично развивающихся проектов виртуальных машин, как QEMU (свободный проект, доступен по ссылке: **http://wiki.qemu.org**) и VirtualBox — также основанный на коде виртуализации из QEMU проект от Sun Microsystems (ныне от Oracle) и доступный по ссылке: **http://www.oracle.com/technetwork/server-storage/virtualbox/downloads/index.html**. Существует и независимый ресурс этого проекта: **https://www.virtualbox.org/wiki/Downloads**. Хотя, естественно, может использоваться и любой другой проект для создания виртуальных сред выполнения: XEN, Bochs и т. п. — эта область продуктов очень быстро прогрессирует.

Отладка в среде виртуальной машины (естественно, с учетом всех минусов, привносимых любым моделированием) создает целый ряд дополнительных преимуществ:

- отработка модуля ядра производится в изолированном окружении — нет риска разрушения базовой операционной системы и необходимости постоянных перезагрузок, что сильно замедляет темп разработки проекта;
- простота связи (загрузка модуля, наблюдение результатов) со средой разработки по внутренней TCP/IP виртуальной сети на основе тунельного интерфейса Linux. Для проектов сетевых модулей всегда может создаваться несколько *виртуальных* сетевых интерфейсов — при разрушении отлаживаемого интерфейса всегда сохраняется возможность доступа по резервным и восстановления работоспособности среды;
- возможность использования отладчика GDB в базовой (хостовой) системе для наблюдения «извне» за процессами, происходящими в виртуальной машине. Особенно гибко это реализовано и используется в среде QEMU;
- возможность ведения разработки для иных процессорных архитектур (ARM, PPC, MIPS) на развитой рабочей станции *x*86 с наличием обширного инструментария (эта возможность реализуется только в QEMU, VirtualBox поддерживает лишь *x*86-архитектуру);
- в единой среде виртуализации может быть создано много виртуальных машин с разными инсталлированными версиями (сигнатурами) ядра. Это позволяет проводить тестирование и обкатку промышленного проекта (накануне сдачи) одновременно для широкого спектра применений;

Примечание

Интересно отметить, что на хостовой машине с объемом RAM, скажем, даже 4 Гбайт, одновременно могут *выполняться*, например, 10–12 VM, *каждой* из которых отведено по 1 Гбайт RAM, — при этом хостовая машина все еще будет достаточно активно реагировать на интерфейс работы с пользователем.

- в среде виртуализации может быть установлено много различных дистрибутивов Linux со всеми свойственными им различиями. Помимо этого, там же могут быть установлены VM и отличающихся операционных систем: не POSIX операционные системы Windows или POSIX, но не Linux, микроядерная операционная система QNX. Это позволяет вести отработку платформенно независимых компонентов проекта (рис. 8.1).

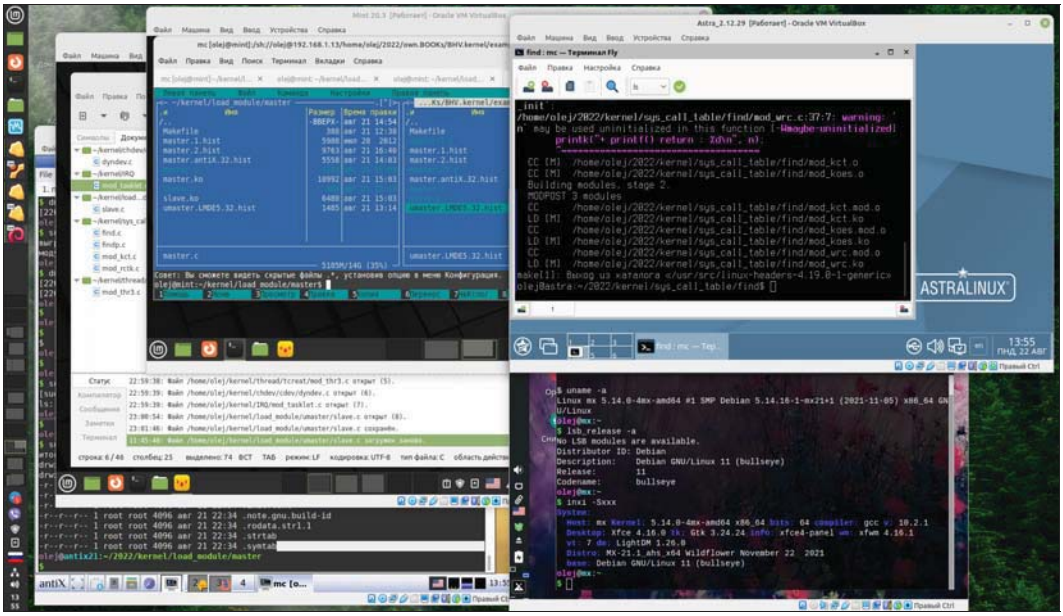


Рис. 8.1. Работают одновременно пять виртуальных машин с совершенно различающимися дистрибутивами, разрядностями, версиями ядра…

Хотелось бы специально обратить ваше внимание на предпоследний пункт приведенного списка, не совсем очевидный. Вспомним, что модуль, скомпилированный в одном ядре, неработоспособен в отличающемся ядре (даже просто *по написанию* сигнатуры, имени ядра в команде `uname -r`). А из-за изменчивости API ядра, о которой уже много сказано, ваш модуль может вообще даже не компилироваться в следующем по номеру ядре (изменение состава функций API, изменения их прототипов, типов параметров и многое другое). Но вот как раз подготовленный заранее тестовый набор виртуальных машин для последовательного набора ядер позволяет протестировать и отработать разрабатываемый модуль и подготовить наиболее гибкую его промышленную поставку (за счет, например, использования препроцессорных `#defined` относительно версий ядер в коде модуля, что и показывалось в некоторых приводившихся ранее примерах):

```
$ ps -A | grep Virt
   5820 ?        00:00:50 VirtualBox
   6350 ?        00:11:04 VirtualBoxVM
   6442 ?        00:04:41 VirtualBoxVM
   6527 ?        00:42:24 VirtualBoxVM
   6618 ?        00:15:44 VirtualBoxVM
   6664 ?        00:04:54 VirtualBoxVM
```

Из упомянутых двух близких систем виртуализации у каждой в отладочной технологии своя роль: QEMU является более гибким и универсальным инструментом, допускающим отработку для разной архитектуры (ARM, MIPS, PPC, ...), но

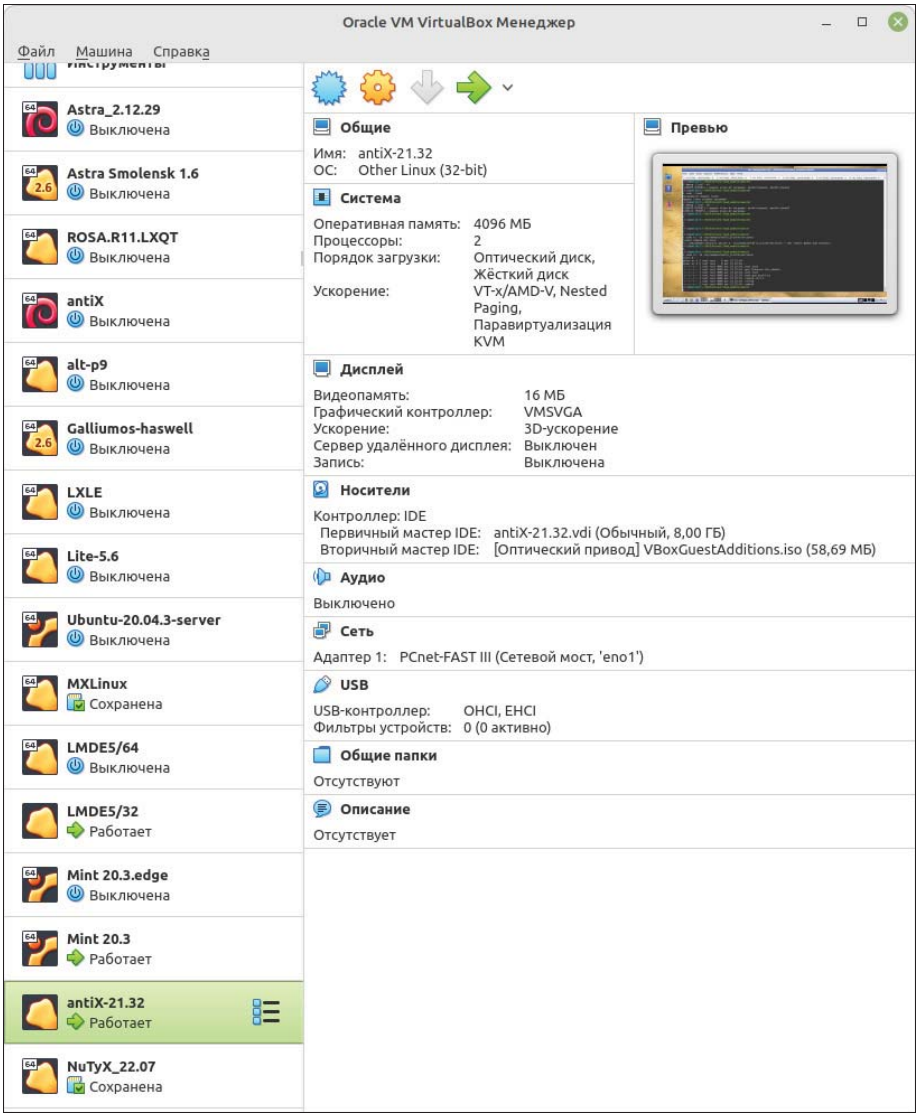


Рис. 8.2. Отладочный стенд под VirtualBox для перекрестных проверок совместимости модулей ядра

VirtualBox имеет более дружественные инструменты конфигурирования и управления виртуальными машинами. О технике отладки в виртуальной среде, особенно на кроссовых платформах, можно и должно сказать очень много, но это уже предмет отдельного большого разговора.

Следует напомнить, что в силу *сетевой природы* графического протокола X11 в UNIX любое графическое приложение Linux может быть запущено на удаленном сетевом хосте с отображением GUI приложения на локальном хосте. Не является исключением и VirtualBox, что открывает возможности для групповой работы над проектом в единой среде удаленной виртуальной машины — это создает существенно новые возможности. Добиться удаленного запуска X11 приложения можно (это не единственный способ), например, посредством тунелирования X11 протокола над протоколом SSH (опция `-X`, или еще более: `-Y` — без шифрования сетевого потока). Так был запущен и сеанс, показанный на рис. 8.2:

```
$ ssh -Y olej@192.168.1.13
olej@192.168.1.13's password:
Last login: Wed Aug 17 13:06:13 2022 from 192.168.1.241
$ VirtualBox &
[1] 38373
```

> ***Примечание***
>
> Вопреки интуитивным ожиданиям начальная отладка проектов модулей в виртуальной машине VirtualBox возможна и для модулей, работающих с периферийными устройствами (USB и PCI), аппаратными ресурсами, реакцией на прерывания IRQ и т. д. ...естественно в «виртуальном отображении» (параметров) этого оборудования.

Отдельные отладочные приемы и трюки

Многолетний практический опыт работы с модулями ядра привел меня к убеждению, что реальным отладочным инструментом в планомерной работе по отладке является только отладочная печать с помощью `printk()`. Но вместо развитых отладочных инструментов на помощь здесь приходят отработаные практикой приемы использования отладочной печати. Здесь мы рассмотрим некоторые мелкие приемы, применявшиеся в процессе отладки, которые сложились и показали свою продуктивность в процессе работы над реальными проектами в области модулей ядра.

Модуль, исполняемый как разовая задача

Один из продуктивных трюков, который уже много раз применялся по ходу всего рассмотрения ранее, — это сознательное написание модуля, возвращающего ненулевое (отрицательное) значение из инициализирующей функции модуля, который вовсе и «не собирается» загружаться. Такой модуль выполняется однократно, подобно пользовательскому процессу (`main()`), но отличается тем, что делает он это в супервизорном режиме (с полными привилегиями) и в адресном пространстве ядра. С таким модулем мы можем видеть *диагностику* состояния разнообразных

областей данных, но, более того, можем *изменять* любые данные или текущие конфигурационные параметры ядра. Причем такие изменения — *перманентные*, они остаются в таком виде до следующей перезагрузки системы.

В качестве примера приводится элементарно простейший модуль, дающий возможность посмотреть адрес любого экспортируемого символа ядра (все примеры этого раздела содержатся в папке simple-debug сопровождающего книгу файлового архива):

md.c

```
#include <linux/module.h>
MODULE_LICENSE("GPL");

static int __init do_init(void) {
   void* Addr = (void*)vprintk;
   printk("? интересующий нас адрес: %px\n", Addr);
   return -1;
}

module_init(do_init);
```

Такой модуль в принципе не может загрузиться в ядро, т. к. он возвращает –1 (или, точнее: не 0). В этой связи у модуля даже нет процедуры завершения (она ему не нужна). Но его код может быть однократно выполнен при запуске:

```
$ sudo insmod md.ko
insmod: ERROR: could not insert module md.ko: Operation not permitted
$ dmesg | grep ? -A0
[12210.989138] ? интересующий нас адрес: ffffffff9f704e30
```

Этот же модуль (предварительно перекомпилированный, естественно) мы можем использовать и в любой другой архитектуре... и процессора, и дистрибутива, и ядра, и версии — например, в ARM:

```
$ sudo inxi -MSxxx
System:    Host: orangepione Kernel: 5.15.48-sunxi armv7l bits: 32 compiler: N/A
           Console: tty 1 dm: LightDM 1.26.0 Distro: Armbian GNU/Linux 10 (buster)
Machine:   Type: ARM Device System: Xunlong Orange Pi One details: Allwinner sun8i Family
           rev: N/A serial: 02c000815fd5e717
$ ls -l md.ko
-rw-r--r-- 1 olej olej 3324 авг 22 14:48 md.ko
$ sudo grep ' T ' /proc/kallsyms | grep ' vprintk'$
801865e4 T vprintk
$ sudo insmod md.ko
insmod: ERROR: could not insert module md.ko: Operation not permitted
$ dmesg | grep ? -A0
[881089.811458] ? интересующий нас адрес: c0166a81
```

Или в совершенно другом ARM:

```
$ sudo inxi -MSxxx
System:
  Host: raspberrypi Kernel: 5.15.32-v7+ armv7l bits: 32 compiler: gcc v: 10.2.1
  Console: tty 1 DM: LightDM 1.26.0 Distro: Raspbian GNU/Linux 11 (bullseye)
Machine:
  Type: ARM Device System: Raspberry Pi 2 Model B Rev 1.1 details: BCM2835 rev: a21041
  serial: 00000000f57e2ca8
$ ls -l md.ko
-rw-r--r-- 1 olej olej 3472 авг 22 14:42 md.ko
$ sudo grep ' T ' /proc/kallsyms | grep ' vprintk'$
801865e4 T vprintk
$ sudo insmod md.ko
insmod: ERROR: could not insert module md.ko: Operation not permitted
$ dmesg | grep ? -A0
[880827.647054] ? интересующий нас адрес: 801865e4
```

Обратите внимание на разницу в размерах собранного одного и того же модуля в, казалось бы, даже очень близких архитектурах!

В рассмотренном качестве подобный модуль (который не загрузится, но и не навредит) становится интересным средством отладки, особенно на начальных этапах отработки, когда можно проверить все инициализированные значения модуля и используемые им экспортируемые символы ядра (учитывая, что на начальных этапах отработки ошибки в этих значениях заканчиваются крахом системы).

> ***Пояснение***
>
> Вот здесь, говоря об отладке, уместно остановиться на вопросе, который часто задавали читатели предварительной версии рукописи книги: зачем на предыдущих сотнях страниц я предваряю строку вывода `printk()` всякими неуместными и «непотребными» символами (префиксами) вида `?`, `!`, `+`, `++`... ? Дело в том, что в некоторых дистрибутивах (далеко не во всех) ядро сыплет в системный лог много разнообразных диагностических сообщений (`audit` и др.). При этом вы просто можете потерять в потоке вывода сообщения своего модуля.
>
> Префиксы же позволяют с помощью `grep` фильтровать только нужные сообщения ядра (а опция `-A`, напомню, определяет, сколько *дополнительных* строк выводить *после* найденной строки):
>
> ```
> $ dmesg | grep ? -A2 -B2
> [880807.435886] hwmon hwmon1: Undervoltage detected!
> [880813.675740] hwmon hwmon1: Voltage normalised
> [880827.647054] ? интересующий нас адрес: 801865e4
> [880828.235703] hwmon hwmon1: Undervoltage detected!
> [880834.475684] hwmon hwmon1: Voltage normalised
> ```

Тестирующий модуль

Популярным (модным) способом стала организация прямо в процессе разработки (или даже сразу от его начала) модульного тестирования (юнит-тестирования, unit testing) — непрерывного (циклического) повторения последовательности тестовых

операций над каждым добавляемым в разработку компонентом. Но при организации модульного тестирования в области ядра разработчик может с недоумением столкнуться с тем, как оформлять тесты, — ведь создаваемый код теста не может быть скомпилирован для работы в пользовательском режиме. Но в этом случае в коде проектируемого модуля (umd1.c) могут быть созданы *экспортируемые* точки входа вида `test_01()`, `test_02()`, ... `test_N()`, а для последовательного вызова тестовых входов создан отдельный *тестирующий* модуль (umt1.c), основанный на показанном ранее трюке (разовое исполнение), весь код которого умещается в единственную функцию инициализации... Оба модуля используют (для связи) только общий заголовочный файл (туда же включены общие определения для двух модулей, чтобы не раздувать их объем):

umd1.h

```
#include <linux/module.h>
MODULE_LICENSE("GPL");
static int __init init(void);
module_init(init);

extern char* test_01(void);
extern char* test_02(void);
extern char* test_03(void);

typedef char* (*utest)(void);
utest tests[] = { test_01, test_02, test_03 };
```

Сам тестируемый (развивающийся) модуль, в котором тестируются некоторые возможности:

umd1.c

```
#include "umd1.h"

inline char* mesg(const char* what) {
   static char res[120];
   sprintf(res, "this string returned from %s", what);
   return res;
}

char* test_01(void) {
   // ... здесь код теста 1
   return mesg(__FUNCTION__);
}
EXPORT_SYMBOL(test_01);

char* test_02(void) {
   // ... здесь код теста 2
```

```
   return mesg(__FUNCTION__);
}
EXPORT_SYMBOL(test_02);

char* test_03(void) {
   // ... здесь код теста 3
   return mesg(__FUNCTION__);
}
EXPORT_SYMBOL(test_03);

static int __init init(void) {
   return 0;
}

static void __exit cleanup(void) {}
module_exit(cleanup);
```

А вот и весь код тестирующего модуля, который запускает последовательность модульных тестов (пишем, как и было сказано ранее, для однократного выполнения):

umt1.c

```
#include "umd1.h"

static int __init init(void) {
   int i;
   for(i = 0; i < sizeof(tests) / sizeof(tests[0]); i++)
      printk("%s\n", tests[i]());
   return -1;
}
```

Теперь мы можем произвольно *наращивать число* модульных тестов, минимально внося изменения, чисто формальные, только в заголовочный файл umd1.h. И вот как выглядит выполнение последовательности тестов проектируемого модуля:

```
$ sudo insmod umd1.ko
$ sudo insmod umt1.ko
insmod: ERROR: could not insert module umt1.ko: Operation not permitted
$ dmesg | tail -n3
[26164.146722] this string returned from test_01
[26164.146724] this string returned from test_02
[26164.146725] this string returned from test_03
$ sudo rmmod umd1
```

Если угодно, вывод результатов тестов — то, что делает dmesg, — можете вывести даже на терминал, запускающий insmod, — мы это проделывали несколькими страницами ранее.

Интерфейсы пространства пользователя к модулю

Для контроля значений ключевых переменных (и даже их изменений) внутри модуля их можно отобразить в псевдофайловые системы /proc, а еще лучше в /sys. Это часто делается, например, для счетчика обработанных в драйвере прерываний, как это показано в следующем примере (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 ранее, так что все механизмы нам знакомы. Обработка ошибок при установке модуля здесь практически отсутствует, чтобы не загромождать текст.

> ***Примечание***
>
> Модули этого и следующего разделов — под sysfs — разрабатывались и проверялись под ядро 3.13.10. Позже определение макросов `CLASS_ATTR()` было в </linux/device.h> изменено, но код уже далее в этой «гонке за ядром» не сопровождался, потому что все принципы он достаточно поясняет, а какой-то конкретной ценности не представляет.

Для проверки того, как это работает, загружаем модуль для контроля выбранной 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 счетчика сработавших прерываний. Изменим начальное значение этого счетчика, 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 показано ранее, 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 при этом сохраняется возможность очень просто разрушить нормальную работу изделия. Однако переписывать код модуля перед его сдачей — это тоже мало приемлемый вариант, т. к. такой редактурой можно внести в его код существенные ошибки. 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 их) при своей загрузке...
- и создает для них интерфейсы к диагностическим и управляющим переменным;
- после завершения отладки отладочный модуль просто изымается из проекта.

Чтобы увидеть в деталях, о чем речь, трансформируем в эту схему пример, описанный в предыдущем разделе. Причем сделаем это безо 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 в /sys:

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 из 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 уже исчезли:

```
$ lsmod | head -n3
Module                  Size  Used by
mdsys2                   844  0
lp                      6794  0
$ cat /sys/class/mds-class/mds
cat: /sys/class/mds-class/mds: Нет такого файла или каталога
$ sudo rmmod mdsys2
```

Пишите в файлы протоколов...

У вас всегда остается возможность писать отладочные сообщения из модуля ядра в собственный *файл протокола* выполнения, который позже будет доступен для детального анализа. Имеется в виду свой *собственный* файл *данных*, который создает и пишет модуль. При этом вы ничем не ограничены в степени детализации сообщений, направляемых в файл протокола. С самой техникой такой записи в файл мы познакомились ранее, при рассмотрении работы модуля с файлами данных (см. папку file в сопровождающем книгу файловом архиве).

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 самым малозаметные (или поздно обнаруживаемые) аномалии в поведении системы. Особенно часто это наблюдается, например, при отработке интерфейсов драйвера в файловую систему /proc.

Побочные эффекты от накопленных ошибок в ядре системы могут доходить до того состояния, что при дальнейших улучшениях отрабатываемого кода, даже компилятор GCC станет сообщать вам о каких-то загадочных внутренних ошибках компилятора... Это уже явная народная примета того, что... пришла пора перезагружаться!

И чтобы избежать десятков часов *бездарно* потерянного времени, при работе над модулями перезагружайте время от времени свою Linux, даже если вам кажется, что она 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 жесткое тестирование драйвера, проверьте его реакцию по чтению и записи 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, отработку 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 от того, 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 операций записи, но в значительно меньшей степени.

Заключение

Отпускай хлеб твой по водам,
потому что по прошествии многих дней опять найдёшь его.
Екклисиаст, XI:1

Есть еще множество механизмов, API и трюков, которые используются в программировании ядра. Их просто нет возможности описать в любом издании обозримого объема — для сравнения обратитесь к POSIX API пользовательского пространства, которое описывают тысячи и тысячи страниц публикаций... А API ядра должно иметь и имеет аналоги практически всех механизмов, предоставляемых в пространстве пользователя.

Существует некоторое предубеждение, что программирование в ядре и, в частности, модулей ядра требует особого аскетизма в выборе используемых возможностей и вообще весьма ограничено в том, что вам при этом доступно. Целью моих примеров было показать: при программировании в технике модулей ядра вам доступны *все возможности*, о которых вы слышали из POSIX... плюс еще изрядное количество сверх того. Некоторое ограничение на этом пути составляет отсутствие внятных описаний API ядра, но это ограничение преодолевается дотошным изучением («верить никому нельзя») и изобретательным экспериментированием.

Основную (а временами и единственную) помощь в поиске адекватных нашим намерениям API ядра дает рассмотрение открытых исходных кодов ядра и главным образом *заголовочных файлов* определений кода ядра (с чего и нужно начинать рассмотрение).

- Приложение 1 -
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, причем для довольно изощренных случаев. Детальное описание `make` доступно в [22]. Здесь же приведем только самую краткую справку (главным образом для напоминания об умалчиваемых значениях переменных `make`).

Утилита `make` существует в разных ОС, причем из-за особенностей выполнения, наряду с «родной» реализацией, во многих ОС присутствует GNU-реализация `gmake`, и поведение этих реализаций может достаточно существенно различаться, поэтому совсем не лишним бывает проверить, с чем мы имеем дело:

```
$ make --version
GNU Make 4.2.1
Эта программа собрана для x86_64-pc-linux-gnu
Copyright (C) 1988-2016 Free Software Foundation, Inc.
Лицензия GPLv3+: GNU GPL версии 3 или новее <http://gnu.org/licenses/gpl.html>
...
```

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. Ее можно использовать и для решения любых задач, где одни файлы должны автоматически обновляться при изменении других файлов.

Многократно выполняемая сборка приложений проекта с учетом зависимостей и обновлений делается утилитой `make`, использующей оформленный сценарий сборки. По умолчанию имя файла сценария сборки — Makefile. Утилита `make` обеспечивает полную сборку одной указанной *цели* в сценарии сборки, например:

```
$ make
$ make clean
```

Если цель не указывается, то выполняется *первая последовательная* цель в файле сценария (почему-то существует суеверие, что собирается цель с именем `all` — просто цель `all` ставится в файле часто выше всех остальных, вот она по умолчанию

и собирается). Может использоваться и любой другой сценарный файл сборки, тогда он указывается так:

```
$ make -f Makefile.my
```

Сценарий Makefile состоит из синтаксических конструкций всего двух типов: целей и макроопределений. Описание цели состоит из трех частей: а) имени цели; б) списка зависимостей и в) списка команд интерпретатора shell, требуемых для построения цели. Имя цели — непустой список имен файлов, которые предполагается создать. Список зависимостей — список имен 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 (еще одна, следующая команда) или # (комментарий), считается завершением текущей цели и началом новой.

Утилита `make` имеет множество умалчиваемых значений (переменных, суффиксов, ...), интересных и значащих, важнейшими из которых являются правила обработки суффиксов, а также определения внутренних переменных окружения. Эти данные называются *базой данных* `make` и могут быть получены выполнением самой команды `make` с опцией: `--print-data-base` (или ее короткой формой: `-p`). Объем вывода очень велик, поэтому смотрим его лучше перенаправлением в файл (да и вообще *такой* файл полезно постоянно хранить под рукой в качестве справочника):

```
$ make -f/dev/null --print-data-base > make.data-base
make: *** Нет целей.  Останов.
$ echo $?
2
```

Вывод команды, при этом выглядящий как ошибка, не должен вас смущать, не принимайте это во внимание:

```
$ ls -l make.data-base
-rw-rw-r-- 1 olej olej 48290 авг 24 21:13 make.data-base
$ cat make.data-base
# GNU Make 4.2.1
# Эта программа собрана для x86_64-pc-linux-gnu
...
# База данных Make, напечатана Wed Aug 24 21:17:56 2022
...
# Переменные
COMPILE.cpp = $(COMPILE.cc)
...
# по умолчанию
CC = cc
...
CPP = $(CC) -E
LINK.cc = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
...
```

```
COMPILE.cc = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
CXX = g++
...
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
...
LINK.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
...
.INCLUDE_DIRS = /usr/include /usr/local/include /usr/include
# Неявные правила
...
%.c:
%: %.c
#  способ, который следует применить (встроенные):
        $(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
...
%.cc:

%: %.cc
#  способ, который следует применить (встроенные):
        $(LINK.cc) $^ $(LOADLIBES) $(LDLIBS) -o $@
...
%.cpp:
%: %.cpp
#  способ, который следует применить (встроенные):
        $(LINK.cpp) $^ $(LOADLIBES) $(LDLIBS) -o $@
...
```

Здесь показаны только некоторые макроопределения, но они интуитивно понятны и в большинстве своем знакомы (тем более что нам предстоит не писать некоторые правила в этом синтаксисе, а только *заимствовать* отсюда переменные периода сборки и правила). Обратите внимание, что это не набор определений, жестко вшитый в реализацию `make`, — он преломлен через конкретные настройки (язык, пути... и пр.) вашей системы!

Все эти значения (переменных: `CC`, `LD`, `AR`, ...) могут использоваться файлом сценария как неявные определения со значениями по умолчанию. Кроме этого, вы можете определить и свои правила обработки по умолчанию для указанных вами суффиксов (расширений файловых имен), как это показано в приведенном примере для исходных файлов кода на языке C: `*.c`.

- Приложение 2 -
Тесты распределителя памяти

Принципы динамического выделения памяти детально обсуждались в этой книге ранее. Но в литературе и обсуждениях фигурируют самые разнообразные и противоречивые цифры и рекомендации по использованию (или неиспользованию) механизмов `kmalloc()`, `vmalloc()`, `__get_free_pages()`. Выполним их некоторые грубые оценки на различных компьютерах, с различными объемами реальной RAM и с установленными Linux различных версий ядра.

Начнем с размера *страниц*, которыми оперирует с памятью ядро Linux (мы не станем углубляться в механизм физического управления страницами через таблицы страниц TLB, число уровней TLB и подобные вопросы — они сложны и не существенны для разработки модулей ядра). Ядро манипулирует страницами, размер которых определяется константой *периода компиляции* ядра `PAGESIZE`. Посмотреть это значение можно разными способами (ввиду его предопределяющего смысла!):

```
$ getconf PAGESIZE
4096
```

Или его можно динамически проследить в ходе выполнения (runtime) ядра (эти значения нужно снимать практически одновременно, поскольку они постоянно меняются):

```
$ grep nr_mapped /proc/vmstat
nr_mapped 205437
$ grep ^Mapped /proc/meminfo
Mapped:           821740 kB
```

Рассчитайте: 821740 × 1024 : 205437 = 4095,960124028 — получаете то же значение (минимальные расхождения связаны с быстрой изменчивостью этих значений).

Наконец, мы можем сделать (и оно того стоит) «игрушечный» модуль для проверки размера используемой страницы памяти:

page.c

```
#include <linux/module.h>
static int __init init(void) {

#ifdef __i386__
   printk("разрядность системы 32\n");
```

```
#else
   printk("разрядность системы 64\n");
#endif
   printk("размер страницы RAM = %lu [%luKb]\n", PAGE_SIZE, PAGE_SIZE / 1024);
   return -1;
}

MODULE_LICENSE("GPL");
module_init(init);
```

```
$ sudo insmod page.ko
insmod: ERROR: could not insert module page.ko: Operation not permitted
$ dmesg | tail -n2
[ 7966.807993] разрядность системы 64
[ 7966.807995] размер страницы RAM = 4096 [4Kb]
```

В архитектуре *x*86_64 в принципе возможно (так утверждают) использование страниц размером 4 килобайта (4096 байтов), 2 мегабайта и (в некоторых AMD64) 1 гигабайт, но в 32-битной архитектуре *x*86 используются страницы только по 4 килобайта. И я *никогда*, в десятках испытанных дистрибутивов, не встречал в архитектуре *x*86, *x*86_64 и в 32-битной архитектуре ARM значения, отличающегося от 4096.

На сегодня определение этой фундаментальной для системы константы `PAGE_SIZE` (возможно, завтра она куда-то перекочует) вы найдете в заголовочном файле: /lib/modules/`uname -r`/build/include/asm-generic/page.h:

```
...
/* PAGE_SHIFT determines the page size */
#define PAGE_SHIFT      12
#ifdef __ASSEMBLY__
#define PAGE_SIZE       (1 << PAGE_SHIFT)
#else
#define PAGE_SIZE       (1UL << PAGE_SHIFT)
#endif
```

Для тестирования того, как работает механизм выделения памяти, подготовлены два теста (см. папку memory/mtest в сопровождающем книгу файловом архиве). Первый из них запрашивает выделение (тестирует и тут же освобождает) блока, кратного *целому числу* страниц: `2^order`, причем значение `order` инкрементируется (размер блока *удваивается*) до тех пор, пока очередной блок уже не может быть выделен (завершится ошибкой):

memmax.c

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

```
// выделение памяти: 0 - kmalloc(), 1 - __get_free_pages(), 2 - vmalloc()
static int mode = 0;
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 -EINVAL;
   }
   for(size = PAGE_SIZE, order = 0; ; order++, size *= 2) {
      char msg[200];
      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 | __GFP_NOWARN);
            break;
         case 1:
            kbuf = (char *)__get_free_pages(GFP_KERNEL | __GFP_NOWARN, 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");
MODULE_DESCRIPTION("memory allocation size test");
MODULE_LICENSE("GPL v2");
```

Тестирование проделываем для трех принципиальных механизмов (параметр `mode` загрузки модуля равный, соответственно: `0`, `1`, `2`) распределения памяти в ядре (поднимитесь по тексту и освежите в памяти): `kmalloc`, `__get_free_pages`, `vmalloc`.

Все это можно и нужно (очень интересно) проследить по нескольким экземплярам компьютеров — с разной *архитектурой* процессоров, разными *объемами* установленной памяти, *разными* Linux (как по разрядности, так и по дистрибутиву и по версиям ядра). Невзирая на объем, приведу такие результаты полностью (оно того стоит)...

> ***Примечание***
>
> Обратите внимание: тест показывает не максимально возможный размер блока, который тот или иной механизм выделения памяти способен разместить (и такой тест тоже несложно соорудить из показанного), а грубо оценивает блок, который уже нельзя разместить.

Начнем с самой малой архитектуры — одноплатный ARM Orange Pi One, объем памяти на плате всего 512 Мбайт:

```
$ inxi -Mmxxx
Machine:   Type: ARM Device System: Xunlong Orange Pi One details: Allwinner sun8i Family
           rev: N/A serial: 02c000815fd5e717
Memory:    RAM: total: 491.7 MiB used: 189.8 MiB (38.6%)
           RAM Report: missing: Required program dmidecode not available
$ cat /proc/meminfo | grep MemTotal
MemTotal:         503484 kB
$ uname -a
Linux orangepione 5.15.48-sunxi #22.05.3 SMP Wed Jun 22 07:35:10 UTC 2022 armv7l GNU/Linux
$ free
               total        used        free      shared  buff/cache   available
Mem:          503484      212508       65684       11108      225292      268388
Swap:        2348888       97280     2251608
$ getconf PAGESIZE
4096
$ sudo insmod memmax.ko mode=0
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | grep order | tail -n11
[980839.955854] order= 0, pages=     1, size=     4096 - kmalloc OK
[980839.955885] order= 1, pages=     2, size=     8192 - kmalloc OK
[980839.955907] order= 2, pages=     4, size=    16384 - kmalloc OK
[980839.955936] order= 3, pages=     8, size=    32768 - kmalloc OK
[980839.955973] order= 4, pages=    16, size=    65536 - kmalloc OK
[980839.956037] order= 5, pages=    32, size=   131072 - kmalloc OK
[980839.956148] order= 6, pages=    64, size=   262144 - kmalloc OK
[980839.956354] order= 7, pages=   128, size=   524288 - kmalloc OK
```

```
[980839.956755] order= 8, pages=   256, size=  1048576 - kmalloc OK
[980839.957543] order= 9, pages=   512, size=  2097152 - kmalloc OK
[980839.970235] order=10, pages=  1024, size=  4194304 - kmalloc failed
$ sudo insmod memmax.ko mode=1
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | grep order | tail -n12
[968198.179911] order= 0, pages=     1, size=     4096 - __get_free_pages OK
[968198.179946] order= 1, pages=     2, size=     8192 - __get_free_pages OK
[968198.179964] order= 2, pages=     4, size=    16384 - __get_free_pages OK
[968198.179992] order= 3, pages=     8, size=    32768 - __get_free_pages OK
[968198.180029] order= 4, pages=    16, size=    65536 - __get_free_pages OK
[968198.180093] order= 5, pages=    32, size=   131072 - __get_free_pages OK
[968198.180204] order= 6, pages=    64, size=   262144 - __get_free_pages OK
[968198.180409] order= 7, pages=   128, size=   524288 - __get_free_pages OK
[968198.180805] order= 8, pages=   256, size=  1048576 - __get_free_pages OK
[968198.181584] order= 9, pages=   512, size=  2097152 - __get_free_pages OK
[968198.183224] order=10, pages=  1024, size=  4194304 - __get_free_pages OK
[968198.183281] order=11, pages=  2048, size=  8388608 - __get_free_pages failed
$ sudo insmod memmax.ko mode=2
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | grep order | tail -n18
[968268.886565] order= 0, pages=     1, size=     4096 - vmalloc OK
[968268.886605] order= 1, pages=     2, size=     8192 - vmalloc OK
[968268.886634] order= 2, pages=     4, size=    16384 - vmalloc OK
[968268.886672] order= 3, pages=     8, size=    32768 - vmalloc OK
[968268.886730] order= 4, pages=    16, size=    65536 - vmalloc OK
[968268.886837] order= 5, pages=    32, size=   131072 - vmalloc OK
[968268.887049] order= 6, pages=    64, size=   262144 - vmalloc OK
[968268.887411] order= 7, pages=   128, size=   524288 - vmalloc OK
[968268.888095] order= 8, pages=   256, size=  1048576 - vmalloc OK
[968268.889684] order= 9, pages=   512, size=  2097152 - vmalloc OK
[968268.892555] order=10, pages=  1024, size=  4194304 - vmalloc OK
[968268.898242] order=11, pages=  2048, size=  8388608 - vmalloc OK
[968268.909663] order=12, pages=  4096, size= 16777216 - vmalloc OK
[968268.933189] order=13, pages=  8192, size= 33554432 - vmalloc OK
[968269.047460] order=14, pages= 16384, size= 67108864 - vmalloc OK
[968269.527135] order=15, pages= 32768, size=134217728 - vmalloc OK
[968271.135626] order=16, pages= 65536, size=268435456 - vmalloc OK
[968271.204675] order=17, pages=131072, size=536870912 - vmalloc failed
```

Что мы здесь видим? Что `kmalloc` выделяет чуть больше 512 страниц единым блоком (2 Мбайт), `__get_free_pages` — порядка 1000–1500 страниц (4 Мбайт), и только `vmalloc` легко выделяет 250 Мбайт, и это притом что весь объем памяти вместе с занимаемым самой системой этого экземпляра составляет 512 Мбайт — в высшей степени эффективно. Напомню, что в последнем случае выделен непрерывный блок *логической* памяти, который не отображается на *непрерывный* блок в физической памяти

(виртуально отображается «кусками» — блок памяти собран из таких фрагментов), в отличие от первых двух способов.

Смотрим далее конфигурацию покрупнее — Raspberry Pi с объемом RAM на плате 1 Гбайт:

```
$ inxi -MCmxxx
Machine:   Type: ARM Device System: Raspberry Pi 2 Model B Rev 1.1 details: BCM2835 rev: a21041
        serial: 00000000f57e2ca8
Memory: RAM: total: 999.1 MiB used: 235.6 MiB (23.6%) gpu: 76 MiB
        RAM Report: unknown-error: Unknown dmidecode error. Unable to generate data.
CPU:    Info: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32 type: MCP arch: v7l rev: 5
        features: Use -f option to see features bogomips: 0
        Speed: 700 MHz min/max: 600/1000 MHz Core speeds (MHz): 1: 700 2: 700 3: 700 4: 700
$ cat /proc/meminfo | grep MemTotal
MemTotal:      945300 kB
$ uname -a
Linux raspberrypi 5.15.32-v7+ #1538 SMP Thu Mar 31 19:38:48 BST 2022 armv7l GNU/Linux
$ sudo insmod memmax.ko mode=0
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | grep order | tail -n12
[976262.303285] order= 0, pages=     1, size=     4096 - kmalloc OK
[976262.303320] order= 1, pages=     2, size=     8192 - kmalloc OK
[976262.303342] order= 2, pages=     4, size=    16384 - kmalloc OK
[976262.303367] order= 3, pages=     8, size=    32768 - kmalloc OK
[976262.303391] order= 4, pages=    16, size=    65536 - kmalloc OK
[976262.303417] order= 5, pages=    32, size=   131072 - kmalloc OK
[976262.303440] order= 6, pages=    64, size=   262144 - kmalloc OK
[976262.303578] order= 7, pages=   128, size=   524288 - kmalloc OK
[976262.303629] order= 8, pages=   256, size=  1048576 - kmalloc OK
[976262.303682] order= 9, pages=   512, size=  2097152 - kmalloc OK
[976262.303763] order=10, pages=  1024, size=  4194304 - kmalloc OK
[976262.303808] order=11, pages=  2048, size=  8388608 - kmalloc failed
$ sudo insmod memmax.ko mode=1
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | grep order | tail -n12
[965306.210599] order= 0, pages=     1, size=     4096 - __get_free_pages OK
[965306.210656] order= 1, pages=     2, size=     8192 - __get_free_pages OK
[965306.210686] order= 2, pages=     4, size=    16384 - __get_free_pages OK
[965306.210715] order= 3, pages=     8, size=    32768 - __get_free_pages OK
[965306.210749] order= 4, pages=    16, size=    65536 - __get_free_pages OK
[965306.210850] order= 5, pages=    32, size=   131072 - __get_free_pages OK
[965306.210885] order= 6, pages=    64, size=   262144 - __get_free_pages OK
[965306.210920] order= 7, pages=   128, size=   524288 - __get_free_pages OK
[965306.210968] order= 8, pages=   256, size=  1048576 - __get_free_pages OK
[965306.211082] order= 9, pages=   512, size=  2097152 - __get_free_pages OK
[965306.211173] order=10, pages=  1024, size=  4194304 - __get_free_pages OK
[965306.211237] order=11, pages=  2048, size=  8388608 - __get_free_pages failed
```

```
$ sudo insmod memmax.ko mode=2
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | grep order | tail -n19
[965381.550356] order= 0, pages=     1, size=     4096 - vmalloc OK
[965381.550433] order= 1, pages=     2, size=     8192 - vmalloc OK
[965381.550488] order= 2, pages=     4, size=    16384 - vmalloc OK
[965381.550555] order= 3, pages=     8, size=    32768 - vmalloc OK
[965381.550653] order= 4, pages=    16, size=    65536 - vmalloc OK
[965381.550842] order= 5, pages=    32, size=   131072 - vmalloc OK
[965381.551133] order= 6, pages=    64, size=   262144 - vmalloc OK
[965381.551688] order= 7, pages=   128, size=   524288 - vmalloc OK
[965381.552791] order= 8, pages=   256, size=  1048576 - vmalloc OK
[965381.555206] order= 9, pages=   512, size=  2097152 - vmalloc OK
[965381.559872] order=10, pages=  1024, size=  4194304 - vmalloc OK
[965381.569045] order=11, pages=  2048, size=  8388608 - vmalloc OK
[965381.591305] order=12, pages=  4096, size= 16777216 - vmalloc OK
[965381.750376] order=13, pages=  8192, size= 33554432 - vmalloc OK
[965382.135365] order=14, pages= 16384, size= 67108864 - vmalloc OK
[965382.885498] order=15, pages= 32768, size=134217728 - vmalloc OK
[965383.582883] order=16, pages= 65536, size=268435456 - vmalloc OK
[965404.678219] order=17, pages=131072, size=536870912 - vmalloc OK
[965405.098725] order=18, pages=262144, size=1073741824 - vmalloc failed
```

Оба способа: `kmalloc` и `__get_free_pages` (которые еще называют *когерентными* методами выделения памяти), выделяют свыше 1000 страниц (больше 4 Мбайт) *непрерывной* физической памяти, а `vmalloc` — больше 500 Мбайт (более 50% доступного на плате), но не гарантируя, что эта память размещена в физической памяти непрерывно.

Движемся далее, к Intel-архитектуре... Достаточно современный HP тонкий клиент T520 (процессор ARM GX-212JC, что может иметь влияние):

```
$ sudo inxi -MSCmxxx
System:
  Host: antix21 Kernel: 4.9.0-279-antix.1-amd64-smp arch: x86_64 bits: 64
    compiler: gcc v: 10.2.1 Desktop: IceWM v: 2.9.8 dm: slimski v: 1.5.0
    Distro: antiX-21_x64-base Grup Yorum 31 October 2021
    base: Debian GNU/Linux 11 (bullseye)
Machine:
  Type: Desktop System: Hewlett-Packard product: HP t520 Flexible Series TC
    v: N/A serial: MXL4470SVF Chassis: type: 4 serial: MXL4470SVF
  Mobo: Hewlett-Packard model: 21EF v: 00~ serial: MXL4470SVF UEFI: AMI
    v: L41 v01.09 date: 10/19/2016
Memory:
  RAM: total: 6.75 GiB used: 535.2 MiB (7.7%)
  Array-1: capacity: 8 GiB slots: 1 EC: None max-module-size: 8 GiB
    note: est.
```

```
  Device-1: Top type: DDR3 detail: synchronous unbuffered (unregistered)
    size: 8 GiB speed: 1333 MT/s volts: curr: 1.5 min: 1.35 max: 1.5
    width (bits): data: 64 total: 64 manufacturer: Samsung
    part-no: M471B1G73EB0-YK0 serial: 146F35D0
CPU:
  Info: dual core model: AMD GX-212JC SOC with Radeon R2E Graphics bits: 64
    type: MCP smt: <unsupported> arch: Puma rev: 1 cache: L1: 128 KiB
    L2: 1024 KiB
  Speed (MHz): avg: 1198 min/max: N/A volts: 0.9 V ext-clock: 100 MHz
    cores: 1: 1198 2: 1198 bogomips: 4790
  Flags: avx ht lm nx pae sse sse2 sse3 sse4_1 sse4_2 sse4a ssse3 svm
$ cat /proc/meminfo | grep MemTotal
MemTotal:        7075628 kB
$ free
               total        used        free      shared  buff/cache   available
Mem:         7075628      277084     6393892        5964      404652     6556848
Swap:         393212           0      393212
$ uname -a
Linux antix21 4.9.0-279-antix.1-amd64-smp #1 SMP PREEMPT Sun Aug 8 15:04:18 EEST 2021 x86_64
GNU/Linux
$ getconf PAGESIZE
4096
$ sudo insmod memmax.ko mode=0
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | tail -n12
[29432.858105] order= 0, pages=     1, size=     4096 - kmalloc OK
[29432.858111] order= 1, pages=     2, size=     8192 - kmalloc OK
[29432.858117] order= 2, pages=     4, size=    16384 - kmalloc OK
[29432.858123] order= 3, pages=     8, size=    32768 - kmalloc OK
[29432.858128] order= 4, pages=    16, size=    65536 - kmalloc OK
[29432.858132] order= 5, pages=    32, size=   131072 - kmalloc OK
[29432.858137] order= 6, pages=    64, size=   262144 - kmalloc OK
[29432.858144] order= 7, pages=   128, size=   524288 - kmalloc OK
[29432.858154] order= 8, pages=   256, size=  1048576 - kmalloc OK
[29432.858167] order= 9, pages=   512, size=  2097152 - kmalloc OK
[29432.858200] order=10, pages=  1024, size=  4194304 - kmalloc OK
[29432.858213] order=11, pages=  2048, size=  8388608 - kmalloc failed
$ sudo insmod memmax.ko mode=1
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | tail -n12
[29659.592424] order= 0, pages=     1, size=     4096 - __get_free_pages OK
[29659.592432] order= 1, pages=     2, size=     8192 - __get_free_pages OK
[29659.592438] order= 2, pages=     4, size=    16384 - __get_free_pages OK
[29659.592442] order= 3, pages=     8, size=    32768 - __get_free_pages OK
[29659.592445] order= 4, pages=    16, size=    65536 - __get_free_pages OK
[29659.592449] order= 5, pages=    32, size=   131072 - __get_free_pages OK
[29659.592454] order= 6, pages=    64, size=   262144 - __get_free_pages OK
[29659.592460] order= 7, pages=   128, size=   524288 - __get_free_pages OK
```

```
[29659.592468] order= 8, pages=   256, size=  1048576 - __get_free_pages OK
[29659.592479] order= 9, pages=   512, size=  2097152 - __get_free_pages OK
[29659.592503] order=10, pages=  1024, size=  4194304 - __get_free_pages OK
[29659.592517] order=11, pages=  2048, size=  8388608 - __get_free_pages failed
$ sudo insmod memmax.ko mode=2
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | grep order | tail -n22
[29765.017161] order= 0, pages=     1, size=     4096 - vmalloc OK
[29765.017168] order= 1, pages=     2, size=     8192 - vmalloc OK
[29765.017202] order= 2, pages=     4, size=    16384 - vmalloc OK
[29765.017467] order= 3, pages=     8, size=    32768 - vmalloc OK
[29765.017477] order= 4, pages=    16, size=    65536 - vmalloc OK
[29765.017494] order= 5, pages=    32, size=   131072 - vmalloc OK
[29765.017516] order= 6, pages=    64, size=   262144 - vmalloc OK
[29765.017556] order= 7, pages=   128, size=   524288 - vmalloc OK
[29765.017627] order= 8, pages=   256, size=  1048576 - vmalloc OK
[29765.017804] order= 9, pages=   512, size=  2097152 - vmalloc OK
[29765.018629] order=10, pages=  1024, size=  4194304 - vmalloc OK
[29765.019289] order=11, pages=  2048, size=  8388608 - vmalloc OK
[29765.020615] order=12, pages=  4096, size= 16777216 - vmalloc OK
[29765.023870] order=13, pages=  8192, size= 33554432 - vmalloc OK
[29765.029267] order=14, pages= 16384, size= 67108864 - vmalloc OK
[29765.040385] order=15, pages= 32768, size=134217728 - vmalloc OK
[29765.063024] order=16, pages= 65536, size=268435456 - vmalloc OK
[29765.108266] order=17, pages=131072, size=536870912 - vmalloc OK
[29765.198108] order=18, pages=262144, size=1073741824 - vmalloc OK
[29765.377598] order=19, pages=524288, size=2147483648 - vmalloc OK
[29765.741111] order=20, pages=1048576, size=4294967296 - vmalloc OK
[29765.931037] order=21, pages=2097152, size=8589934592 - vmalloc failed
```

Картина для AMD-архитектуры близкая к ARM: *когерентные* методы в состоянии выделить участок памяти порядка 1000 страниц, а `vmalloc` позволяет отобразить в единый блок (из кусков физической памяти) больше 50% доступной RAM — больше чем 4 Гбайт.

И, наконец, для полноты картины возьмем максимально крупную архитектуру — сервер промышленного уровня DELL PowerEdge R420 с объемом RAM 96 Гбайт:

```
$ sudo inxi -MCSxxx
System:    Host: R420 Kernel: 5.4.0-124-generic x86_64 bits: 64 compiler: gcc v: 9.4.0
           Desktop: Cinnamon 5.2.7
           wm: muffin 5.2.1 dm: LightDM 1.30.0 Distro: Linux Mint 20.3 Una
           base: Ubuntu 20.04 focal
Machine:   Type: Server System: Dell product: PowerEdge R420 v: N/A serial: 9DDFKY1
           Chassis: type: 23 serial: 9DDFKY1
           Mobo: Dell model: 0CN7CM v: A06 serial: ..CN1374035400RO.
           BIOS: Dell v: 2.9.0 date: 01/09/2020
CPU:       Topology: 2x 10-Core model: Intel Xeon E5-2470 v2 bits: 64
           type: MT MCP SMP arch: Ivy Bridge rev: 4
           L1 cache: 640 KiB L2 cache: 50.0 MiB L3 cache: 50.0 MiB
```

```
          flags: avx lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 192027
          Speed: 1200 MHz min/max: 1200/3200 MHz Core speeds (MHz): 1: 1312 2: 1424 3: 1569
          4: 1552 5: 1338 6: 1931 7: 1732 8: 1653 9: 1398 10: 1568 11: 1372 12: 1433 13: 1562
          14: 1359 15: 1469 16: 1655 17: 1425 18: 1339 19: 1824 20: 1533 21: 1819 22: 1497
          23: 1429 24: 1526 25: 2070 26: 1375 27: 1638 28: 1591 29: 1695 30: 1569 31: 1593
          32: 2360 33: 1599 34: 1264 35: 1543 36: 1866 37: 1861 38: 1786 39: 1790 40: 1560
$ cat /proc/meminfo | grep MemTotal
MemTotal:       98936056 kB
$ free
               всего        занято        свободно      общая    буф./врем.   доступно
Память:     98936048       1804512        95151500       67068     1980036    96257540
Подкачка:    2097148             0         2097148
$ sudo insmod memmax.ko mode=0
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | tail -n12
[11121.592421] order= 0, pages=     1, size=     4096 - kmalloc OK
[11121.592427] order= 1, pages=     2, size=     8192 - kmalloc OK
[11121.592432] order= 2, pages=     4, size=    16384 - kmalloc OK
[11121.592442] order= 3, pages=     8, size=    32768 - kmalloc OK
[11121.592458] order= 4, pages=    16, size=    65536 - kmalloc OK
[11121.592489] order= 5, pages=    32, size=   131072 - kmalloc OK
[11121.592549] order= 6, pages=    64, size=   262144 - kmalloc OK
[11121.592640] order= 7, pages=   128, size=   524288 - kmalloc OK
[11121.592847] order= 8, pages=   256, size=  1048576 - kmalloc OK
[11121.593511] order= 9, pages=   512, size=  2097152 - kmalloc OK
[11121.594340] order=10, pages=  1024, size=  4194304 - kmalloc OK
[11121.594346] order=11, pages=  2048, size=  8388608 - kmalloc failed
$ sudo insmod memmax.ko mode=1
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | tail -n12
[31651.671219] order= 0, pages=     1, size=     4096 - __get_free_pages OK
[31651.671222] order= 1, pages=     2, size=     8192 - __get_free_pages OK
[31651.671226] order= 2, pages=     4, size=    16384 - __get_free_pages OK
[31651.671233] order= 3, pages=     8, size=    32768 - __get_free_pages OK
[31651.671248] order= 4, pages=    16, size=    65536 - __get_free_pages OK
[31651.671278] order= 5, pages=    32, size=   131072 - __get_free_pages OK
[31651.671334] order= 6, pages=    64, size=   262144 - __get_free_pages OK
[31651.671448] order= 7, pages=   128, size=   524288 - __get_free_pages OK
[31651.671583] order= 8, pages=   256, size=  1048576 - __get_free_pages OK
[31651.671852] order= 9, pages=   512, size=  2097152 - __get_free_pages OK
[31651.672385] order=10, pages=  1024, size=  4194304 - __get_free_pages OK
[31651.672389] order=11, pages=  2048, size=  8388608 - __get_free_pages failed
$ sudo insmod memmax.ko mode=2
insmod: ERROR: could not insert module memmax.ko: Operation not permitted
$ dmesg | tail -n100 | grep vmalloc | grep order
[47225.572831] order= 0, pages=     1, size=     4096 - vmalloc OK
[47225.572835] order= 1, pages=     2, size=     8192 - vmalloc OK
[47225.572841] order= 2, pages=     4, size=    16384 - vmalloc OK
```

```
[47225.572850] order= 3, pages=     8, size=    32768 - vmalloc OK
[47225.572869] order= 4, pages=    16, size=    65536 - vmalloc OK
[47225.572894] order= 5, pages=    32, size=   131072 - vmalloc OK
[47225.572940] order= 6, pages=    64, size=   262144 - vmalloc OK
[47225.573046] order= 7, pages=   128, size=   524288 - vmalloc OK
[47225.573241] order= 8, pages=   256, size=  1048576 - vmalloc OK
[47225.573699] order= 9, pages=   512, size=  2097152 - vmalloc OK
[47225.574611] order=10, pages=  1024, size=  4194304 - vmalloc OK
[47225.576052] order=11, pages=  2048, size=  8388608 - vmalloc OK
[47225.579312] order=12, pages=  4096, size= 16777216 - vmalloc OK
[47225.584758] order=13, pages=  8192, size= 33554432 - vmalloc OK
[47225.601853] order=14, pages= 16384, size= 67108864 - vmalloc OK
[47225.644835] order=15, pages= 32768, size=134217728 - vmalloc OK
[47225.729240] order=16, pages= 65536, size=268435456 - vmalloc OK
[47225.940186] order=17, pages=131072, size=536870912 - vmalloc OK
[47226.323560] order=18, pages=262144, size=1073741824 - vmalloc OK
[47226.961322] order=19, pages=524288, size=2147483648 - vmalloc OK
[47228.104785] order=20, pages=1048576, size=4294967296 - vmalloc OK
[47230.397710] order=21, pages=2097152, size=8589934592 - vmalloc OK
[47235.035695] order=22, pages=4194304, size=17179869184 - vmalloc OK
[47244.582550] order=23, pages=8388608, size=34359738368 - vmalloc OK
[47268.927730] order=24, pages=16777216, size=68719476736 - vmalloc OK
[47270.248160] order=25, pages=33554432, size=137438953472 - vmalloc failed
```

При объеме памяти на порядок больше (96 Гбайт) *когерентные* методы всё так же выделяют максимальный участок памяти порядка 1000 страниц, но `vmalloc` позволяет *отобразить* в *единый* блок логической памяти из участков физической памяти больше 65 Гбайт.

Следующая параметр, который вам явно потребуется оценить при тщательной проработке нового модуля ядра, — это порядок временных затрат на выделение блока при использовании того или иного механизма. Насколько велики будут временные потери? Вот код такого модуля-теста:

memtim.c

```
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/vmalloc.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
   t1 = rdtsc();
   t2 = rdtsc();
   calibr += (t2 - t1);
}
calibr = calibr / CYCLES;
for(i = 0; i < sizeof(mfun) / sizeof(mfun[0]); i++) {
   char *kbuf;
   char msg[200];
   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
      t1 = rdtsc();
      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;
      t2 = rdtsc();
      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");
MODULE_DESCRIPTION("memory allocation speed test");
MODULE_LICENSE("GPL v2");
```

Результаты этого теста я приведу только для одной системы — из-за их объемности и громоздкости. Демонстрировать это для процессоров ARM не могу — в ARM не существует аппаратного счетчика процессорных тактов (или я его не знаю) — команды процессора с ассемблерной мнемоникой `RDTSC` и соответственно вызова `rdtsc()` в API ядра (вот так в самых неожиданных местах «вылазит» разнообразие поддерживаемых Linux аппаратных платформ — это всегда нужно иметь в виду).

Вы это можете повторить для своего компьютера и своей версии ядра:

```
$ sudo inxi -M
Machine:
  Type: Desktop System: Hewlett-Packard product: HP t520 Flexible Series TC
    v: N/A serial: MXL4470SVF
  Mobo: Hewlett-Packard model: 21EF v: 00~ serial: MXL4470SVF UEFI: AMI
    v: L41 v01.09 date: 10/19/2016
$ sudo insmod memtim.ko size=5
insmod: ERROR: could not insert module memtim.ko: Operation not permitted
$ dmesg | tail -n4
[34296.253657] size = 5 order = 1(4096)
[34296.254609] proc. cycles for allocate kmalloc : 67
[34296.255699] proc. cycles for allocate __get_free_pages : 391
[34296.258486] proc. cycles for allocate vmalloc : 1703
$ sudo insmod memtim.ko size=1000
insmod: ERROR: could not insert module memtim.ko: Operation not permitted
$ dmesg | tail -n4
[34325.154064] size = 1000 order = 1(4096)
[34325.155003] proc. cycles for allocate kmalloc : 70
[34325.156074] proc. cycles for allocate __get_free_pages : 389
[34325.158651] proc. cycles for allocate vmalloc : 1467
$ sudo insmod memtim.ko size=4096
insmod: ERROR: could not insert module memtim.ko: Operation not permitted
$ dmesg | tail -n4
[34350.536784] size = 4096 order = 1(4096)
[34350.537734] proc. cycles for allocate kmalloc : 69
[34350.538839] proc. cycles for allocate __get_free_pages : 383
[34350.541707] proc. cycles for allocate vmalloc : 1743
```

```
$ sudo insmod memtim.ko size=32768
insmod: ERROR: could not insert module memtim.ko: Operation not permitted
$ dmesg | tail -n4
[34442.371340] size = 32768 order = 4(32768)
[34442.373066] proc. cycles for allocate kmalloc : 540
[34442.374383] proc. cycles for allocate __get_free_pages : 478
[34442.379637] proc. cycles for allocate vmalloc : 3263
$ sudo insmod memtim.ko size=65535
insmod: ERROR: could not insert module memtim.ko: Operation not permitted
$ dmesg | tail -n4
[34460.816123] size = 65535 order = 5(65536)
[34460.818068] proc. cycles for allocate kmalloc : 672
[34460.819638] proc. cycles for allocate __get_free_pages : 608
[34460.826817] proc. cycles for allocate vmalloc : 4816
$ sudo insmod memtim.ko size=262144
insmod: ERROR: could not insert module memtim.ko: Operation not permitted
$ dmesg | tail -n4
[34486.646728] size = 262144 order = 7(262144)
[34486.649490] proc. cycles for allocate kmalloc : 1158
[34486.652320] proc. cycles for allocate __get_free_pages : 1278
[34486.672277] proc. cycles for allocate vmalloc : 14984
$ sudo insmod memtim.ko size=4096000
insmod: ERROR: could not insert module memtim.ko: Operation not permitted
$ dmesg | tail -n4
[34518.148307] size = 4096000 order = 11(4194304)
[34518.175789] proc. cycles for allocate kmalloc : 19623
[34518.175794] proc. cycles for allocate __get_free_pages : failed
[34518.567515] proc. cycles for allocate vmalloc : 265116
$ sudo insmod memtim.ko size=40960000
insmod: ERROR: could not insert module memtim.ko: Operation not permitted
$ dmesg | tail -n4
[34533.542741] size = 40960000 order = 15(67108864)
[34533.543166] proc. cycles for allocate kmalloc : failed
[34533.543169] proc. cycles for allocate __get_free_pages : failed
[34537.717573] proc. cycles for allocate vmalloc : 2751904
```

Можно укрупненно заключить следующее:

- при размерах распределяемых блоков меньше размера страницы (4096) время распределения практически *не зависит* от запрашиваемого размера — блок выделяется страницами;
- при распределении малых блоков (1–2 страницы) разница между `kmalloc()` и `vmalloc()` внушительная и составляет 20–30 раз;
- при увеличении размеров запрашиваемого блока различие смягчается, но на больших объемах тоже составляет до порядка.

В этих различиях нет ничего страшного, учитывая ту гибкость и диапазон, которые обеспечивает как раз `vmalloc()`, если только речь не идет о быстром и частом получении/удалении малых блоков (как, например, сокеты и буферы) в динамике.

- Приложение 3 -
Четыре способа записи в защищенную страницу

> В этом приложении приводится моя статья 2015 года, опубликованная впервые, кажется, на Хабрахабре. Статья представлена в своем первозданном виде (с некоторыми стилистическими правками), поскольку позже я не занимался этим вопросом, но она может представлять серьезный интерес для разработчиков модулей ядра, потому как содержит обобщение предварительно опубликованных материалов на эту тему (все упоминаемые коды находятся сейчас в папке 4write сопровождающего книгу файлового архива). Итак (далее следует текст статьи)...

Целью выяснения является выполнение записи по *аппаратно* защищенному от записи адресу памяти в архитектуре *x*86/*x*86_64. И то, как это делается в операционной системе Linux. И, естественно, в режиме ядра Linux, потому что в пользовательском пространстве такие трюки запрещены. Бывает, знаете ли, непреодолимое желание записать в защищенную область... когда программируешь вирус или троян...

Описание проблемы

А если серьезно, то проблема записи в защищенные от записи страницы оперативной памяти возникает время от времени при программировании модулей ядра под Linux. Например, при модификации селекторной таблицы системных вызовов `sys_call_table` для модификации, встраивания, имплементации, подмены, перехвата системного вызова — в разных публикациях это действие называют по-разному. Но не только для этих целей...

Очень коротко ситуация выглядит так:

- в архитектуре *x*86 существует защитный механизм, который при попытке записи в защищенные от записи страницы памяти приводит к возбуждению исключения;
- права доступа к странице (разрешение или запрет записи) описываются битом `_PAGE_BIT_RW` (первый) в соответствующей этой странице структуре типа `pte_t`. Сброс этого бита запрещает запись в страницу;
- со стороны процессора контролем защитой записи управляет бит `X86_CR0_WP` (16-й) системного управляющего регистра `CR0` — при установленном этом бите попытка записи в защищенную от записи страницу возбуждает исключение процессора.

О актуальности задачи записи в аппаратно защищенную область памяти говорит и заметное число публикаций на эту тему, и число предлагаемых способов решения задачи. Последовательному рассмотрению способов и посвящена оставшаяся часть обзора... По каждому из способов будет приведено:

- образец кода, испытанный и пригодный для использования;
- известные мне ссылки на авторство подобного кода (хотя это очень относительно, потому как по решающим эту задачу способам существует достаточно много независимых источников).

Отключение страничной защиты: ассемблер

Простейшим решением проблемы является временное отключение страничной защиты сбросом бита `X86_CR0_WP` регистра `CR0`. Этот способ я использую добрый десяток лет, и он упоминается в нескольких публикациях разных лет — например, в «WP: Safe or Not?» (Dan Rosenberg, 2011 г., **http://vulnfactory.org/blog/2011/08/12/wp-safe-or-not/**). Один из путей такой реализации — инлайновые ассемблерные вставки (макросы, расширение компилятора GCC). В моем варианте и в демонстрационном тесте этот вариант выглядит так (файл rw_cr0.c):

```
static inline void rw_enable( void ) {
   asm( "cli \n"
        "pushl %eax \n"
        "movl %cr0, %eax \n"
        "andl $0xfffeffff, %eax \n"
        "movl %eax, %cr0 \n"
        "popl %eax" );
}

static inline void rw_disable( void ) {
   asm( "pushl %eax \n"
        "movl %cr0, %eax \n"
        "orl $0x00010000, %eax \n"
        "movl %eax, %cr0 \n"
        "popl %eax \n"
        "sti " );
}
```

Сохранение и восстановление регистра `eax` можно исключить — здесь это сохранено... для чистоты эксперимента.

Первое, что всегда возражают на такой метод по первому взгляду: поскольку это основано на управлении конкретным процессором, в SMP-системах *между* установкой регистра `CR0` и записью в защищаемую область выполнение модуля может быть перепланировано на другой процессор, для которого страничная защита не отключена. Вероятность такого стечения обстоятельств не выше, чем если бы вас в центре Москвы укусила змея, ускользнувшая из зоопарка. Но вероятность такая существует, и она конечна, хотя и исчезающе мала. Чтобы воспрепятствовать воз-

никновению этой ситуации, из ассемблерного кода мы запрещаем локальные прерывания процессора операцией `cli` перед записью и освобождаем прерывания только после завершения записи операцией `sti` (точно так же делает и Dan Rosenberg в упоминавшейся публикации).

Что намного неприятнее в показанном коде, это то, что он написан для 32-битной архитектуры (*i*386), а в 64-битной архитектуре не будет не только выполняться, но даже компилироваться. Разрешить это можно тем, что иметь различные коды, зависящие от архитектуры:

```
#ifdef __i386__
   // ... то, что было показано выше
#else
static inline void rw_enable( void ) {
   asm( "cli \n"
        "pushq %rax \n"
        "movq %cr0, %rax \n"
        "andq $0xfffffffffffeffff, %rax \n"
        "movq %rax, %cr0 \n"
        "popq %rax " );
}

static inline void rw_disable( void ) {
   asm( "pushq %rax \n"
        "movq %cr0, %rax \n"
        "xorq $0x0000000000001000, %rax \n"
        "movq %rax, %cr0 \n"
        "popq %rax \n"
        "sti " );
}
#endif
```

Отключение страничной защиты: API ядра

Можно сделать то же самое, но опираясь не ассемблерный код, а на API ядра (файл rw_pax.c). Вот фрагмент такого кода почти в том же неизменном виде, как его приводит Dan Rosenberg:

```
#include <linux/preempt.h>
#include <asm/paravirt.h>
#include <asm-generic/bug.h>
#include <linux/version.h>

static inline unsigned long native_pax_open_kernel(void) {
    unsigned long cr0;
    preempt_disable();
    barrier();
    cr0 = read_cr0() ^ X86_CR0_WP;
```

```
    BUG_ON(unlikely(cr0 & X86_CR0_WP));
    write_cr0(cr0);
    return cr0 ^ X86_CR0_WP;
}

static inline unsigned long native_pax_close_kernel(void) {
    unsigned long cr0;
    cr0 = read_cr0() ^ X86_CR0_WP;
    BUG_ON( unlikely(!(cr0 & X86_CR0_WP)));
    write_cr0(cr0);
    barrier();
#if LINUX_VERSION_CODE < KERNEL_VERSION(3,14,0)
    preempt_enable_no_resched();
#else
    preempt_count_dec();
#endif
    return cr0 ^ X86_CR0_WP;
}
```

Примечание «почти» относится к тому, что вызов `preempt_enable_no_resched()` был доступен до ядра 3.13 (в 2011 г., когда писалась статья). Начиная с ядра 3.14 и далее, этот вызов закрыт вот таким условным препроцессорным определением:

```
#ifdef MODULE
/*
 * Modules have no business playing preemption tricks.
 */
#undef sched_preempt_enable_no_resched
#undef preempt_enable_no_resched
```

Но макросы `preempt_enable_no_resched()` и `preempt_count_dec()` определены в более поздних ядрах практически идентично.

Куда неприятнее то обстоятельство, что показанный код благополучно выполняется и в поздних версиях (старше 3.14) ядра, но вскоре после его выполнения из других приложений появляются предупреждающие (warning) сообщения ядра вида:

```
[  337.230937] ------------[ cut here ]------------
[  337.230949] WARNING: CPU: 1 PID: 3410 at /build/buildd/linux-lts-utopic-
3.16.0/init/main.c:802 do_one_initcall+0x1cb/0x1f0()
[  337.230955] initcall rw_init+0x0/0x1000 [srw] returned with preemption imbalance
```

Я не вникал детально в происходящее, но это как-то связано с нарушением балансировки работ между процессорами SMP или оценкой такой балансировки.

Возникающие в ядре даже предупреждения — это уже достаточно серьезно, от них хотелось бы избавиться. Этого можно достигнуть, повторив трюк с локальными прерываниями из ранее рассмотренного ассемблерного кода (файл rw_pai.c):

```
static inline unsigned long native_pai_open_kernel(void) {
   unsigned long cr0;
```

```
    local_irq_disable();
    barrier();
    cr0 = read_cr0() ^ X86_CR0_WP;
    BUG_ON( unlikely(cr0 & X86_CR0_WP));
    write_cr0(cr0);
    return cr0 ^ X86_CR0_WP;
}

static inline unsigned long native_pai_close_kernel( void ) {
    unsigned long cr0;
    cr0 = read_cr0() ^ X86_CR0_WP;
    BUG_ON( unlikely( !( cr0 & X86_CR0_WP ) ) );
    write_cr0( cr0 );
    barrier();
    local_irq_enable();
    return cr0 ^ X86_CR0_WP;
}
```

Этот код успешно и компилируется, и работает в архитектурах как 32-, так и 64-битных, и в этом его достоинство перед предыдущим.

Снятие защиты со страницы памяти

Следующий предложенный способ — установка бита `_PAGE_BIT_RW` в PTE-записи, описывающей интересующую нас страницу памяти (файл rw_pte.c):

```
#include <asm/pgtable_types.h>
#include <asm/tlbflush.h>

static inline void mem_setrw(void **table) {
    unsigned int l;
    pte_t *pte = lookup_address((long unsigned int)table, &l);
    pte->pte |= _PAGE_RW;
    __flush_tlb_one((unsigned long)table);
}

static inline void mem_setro(void **table) {
    unsigned int l;
    pte_t *pte = lookup_address((long unsigned int)table, &l);
    pte->pte &= ~_PAGE_RW;
    __flush_tlb_one((unsigned long)table);
}
```

По логике выполнения код абсолютно понятен. Сам код в виде *почти* том, как он здесь показан, я впервые встречал в обсуждении на Хабрахабр («Кошерный способ модификации защищенных от записи областей ядра Linux», Ilya V. Matveychikov, 2013 г., **https://habr.com/ru/post/207122/**), а позже, гораздо обстоятельнее, в обсуж-

дении на форуме («Модификация системных вызовов», 2015 г., **https://linux-ru.ru/viewtopic.php?f=18&t=4346&start=10**).

Этот код проверен и в 32- и в 64-битной архитектуре.

Наложение отображения участка памяти

Еще один способ (последний из рассматриваемых на сегодня) предложен в той же статье «Кошерный способ модификации защищенных от записи областей ядра Linux». Я не могу сказать ничего ни хорошего, ни плохого о кулинарных пристрастиях автора по части его национальной кухни, но в отношении предложенного технического приема должен отметить, что способ оригинален и красив (файл rw_map.c):

```
static void *map_writable( void *addr, size_t len ) {
   void *vaddr;
   int nr_pages = DIV_ROUND_UP( offset_in_page( addr ) + len, PAGE_SIZE );
   struct page **pages = kmalloc( nr_pages * sizeof(*pages), GFP_KERNEL );
   void *page_addr = (void*)( (unsigned long)addr & PAGE_MASK );
   int i;
   if( pages == NULL )
      return NULL;
   for( i = 0; i < nr_pages; i++ ) {
      if( __module_address( (unsigned long)page_addr ) == NULL ) {
         pages[ i ] = virt_to_page( page_addr );
         WARN_ON( !PageReserved( pages[ i ] ) );
      } else {
         pages[i] = vmalloc_to_page(page_addr);
      }
      if( pages[ i ] == NULL ) {
         kfree( pages );
         return NULL;
      }
      page_addr += PAGE_SIZE;
   }
   vaddr = vmap( pages, nr_pages, VM_MAP, PAGE_KERNEL );
   kfree( pages );
   if( vaddr == NULL )
      return NULL;
   return vaddr + offset_in_page( addr );
}

static void unmap_writable( void *addr ) {
   void *page_addr = (void*)( (unsigned long)addr & PAGE_MASK );
   vfree( page_addr );
}
```

Этот способ работает и в 32-, и в 64-битной архитектуре. В некоторый минус ему можно отнести определенную громоздкость для решения достаточно простой зада-

чи («из пушки по воробьям») — притом что, на первый взгляд, в нем не видно существенных преимуществ перед предыдущими способами. Но эта техника (и практически в неизменном виде этот код) может быть с успехом использована для более широкого круга задач, чем обсуждаемая.

Тест выполнения

А теперь, чтобы не быть голословным, пришло время проверить все здесь описанное натурным экспериментом. Для проверки создадим модуль ядра (файл srw.c):

```
#include "rw_cr0.c"
#include "rw_pte.c"
#include "rw_pax.c"
#include "rw_map.c"
#include "rw_pai.c"

#define PREFIX "! "
#define LOG(...) printk( KERN_INFO PREFIX __VA_ARGS__ )
#define ERR(...) printk( KERN_ERR PREFIX __VA_ARGS__ )

#define __NR_rw_test 31                          // неиспользуемая позиция sys_call_table

static int mode = 0;
module_param( mode, uint, 0 );

#define do_write( addr, val ) {         \
   LOG( "writing address %p\n", addr ); \
   *addr = val;                         \
}

static bool write( void** addr, void* val ) {
   switch( mode ) {
      case 0:
         rw_enable();
         do_write( addr, val );
         rw_disable();
         return true;
      case 1:
         native_pax_open_kernel();
         do_write( addr, val );
         native_pax_close_kernel();
         return true;
      case 2:
         mem_setrw( addr );
         do_write( addr, val );
         mem_setro( addr );
         return true;
```

```
      case 3:
         addr = map_writable( (void*)addr, sizeof( val ) );
         if( NULL == addr ) {
            ERR( "wrong mapping\n" );
            return false;
         }
         do_write( addr, val );
         unmap_writable( addr );
         return true;
      case 4:
         native_pai_open_kernel();
         do_write( addr, val );
         native_pai_close_kernel();
         return true;
      default:
         ERR( "illegal mode %d\n", mode );
         return false;
   }
}

static int __init rw_init( void ) {
   void **taddr;                                  // адрес sys_call_table
   asmlinkage long (*sys_ni_syscall) ( void );    // оригинальный вызов __NR_rw_test
   if( NULL == ( taddr = (void**)kallsyms_lookup_name( "sys_call_table" ) ) ) {
      ERR( "sys_call_table not found\n" ); return -EFAULT;
   }
   LOG( "sys_call_table address = %p\n", taddr );
   sys_ni_syscall = (void*)taddr[ __NR_rw_test ]; // сохранить оригинал
   if( !write( taddr + __NR_rw_test, (void*)0x12345 ) ) return -EINVAL;
   LOG( "modified sys_call_table[%d] = %p\n", __NR_rw_test, taddr[ __NR_rw_test ] );
   if( !write( taddr + __NR_rw_test, (void*)sys_ni_syscall ) ) return -EINVAL;
   LOG( "restored sys_call_table[%d] = %p\n", __NR_rw_test, taddr[ __NR_rw_test ] );
   return -EPERM;
}

module_init( rw_init );
```

Некоторая тяжеловесность и громоздкость этого кода обусловлены только тем, что:

- в едином коде нужно было согласовать различные прототипы функций, разрешающих запись, но принадлежащих разным способам (по действию они одинаковы, но вызываются по-разному);
- реализация для разных способов сохранялась максимально приближенной к тому, как она записана у разных авторов (изменения вносились только для соответствия синтаксиса более свежим версиям ядрам). Этим и объясняется разнообразие прототипов функций.

И вот как это выглядит только в одной из тестируемых архитектур (реально тестировалось не менее пяти различных архитектур и версий ядра):

```
$ uname -r
3.16.0-48-generic
$ uname -m
x86_64
$ sudo insmod srw.ko mode=0
insmod: ERROR: could not insert module srw.ko: Operation not permitted
$ dmesg | tail -n6
[ 7258.575977] ! detected 64-bit platform
[ 7258.584504] ! sys_call_table address = ffffffff81801460
[ 7258.584579] ! writing address ffffffff81801558
[ 7258.584653] ! modified sys_call_table[31] = 0000000000012345
[ 7258.584654] ! writing address ffffffff81801558
[ 7258.584666] ! restored sys_call_table[31] = ffffffff812db550
$ sudo insmod srw.ko mode=2
insmod: ERROR: could not insert module srw.ko: Operation not permitted
$ dmesg | tail -n6
[ 7282.625539] ! detected 64-bit platform
[ 7282.633020] ! sys_call_table address = ffffffff81801460
[ 7282.633129] ! writing address ffffffff81801558
[ 7282.633178] ! modified sys_call_table[31] = 0000000000012345
[ 7282.633228] ! writing address ffffffff81801558
[ 7282.633291] ! restored sys_call_table[31] = ffffffff812db550
$ sudo insmod srw.ko mode=3
insmod: ERROR: could not insert module srw.ko: Operation not permitted
$ dmesg | tail -n6
[ 7297.040272] ! detected 64-bit platform
[ 7297.059764] ! sys_call_table address = ffffffff81801460
[ 7297.065930] ! writing address ffffc900001e6558
[ 7297.066000] ! modified sys_call_table[31] = 0000000000012345
[ 7297.066035] ! writing address ffffc9000033d558
[ 7297.066073] ! restored sys_call_table[31] = ffffffff812db550
$ sudo insmod srw.ko mode=4
insmod: ERROR: could not insert module srw.ko: Operation not permitted
$ dmesg | tail -n6
[ 7309.831119] ! detected 64-bit platform
[ 7309.836299] ! sys_call_table address = ffffffff81801460
[ 7309.836311] ! writing address ffffffff81801558
[ 7309.836359] ! modified sys_call_table[31] = 0000000000012345
[ 7309.836368] ! writing address ffffffff81801558
[ 7309.836424] ! restored sys_call_table[31] = ffffffff812db550
```

Обсуждение

Этот обзор представлен не в качестве учебника или руководства к действию. Здесь только систематически собраны разные приемы с эквивалентными, по существу, действиями, используемые разными авторами.

Интересно было бы продолжить обсуждение относительно преимуществ и недостатков каждого из рассмотренных способов.

Позднее дополнение

На этом заканчивался текст статьи... А теперь, через 7 лет после ее написания, я проверил выполнение кода тестов без внесения каких-либо изменений в код:

```
$ uname -a
Linux R420 5.4.0-124-generic #140-Ubuntu SMP Thu Aug 4 02:23:37 UTC 2022 x86_64 x86_64 x86_64
GNU/Linux
$ sudo insmod srw.ko mode=0
insmod: ERROR: could not insert module srw.ko: Operation not permitted
olej@R420:~/2022/own.BOOKs/BHV.kernel/examples/4write$ dmesg | tail -n6
[ 8268.065963] ! sys_call_table address = ffffffffa56013c0
[ 8268.065964] ! original sys_call_table address = ffffffffa4603390
[ 8268.065965] ! writing address ffffffffa5601a70
[ 8268.065966] ! modified sys_call_table[214] = 0000000000012345
[ 8268.065967] ! writing address ffffffffa5601a70
[ 8268.065968] ! restored sys_call_table[214] = ffffffffa4603390
$ sudo insmod srw.ko mode=2
insmod: ERROR: could not insert module srw.ko: Operation not permitted
olej@R420:~/2022/own.BOOKs/BHV.kernel/examples/4write$ dmesg | tail -n6
[ 8321.258243] ! sys_call_table address = ffffffffa56013c0
[ 8321.258244] ! original sys_call_table address = ffffffffa4603390
[ 8321.258245] ! writing address ffffffffa5601a70
[ 8321.258247] ! modified sys_call_table[214] = 0000000000012345
[ 8321.258248] ! writing address ffffffffa5601a70
[ 8321.258248] ! restored sys_call_table[214] = ffffffffa4603390
$ sudo insmod srw.ko mode=3
insmod: ERROR: could not insert module srw.ko: Operation not permitted
olej@R420:~/2022/own.BOOKs/BHV.kernel/examples/4write$ dmesg | tail -n6
[ 8378.531808] ! sys_call_table address = ffffffffa56013c0
[ 8378.531809] ! original sys_call_table address = ffffffffa4603390
[ 8378.531815] ! writing address ffffbf788648fa70
[ 8378.531817] ! modified sys_call_table[214] = 0000000000012345
[ 8378.531819] ! writing address ffffbf7886586a70
[ 8378.531820] ! restored sys_call_table[214] = ffffffffa4603390
$ sudo insmod srw.ko mode=4
Убито
```

Напомню — чтобы не лезть глубоко разбираться в непростой код, — что проверки в этих тестах заключаются в полной *подмене* системного вызова Linux на свой собственный адрес (`0x12345`) обработчика, он хотя и бессмысленный (сознательно) в примере, но может быть и адресом вашей собственной функции обработки нового системного вызова.

Примечание

В дерево примеров, в ту же папку 4write, вложены папки: find, add_sys, new_sys, netm, связанные с модификацией и созданием новых собственных системных вызовов, они же включают тексты статей того же периода, детально описывающие эти коды (файлы этих текстов — с расширением odt — могут быть открыты с помощью приложения Writer офисного пакета OpenOffice. MS Word тоже открывает такие файлы, но после ряда предупреждений). Здесь они не будут комментироваться.

Из четырех обсуждаемых способов три на сегодня (ядро 5.4) работают безупречно. С четвертым, думаю, вы, при желании, разберетесь самостоятельно. Можете использовать тот, какой вам больше придется по вкусу... Но не обольщайтесь: то, что это работает в каком-то одном *дистрибутиве* (не версии ядра!), вовсе не означает, что он заработает при переносе в другой дистрибутив, — все зависит от *конфигурационных параметров*, с которыми собиралось ядро этого дистрибутива! Вот этих:

```
$ cat /boot/config-`uname -r` | grep KALLSYMS
CONFIG_KALLSYMS=y
CONFIG_KALLSYMS_ALL=y
CONFIG_KALLSYMS_ABSOLUTE_PERCPU=y
CONFIG_KALLSYMS_BASE_RELATIVE=y
```

Но не только этих... И для того чтобы добиться приемлемых эффектов, вам нужно проявить творческий поиск и произвести множество экспериментов.

Источники информации

1. Peter Jay Salzman, Michael Burian, Ori Pomerantz, Bob Mottram, Jim Huang. The Linux Kernel Module Programming Guide. April 28, 2022. **https://sysprog21.github.io/lkmpg/**.

   Peter Jay Salzman, Michael Burian, Ori Pomerantz Copyright 2001, Peter Jay Salzman. The Linux Kernel Module Programming Guide (Руководство по программированию модулей ядра Linux). 2004-05-16, ver 2.6.0. Перевод Андрей Киселёв: **https://textarchive.ru/c-2878586.html**.

2. Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Linux Device Drivers (3rd ed.), 2005, 2001, 1998, Publisher(s): O'Reilly Media, Inc. ISBN 9780596005900. **https://www.oreilly.com/library/view/linux-device-drivers/0596005903/.**

3. Robert Love. Linux Kernel Development (3rd ed.), 2010.

   Русское 2-е изд.: Р. Лав. Разработка ядра Linux. — М.: Издат. дом «Вильямс», 2006.

4. Wolfgang Mauerer. Professional Linux Kernel Architecture (Wrox Programmer to Programmer), Wiley Publishing Inc., 2008.

5. Sreekrishnan Venkateswaran. Essential Linux Device Drivers. — Prentice Hall, 2008. Сайт книги: **http://elinuxdd.com**. Архив кодов примеров: **http://elinuxdd.com/~elinuxdd/elinuxdd.docs/listings/**.

6. Jerry Cooperstein. Writing Linux Device Drivers. 2009. ISBN-13 9781448672387, ISBN-10 1448672384. T. 1: A guide with exercises. T. 2: Lab Solutions. **https://www.amazon.com/gp/product/1448672384/**.

7. Клаудия Зальзберг Родригес, Гордон Фишер, Стивен Смолски. Linux. Азбука ядра: пер. с англ. — М.: Кудиц-образ, 2007.

8. Гриффитс А. GCC. Полное руководство. Platinum Edition: пер. с англ. — М.: ДиаСофт, 2004. ISBN 966-7992-33-0.

9. Цилюрик О., Горошко Е. QNX/UNIX: анатомия параллелизма. — СПб.: Символ-Плюс, 2005. ISBN 5-93286-088-X.

10. Бовет Д., Чезати М. Ядро Linux: пер. с англ. — 3-е изд. — СПб.: БХВ-Петербург, 2007. ISBN 978-5-94157-957-0.

11. Крищенко В. А., Рязанова Н. Ю. Основы программирования в ядре операционной системы GNU/Linux. Сдано в издательство МГТУ в 2008 году.

12. Greg Kroab-Hartman. Linux Kernel in a Nutshell. O'Reilly Vtdia, Inc., 2007. ISBN-10 0-596-10079-5. **https://www.amazon.com/Linux-Kernel-Nutshell-OReilly/dp/0596100795**.

13. The Linux Kernel API. **https://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.

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.

16. David Mosberger, Stephane Eranian. IA-64 Linux Kernel. — Hewlet-Packard Company, Prentice Hall PTR, 2002.

17. Rajaram Regupathy. Bootstrap Yourself with Linux-USB Stack: Design, Develop, Debug, and Validate Embedded USB. Course Technology, a part of Cengage Lernining, 2012. ISBN-10 1-4354-5786-2.

18. Стивенс У. Р. UNIX: взаимодействие процессов. — СПб.: Питер, 2003. ISBN 5-318-00534-9.

19. У. Р. Стивенс, Стивен А. Раго. UNIX. Профессиональное программирование. — 3-е изд. — СПб.: Издат. дом «Питер». ISBN 978-5-4461-0649-3. **https://www.piter.com/collection/all/product/unix-professionalnoe-programmirovanie-3-e-izd**. Полный архив примеров кодов к этой книге может быть взят здесь: **http://www.kohala.com/start/apue.linux.tar.Z**.

20. W. Richard Stevens. Home Page (ресурс полного собрания книг и публикаций У. Р. Стивенса). **http://www.kohala.com/start/**.

21. Tigran Aivazian (**tigran@veritas.com**). Внутреннее устройство ядра Linux 2.4. — 21 Oct. 2001. Перевод Андрей Киселев: **https://www.opennet.ru/docs/RUS/lki/**.

22. GNU Make. Программа управления компиляцией. GNU make. Версия 3.79. — Апр. 2000. Авторы: Richard M. Stallman и Roland McGrath. Перевод Владимир Игнатов, 2000: **http://linux.yaroslavl.ru/docs/prog/gnu_make_3-79_russian_manual.html**.

23. Отладчик GNU уровня исходного кода. Восьмая редакция, для GDB версии 5.0. — Март 2000. Авторы: Ричард Столмен, Роланд Пеш, Стан Шебс и др. **http://linux.yaroslavl.ru/docs/altlinux/doc-gnu/gdb/gdb.html**.

24. Cristian Benvenuti. Understanding Linux Network Internals. — O'Reilly Media, Inc., 2006. ISBN 978-0-596-00255-8.

25. Соловьев А. Разработка модулей ядра ОС Linux (Kernel newbie's manual). **http://rus-linux.net/MyLDP/BOOKS/knm.pdf**.

26. Зубков С. В. Assembler для DOS, Windows, UNIX. — М.: ДМК Пресс, 2000. ISBN 5-94074-003-0.

27. Грибенко Д. Ассемблер в Linux для программистов С. — 1 февр. 2009. **http://wasm.ru/article.php?article=asm_linux_for_c**.

28. Building a custom kernel, Fedora WIKI. **http://fedoraproject.org/wiki/Docs/CustomKernel**.

29. Распределитель SLAB для управления памятью в Linux (связанный со SLAB код, описанный в этой статье, основан на ядре Linux v 4.7). **https://russianblogs.com/article/6796303641/**.

30. Clark L. Coleman, Using Inline Assembly With gcc. — 11 Jan. 2000. **https://www.cs.virginia.edu/~clc5q/gcc-inline-asm.pdf**.

31. Курочкин П. Разработка драйверов для USB-устройств под Linux. — 19 Nov. 2006. **http://www.opennet.ru/base/dev/write_linux_driver.txt.html**.

32. Реализация низкоуровневой поддержки шины PCI в ядре Linux. — 5 Aug. 2004. **http://www.opennet.ru/base/dev/pci_linux_kernel.txt.html**.

33. Цилюрик О. Модули ядра Linux. Серия из 77 статей, опубликованных на сайте IBM Developer Works. Первая статья цикла: **https://www.ibm.com/developerworks/ru/library/l-linux_kernel_01/**, последняя статья цикла: **https://www.ibm.com/developerworks/ru/library/l-linux_kernel_77/**.

34. Lennart Poettering. systemd для администраторов. Перевод Сергей Пташник — периодически обновляемое руководство по systemd, последняя редакция 28 октября 2017 г. **http://www2.kangran.su/~nnz/pub/s4a/s4a_latest.pdf**.

35. Rami Rosen. Linux Kernel Networking: Implementation and Theory. ISBN-10 143026196X, ISBN-13 9781430261964. — 22 Dec. 2013. **https://www.amazon.com/Linux-Kernel-Networking-Implementation-Experts/dp/143026196X**.

36. Lixiang Yang. The Art of Linux Kernel Design: Illustrating the Operating System Design Principle and Implementation. — 1st ed. ISBN-10 1466518030, ISBN-13 978-1466518032. **https://www.amazon.com/Art-Linux-Kernel-Design/dp/1466518030**.

37. Максвелл С. Ядро Linux в комментариях. — Киев: ДиаСофт, 2000. ISBN 966-7393-46-1.

38. Цилюрик О. 4 статьи о системных вызовах Linux. **https://mylinuxprog.blogspot.com/2015/10/4-linux.html**.

39. Harald Kipp: ARM GCC Inline Assembler Cookbook. **http://www.ethernut.de/en/documents/arm-inline-asm.html**.

40. GCC online documentation, these are manuals for the latest full releases. **http://gcc.gnu.org/onlinedocs/**.

41. Немоляев А. В. Популярно о USB. — Екатеринбург, 2015. **http://rus-linux.net/MyLDP/BOOKS/pop_about_USB.pdf?ysclid=l464pizoev841223049**.

42. Erdfelt J. Libusb Developers Guide. **http://epic-beta.kavli.tudelft.nl/share/doc/libusb-devel-0.1.12/html/index.html**.

43. Libusb-1.0 API Reference. **https://libusb.sourceforge.io/api-1.0/**.

44. The Linux Kernel API. **https://www.kernel.org/doc/html/latest/core-api/kernel-api.html**, **https://www.kernel.org/doc/htmldocs/kernel-api/**, **https://www.kernel.org/doc/html/v5.0/core-api/kernel-api.html**.

45. LXR (Linux Cross-Referencer). Ресурсы перекрестного анализа исходных кодов ядра Linux. **http://lxr.free-electrons.com/source/**, **http://lxr.linux.no/**, **http://lxr.missinglinkelectronics.com/linux**, **http://lxr.oss.org.cn/**.

46] How to get printk format specifiers right. Randy Dunlap (**rdunlap@infradead.org**), Andrew Murray (**amurray@mpc-data.co.uk**). **https://www.kernel.org/doc/Documentation/printk-formats.txt**.

47. Модули ядра Linux. **https://hackware.ru/?p=12514**.

48. Эволюция системных вызовов архитектуры *x*86. **https://habr.com/ru/post/347596/**.

49. Linux Kernel Programming: A comprehensive guide to kernel internals, writing kernel modules, and kernel synchronization. — 1st ed., by Kaiwan N. Billimoria, Kindle Edition. — March 19 2021. — ISBN-13 978-1789953435, ISBN-10 178995343X. **https://www.amazon.com/gp/product/B07RW915K4**.

50. Linux Kernel Programming. Part 2. Char Device Drivers and Kernel Synchronization: Create user-kernel interfaces, work with peripheral I/O, and handle hardware interrupts, by Kaiwan N. Billimoria, Kindle Edition. — 19 March 2021. — ISBN-13 978-1801079518, ISBN-10 180107951X. **https://www.amazon.com/gp/product/B08ZSV58G8**.

51. Мониторинг и настройка сетевого стека Linux: получение данных. Перевод AloneCoder, 21.11.2016: **https://habr.com/ru/company/vk/blog/314168/**.

52. Kernel module signing facility (о подписывании модулей ядра). **https://www.kernel.org/doc/html/v4.10/admin-guide/module-signing.html**.

53. Используйте FUSE для написания собственной файловой системы под Linux. **https://russianblogs.com/article/352517574/**.

54. Code Ninja. Создание файловой системы с помощью FUSE, 2016. **http://rus-linux.net/MyLDP/algol/code_ninja_make_a_filesystem_with_fuse.html**.

55. Типы стандартов USB и разница между ними. **https://club.dns-shop.ru/blog/t-345-interfeisyi-i-kommutatsiya/27840-tipyi-standartov-usb-i-raznitsa-mejdu-nimi**.

56. GPIO Programming: Using the sysfs Interface, By Jeff Tranter. — 2019. **https://www.ics.com/blog/gpio-programming-using-sysfs-interface**.

57. Linux: кнопки, светодиоды и GPIO. — 2019. **https://www.pvsm.ru/linux/69220**.

58. Control GPIO using the new Linux user space GPIO API. **https://blog.lxsang.me/post/id/33**.

59. General Purpose Input/Output (GPIO). **https://www.kernel.org/doc/html/v4.17/driver-api/gpio/index.html**.

Предметный указатель

С

Т

У

Ф

Х

Ш

Э

Я

Об авторе

Цилюрик Олег Иванович — практик-разработчик архитектуры (системотехники) и главным образом программного обеспечения для крупных компьютерных проектов, со стажем практических разработок больше 40 лет. Еще в последнее десятилетие существования СССР участвовал в больших проектах в области ВПК СССР в крупнейших инженерно-научных центрах: ВНИИ РТ и КБ ПМ (Москва).

С того времени непрерывно работал в самых разнообразных разработческих проектах. Как обобщение этого опыта издал в разное время три книги в центральных издательствах Москвы и Санкт-Петербурга и около 20 книг в электронном виде, достаточно известных в программистских кругах. Параллельно работал 5–6 лет научным редактором переводной компьютерной литературы издательства «Символ-Плюс» — хорошо известная серия компьютерной литературы «со зверушками на обложке».

Материал этой книги накапливался и откладывался с разной интенсивностью на протяжении больше 11 лет. Упорядочение и последняя его редакция формировалась летом 2022 года в пригороде города Харькова, в самый разгар боевых действий… Когда казалось полной бессмыслицей писать тексты на такие отвлеченные темы. Но автору хочется надеяться, что этот текст окажется востребованным молодым поколением инженеров-разработчиков, которые начнут новый виток технологического развития.